LIBRARY CLEARANCE

1. (a) I give my permission for my thesis, entitled

....Concurrent...Pascal:...Implementation...and...Usage

.....

.....

to be made available to readers in the Library under the conditions determined by the Librarian.

(b) I agree to my thesis, if asked for by another institution, being sent away on temporary loan under conditions determined by the Librarian.

(c) I also agree that my thesis may be copied for Library use.

2. I do not wish my thesis, entitled

.....

.....

.....

to be made available to readers or to be sent to other institutions without my written consent within the next two years.

Signed ....MHenders.... MARTIN L HENDER

Date ....20/9/83....

Strike out the sentence or phrase which does not apply.

The Library
Massey University
Palmerston North, N.Z.

The copyright of this thesis belongs to the author. Readers must sign their name in the space below to show that they recognise this. They are asked to add their permanent address.

Name and Address Date

.....

.....

.....

.....

.....

CONCURRENT PASCAL:

IMPLEMENTATION AND USAGE

by

Martin L. Hender

A thesis presented in fulfilment

of the requirement for the degree of

Master of Science in Computer Science

at

Massey University

September 1983.

ACKNOWLEDGEMENTS

In presenting this thesis I would like to take this opportunity to express my thanks to the following people:

First and foremost to my supervisor, Ray Kemp, whose continued guidance, criticism and encouragement throughout this project has been most helpful and much appreciated.

Secondly to Dr. Steve Black for his thorough reading of this thesis and the many suggested changes. His help has definitely improved the presentation of this work.

Lastly I would like to thank the staff of the Massey University Computer Centre for their encouragement and interest in this project.

Massey University — Martin L. Hender

September 1983

Table of Contents

Chapter 0

Introduction

In the forty years that the modern electronic computer has existed it has gone through several phases. The third generation computer of today is far removed in sophistication from the early first generation machines. In just the same way as the hardware of these machines has changed or been enhanced through the years, so has the associated software on them. The early computers required the programmer to use machine language or even set switches on the machine, while today the programmer can often use one of several high level programming languages.

Whereas the early machines were used mainly for number crunching, the computer of today is asked to perform a multitude of tasks. To cater for these wide and varied applications the programming languages of today have themselves divided into several areas. Each area designed to support a certain application. There are languages to support numeric scientific problems (e.g. Fortran), data processing processing problems (e.g. Cobol) and list processing (e.g. Lisp) to name but a few. In the course of this thesis one of these applications, that of implementing operating systems, will be covered along with the corresponding languages that have been developed to support it.

Systems languages are relatively new in the software scene with them first appearing around ten years ago. This project will be concentrating on one of these languages, Concurrent Pascal [Brinch Hansen 75], a forerunner in the field. An operating system for the IBM Series/1 has been written in the language and

through this implementation an insight into the power and usefulness of the language has been gained. Brinch Hansen, the designer of Concurrent Pascal, has pointed out that "the language will no doubt have deficiencies" [Brinch Hansen 77b] and this implementation has hopefully brought these weaknesses to the surface. A discussion of facilities offered by other languages in the area is included to compliment those offered by Concurrent Pascal.

Chapter one describes the facilities of Concurrent Pascal that were designed for the systems programming field. Three such facilities are studied in detail, the class, the process and the monitor. It is suggested that the process is the main feature which separates the systems language from others. In the operating system of today several tasks commonly have to be performed simultaneously. The process or an equivalent mechanism is the way a programmer achieves this required concurrency. With several processes working independently though, there is a need in some cases to synchronize them. To support this Concurrent Pascal uses the monitor construct, although other system languages may use totally different mechanisms. The systems type languages are therefore themselves divided into sections depending upon the synchronizing mechanism they use.

In chapter two the implementation of the language Concurrent Pascal on an IBM Series/1 is described. Several problems concerned with portability were found in this part of the project. Through these difficulties several lessons have been learned in respect to producing machine independent and transportable code. It is suggested that problems will always be

encountered when moving symbolic code, or even program source code, from one computer to another. The aim must be therefore to recognise this and try to minimise these difficulties. In doing so the task of transporting the object is made far easier as less effort is required to support it on the new machine.

The implementation of a small interactive operating system (i.e. MUSCLE) is described in chapter three. This implementation has shown that Concurrent Pascal can be used to write simple general purpose operating systems. It is hoped that the implementation has been carried out in such a manner that given appropriate hardware it could easily be extended to a fully fledged general purpose system. It has however been sprinkled with difficulties which cannot entirely been blamed on the hardware or machine used. It is suggested that some of these problems are the result of shortcomings in the language. These shortcomings have been discussed and where appropriate alternatives have been suggested.

Chapter four contains some suggestions as to what properties a systems language must contain. Whereas Concurrent Pascal can be considered to be a small and very restrictive language it is suggested that this causes difficulties to the programmer. The writing of an operating system is recognised as being one of the most difficult programming tasks. The language which is used must therefore aid the programmer as much as possible. It can do this in two ways, firstly by offering a large number of facilities and secondly by relaxing restrictions on the use of certain constructs. Concurrent Pascal was found wanting in both these areas. It is suggested that the systems language must

assume that the programmer knows what he is doing and in some cases allow certain restrictions to be relaxed. Any relaxation though must be done at the specific request of the programmer. This ensures that the programmer knows that liberties are being taken and that it is obvious in the program.

Chapter 1

The Language

Concurrent Pascal

1 Introduction to Concurrent Pascal.

Most programming tasks are sequential in nature. The term sequential in this context means that the task that is to be programmed can be specified by an ordered set or sequence of steps. This has meant that the majority of programming languages that have been developed take advantage of this fact and are themselves totally sequential in that only one task can be done at any one time.

On the other hand, present day operating systems generally support several independent users and user programs whose demands on a system cannot be known in advance. This is one reason why these systems are so complicated to design and implement. The developers have to try and cater for all these individual demands in the best and most efficient way they can. In order to accommodate these unexpected demands the operating system must be concurrent in nature. The term concurrent means that the system must be able to do several tasks at the same time. Since the actual computer is very restricted in the number of tasks it can perform simultaneously, the kernel must switch the processor between these various tasks to give the impression they are running concurrently. The kernel is the lowest level of program on the machine. Its primary task is to share the processor among all the active users.

As the need for concurrent operating systems has grown, a lot of work has been done to provide languages with which to program these concurrent situations. Concurrent Pascal is the result of one such experiment and was developed by Per Brinch Hansen while at the California Institute of Technology. It is based on a subset of the sequential language Pascal. This subset, which is lacking both the file and pointer types of Pascal, has been extended with two concurrent programming tools, the Process and the Monitor. In addition to these two module types Concurrent Pascal makes use of a third module type, the Class. Following is a discussion of what these extensions are, and how they may be used.

1.1 The Class.

The class is probably the main building block of Concurrent Pascal. It is different from the process and the monitor in that it isn't a tool for concurrent programming. But both the monitor and the process can be thought of as being special classes from the structuring point of view. The function of the class is as a tool for encapsulating both data and procedural code into a structure. Pascal has the facility to define structures of data in the form of arrays and records, and structures of code in the shape of functions and procedures. Where it is lacking though, is that there is no method to define a structure containing both code and data. The class is such an extension designed to allow this and therefore afford the structures of data a certain amount of protection by restricting the code that may access this data.

The class is composed of three main parts, the permanent data, the class procedural code and the initialisation code. The class

data is that data which the class is protecting from outside influences. Class data is permanent in that its value is maintained from one call of the class to the next. The procedural code is the code through which the class data is manipulated and can be composed of both procedures and functions. When a routine within a class is labelled of type ENTRY then it is accessible from outside the class. It is through these entry routines that code outside the class access the class data. The initialisation code, as the name suggests, is the code that initialises the class data.

One structure that is commonly used in programming is that of the stack. The stack is a list structure which stores elements and then returns them in a last in first out order. This ordering means that the last element inserted into the stack will be the first element able to be removed.

```
VAR  STACKPTR: STACK_RANGE;
     STACK   : ARRAY[STACK_RANGE] OF STACK_ITEM;


PROCEDURE POP(VAR ITEM: STACK_ITEM);
BEGIN
  ITEM := STACK[STACKPTR];

  STACKPTR := PRED(STACKPTR);
END (* Pop *);

PROCEDURE PUSH(ITEM: STACK_ITEM);
BEGIN
  STACKPTR := SUCC(STACKPTR);

  STACK[STACKPTR] := ITEM;
END (* Push *);
```

Figure 1.1

The piece of Pascal code in figure 1.1 illustrates how a stack might be implemented within standard Pascal. It should be noted that this code contains only the stack handling logic and the

checks for stack exception conditions (i.e. overflow and underflow) are missing. The stack should only be referenced through the procedures 'PUSH' and 'POP', but there is no method in standard Pascal to prohibit the external use of either the STACK, or the STACKPTR. For example the statement

```
STACKPTR := -20;
```

could appear almost anywhere in a Pascal program but would be disasterous for the stack.

The class structure of Concurrent Pascal is what can enforce the proper use of the stack as is shown by the section of Concurrent Pascal source in figure 1.2.

```
TYPE STACKTYPE = CLASS

  VAR STACKPTR: STACK_RANGE;
      STACK   : ARRAY[STACK_RANGE] OF STACK_ITEM;


PROCEDURE ENTRY POP(VAR ITEM: STACK_ITEM);
BEGIN
  ITEM := STACK[STACKPTR];

  STACKPTR := PRED(STACKPTR)
END (* Pop *);

PROCEDURE ENTRY PUSH(ITEM: STACK_ITEM);
BEGIN
  STACKPTR := SUCC(STACKPTR);

  STACK[STACKPTR] := ITEM
END (* Push *);

BEGIN
  STACKPTR := 0
END (* of Stack Type *)
```

Figure 1.2

In this example both the STACK and the STACKPTR are enveloped in this class and are therefore not known outside of it. This means that they cannot therefore be corrupted from any external source

(i.e. The statement STACKPTR := -20; is illegal outside the class). The two procedures have been defined as type 'ENTRY'. This means that they are known outside the class and so may be called from an external routine. In this way the external routine has access to the stack but only through these two procedures.

Notice that the class definition starts with the word TYPE. This is very similar to the TYPE declaration of standard Pascal where the defined types are merely templates and not entities within themselves. The specific class entity is defined in just the same way as a normal Pascal variable, through the VAR declaration as is shown below.

```
VAR STACK: STACKTYPE;
```

Here the variable 'STACK' represents an instance of the STACKTYPE class and would therefore allow a process having made such a declaration to make the following two calls.

```
STACK.PUSH(ITEM);    STACK.POP(ITEM);
```

Before the process makes either of these two calls though, the class data must be initialised by the following statement.

```
INIT STACK;
```

In the case of the STACKTYPE class, this causes the variable STACKPTR to be initialised to zero.

As well as being the building block of Concurrent Pascal the class structure is a very important part of the language because it is the only method by which processes may share code. In a

sequential Pascal program should the same section of code occur in several places then it is generally placed in a procedure which is called from each of the places where that section of code was originally used. This generally has the effect of making the program smaller and more easily understood. Likewise in Concurrent Pascal should a section of code occur in multiple places within a module then it may be grouped into a procedure within that module. This sharing of code though, cannot be done between separate modules (i.e. classes, monitors or processes) other than by placing the common code within a class structure. This restriction was imposed by Brinch Hansen to force programmers using the language to modularize their programs. The intended goal being to make the programs far more easily understood.

1.2 The Process.

The process is the mechanism by which the language obtains its concurrent properties. The process is itself a sequential program and when initiated will perform its designated task in a strict step-wise manner. Within the full Concurrent Pascal program though, there will be several of these processes each running independently and at the same time thus giving the concurrency required in an operating system. An analogy to processes in a program, is that of cars on a network of roads. Each car on the network can go at its own speed and in whatever direction it so desires. In much the same way, processes execute at their own speeds completely independent of outside influences.

To illustrate how this concurrency can be utilized in the operating system environment, consider the case of a machine with

two devices attached, a card reader and a line printer. The section of Concurrent Pascal source in figure 1.3 is an example of how these devices would be supported.

```
TYPE CARDPROCESS = PROCESS

BEGIN (* of Card Process *)
  CYCLE
    <read a card>
  END
END;  (* of Card Process *)


TYPE PRINTERPROCESS = PROCESS

BEGIN (* of Printer Process *)
  CYCLE
    <print a line>
  END
END;  (* of Printer Process *)


  VAR  READER: CARDPROCESS;
       WRITER: PRINTERPROCESS;

BEGIN (* of Main Line *)
  INIT READER, WRITER;
END.  (* of Program *)
```

Figure 1.3

The process definition CARDPROCESS contains the logic to enable the program to read a card from a card reader, this task being one that can be performed by a specific sequence of steps. The second process definition PRINTERPROCESS contains the logic to print a line at a printer, which is like the reading of a card, another strictly sequential task. In this example though, the thing that cannot be determined by the system implementer is when a card will be read or a line is to be printed. For that reason they have been coded as two independent processes which can work simultaneously but at their own natural speeds.

In exactly the same way as with the class, the VAR declaration in this example specifies the actual instances of the processes. In this case there is one instance of each of the process types in the program corresponding to each of the actual devices.

The INIT statement is the method by which a process is initiated or started. This statement is very much like a procedure call in standard Pascal in that execution of the process commences at this point. The difference is that whereas the code following the procedure call is not executed until after the procedure terminates, the code following a process initiation continues to be executed in parallel with the process code. Also unlike the procedure, when a process terminates, control is not passed back to the caller. In this case the process just dies and the level of concurrency is reduced by one in the total system.

The CYCLE loop construct used in this example is a extension to standard Pascal that is supported by Concurrent Pascal. This construct allows the coding of endless loops and is semantically equivalent to the statement "WHILE TRUE DO BEGIN". Although this construct is not commonly desired in ordinary programming tasks it is very handy in operating systems, these systems being designed to execute indefinitely.

1.3 The Monitor.

For discussing the monitor we will continue the analogy of processes in a program being like cars on a system of highways. Although each vehicle can travel at its own speed and in any direction it so chooses, for vehicles travelling on a certain road the overall performance of any one car is governed by the

presence and speed of all other vehicles on or crossing that road. Any car that travels with no regard for the others on the road will be a hazard and will probably cause an accident. In just the same way, although processes can work independently at their own speed, those aiming for a common goal need to be influenced by each other. The mechanism by which they can co-ordinate each other is, within Concurrent Pascal, the monitor.

This can be illustrated by taking the previous example and imposing the restriction that the printer shall print the information that was read in from the cards. In this case a monitor will be used to synchronize the two processes and ensure that after each card is read the corresponding data will be printed.

```
TYPE LINEBUFFER = MONITOR

PROCEDURE ENTRY READ
BEGIN
  <wait for line to arrive>

  <take line>
END;

PROCEDURE ENTRY WRITE
BEGIN
  <wait until last line has been taken>

  <deposit line with monitor>
END;

BEGIN
END;
```

Figure 1.4

Figure 1.4 contains an example of such a monitor written in Concurrent Pascal. The procedure READ will in this case be used by the printer process to get the line that must be printed. The corresponding printer logic will now be as follows:

```
CYCLE
  LINE.READ;  (* Get the line before printing it *)

  <print the line>
END;
```

where LINE is a declared instance of the monitor LINEBUFFER. The procedure WRITE in the monitor will be used by the card process to pass the card image along to the printer and its section of logic can now be presented as follows:

```
CYCLE
  <read a card>

  LINE.WRITE;  (* Pass card image on via monitor *)
END;
```

The reader may have realized that with the above restriction the example has appeared to turn into a sequential one, which could be represented by the following piece of code:

```
CYCLE
  <read a card>

  <print the card image>
END;
```

Although apparently equivalent this sequential solution when compared with the concurrent model suffers from the performance point of view. In the concurrent model the READER process can be reading the next line while the WRITER process is still handling the current one. This means that the system actually travels at the speed of the slower process. In the sequential model on the other hand, the reading of the next line cannot commence until after the current one has been written out. The speed of this system is therefore dependent upon the speed of both the reading and the writing cycle. When these cycle speeds are approximately

equal the speed of the sequential solution will only be about half that of the concurrent model. This lessening of performance is the main reason why the sequential solution is out of the question in operating systems. It is only with the concurrent model that I/O can effectively be overlapped with processing, thereby achieving the required gains in performance.

The mechanism by which a monitor co-ordinates processes is very similar to how traffic lights control traffic. Just as the traffic signals stop and start the flow of traffic, so the monitor has power to stop and start processes.

```
TYPE LINEBUFFER = MONITOR

  VAR BUFFER_FULL: BOOLEAN;
      READER     : QUEUE;
      WRITER     : QUEUE;

PROCEDURE ENTRY READ
BEGIN
  IF NOT BUFFER_FULL
  THEN DELAY(READER);

  <take line>

  BUFFER_FULL := FALSE;
  CONTINUE(WRITER)
END (* Read *);

PROCEDURE ENTRY WRITE
BEGIN
  IF BUFFER_FULL
  THEN DELAY(WRITER);

  <deposit line with monitor>

  BUFFER_FULL := TRUE;
  CONTINUE(READER)
END (* Writer *);

BEGIN
  BUFFER_FULL := FALSE;
END (* Line Type *);
```

Figure 1.5

To illustrate this mechanism the monitor 'LINEBUFFER' depicted in figure 1.4 will be expanded in figure 1.5 to show how the prospective processes are made to wait. In the READ procedure, the process must wait until the line has been deposited with the monitor. The first test sees if the buffer is full or not. Should the buffer be empty then the process must be stopped until it has been filled. This is achieved by the DELAY statement and in this case the process is made to wait on the READER queue. Once in this queue the reading process is delayed until the buffer is filled by the writing process. To activate a process the CONTINUE statement is used. Once past this point the reader empties the line buffer, sets the flag to false indicating the now empty buffer and then starts up any waiting writing process wishing to fill the buffer.

In the WRITE procedure, the process is delayed in the event of the buffer being already full. It then fills the buffer, marks it as being full and then continues any waiting reader. This mechanism will ensure that the reader and writer processes will alternate indefinitely. The mainline of the monitor sets the BUFFER_FULL flag to false indicating that the buffer is initially empty.

Chapter 2

Implementing Concurrent Pascal

2 Introduction.

In order to demonstrate the use of Concurrent Pascal, Brinch Hansen developed an operating system which he called SOLO written in the language. As the name suggests this is a single user system. Although it may be classified as trivial when compared to other operating systems it nevertheless was quite complex in that it provided an editing facility and supported the compilation and running of both sequential and Concurrent Pascal programs. It was initially developed on a PDP11/45 machine but was implemented in such a manner that hopefully it could be easily moved to other machines. Distribution tapes are provided from the University of Colorado at minimal cost for just this purpose.

It is generally recognised that interpretative systems are far more easily transportable than those systems producing machine code. With this in mind the SOLO system is supported on a virtual machine that consists of two programs: a Kernel and an Interpreter. The Kernel has the task of interfacing the system to the machine and handles all the I/O while the Interpreter, as the name suggests, executes or interprets the programs on the system.

In the course of this chapter, the method by which the author moved the language Concurrent Pascal to the IBM Series/1 machine will be described.

The following statement is a description of the distribution tape that was available for this implementation attempt.

> "The SOLO system (including Concurrent Pascal compiler) is distributed on a magnetic tape containing a PDP 11/45 assembly listing of the Kernel and Interpreter with Pascal-like comments together with the source, virtual code and user manuals for SOLO and its utility programs."
>
> [Powell 79]

From this excerpt it can be seen that there are essentially two methods of implementing a running version of the Concurrent Pascal compiler.

Firstly there is the method of implementing SOLO from which the compiler could be run. In this case a virtual machine, comprising the Kernel and Interpreter, would have to be written upon which the provided virtual code files could be supported. This is the method suggested by Brinch Hansen himself. The alternative method is to take the source of the compiler which is written in sequential Pascal and modify it so that it will compile and run on one of the Pascal systems available to the implementer. This option appeared to be much more work than the previous one as it required the compiler to be modified to run on a standard Pascal system as well as the implementation of a virtual machine on which to support output from the compiler.

The following sections discuss how initially an attempt to implement SOLO on an IBM Series/1 was made. When this method proved inappropriate though, the second method, that of changing the compiler, was used. Prior to these discussions though, the work of other implementations will be described.

2.1 Experience of Others.

Much of the work described in this section of transferring SOLO to another machine is directly following the effort made by M. S. Powell [Powell 79] who transferred it to the CTL Modular One machine. One difference between this project and his though, was the aim of the exercise. His sole aim was to transfer SOLO and to use it as a replacement to the "CTL supplied E4 executive and AOF operating system" which they had found highly unsuitable for the applications they had in mind. This project's aim, on the other hand, was to provide a running version of the Concurrent Pascal compiler. Should SOLO itself not prove acceptable in supporting the compiler then this avenue would be terminated.

During his work Powell found that three differences between the architectures of the Modular One and the PDP 11/45 produced problems in portability of the virtual code. These differences are discussed in the following sections.

2.1.1 Store Size.

The DEC machine had eight Kbytes(20%) more words of storage than the Modular One machine and Powell [Powell 79] found that SOLO made use of all the memory of the larger machine. To compound this problem Powell discovered that the CTL machine's instruction set was not as well suited to handling the threaded code used in the Interpreter. Threaded code is a mechanism used to increase efficiency in interpreted systems (See section 2.2). It was therefore found that both the Interpreter and the Kernel required more storage than that originally estimated.

This memory shortage was overcome by splitting the facilities of SOLO into two distinct operating systems. One allowed the user to compile his programs but was restricted in the forms of I/O that could be performed (e.g. no magnetic tape work allowed). The other supported all the I/O but limited the size of programs that could be run (e.g. no compilers could be used).

2.1.2 Memory Addressing.

The DEC machine (i.e. PDP11) has virtual memory and although the implementation didn't require virtual memory as such, it made use of the associated addressing so that every process thought that its memory was contiguous. This wasn't the case with the Modular One which has a simple segment base and limit register system. This meant that extra logic had to be incorporated into the Kernel thus aggravating the previous problem of memory size.

2.1.3 Floating Point.

The Modular One had no floating point hardware which meant that real arithmetic couldn't be supported within the system. Fortunately real arithmetic is only used in three places in the entire SOLO system and so Powell was able to change those routines to make use of integer arithmetic only. These changes, however, had to be coded by hand with the actual binary code being patched.

2.2 This Implementation of SOLO.

> "The system can be moved to another computer by rewriting an assembly language program, called the system Kernel, that simulates a virtual machine and its peripherals."
>
> [Concurrent Pascal Distribution Notes]

As mentioned above this method of implementation appeared the most logical course to follow from the start for the following reasons.

Firstly because there appeared to be less work as only two programs needed to be written. These two programs are the Interpreter and Kernel which would probably have had to be written no matter which course was followed. Although both had to be written in assembler the logic seemed very straightforward and mostly able to be transferred from the listings of the PDP11/45 version supplied with the tape.

Secondly SOLO itself could prove to be an ideal tool in the implementation of any further operating system. This is because as well as supporting both the Concurrent and Sequential Pascal compilers it provided the ability to run and thereby test any new operating system written in the language.

The third and final reason as to why this method was initially attempted was the result of having seen the source for the compilers. Although they were supposedly written in the language Pascal, Brinch Hansen and his team had taken large liberties with the language. The syntax had been changed to such an extent that it was expected to be a large task for any person wishing to support the compilers using a standard Pascal system.

The initial work in this implementation was the coding of the Interpreter and the Kernel. As was initially thought, these two programs proved very straightforward to implement. When testing was commenced however it became obvious that the differences between the two machine architectures were, as in the experience of Powell, going to create problems. These problems lay in two distinct areas, the first is the format of the virtual code and the second is the difference in character sets between the PDP11 and the Series/1. These problem areas are discussed in the following two sections.

2.2.1 Format of Virtual Code.

The virtual code is essentially a binary file of 16 bit integers. Unfortunately when the distribution tape was read, it was found that the two bytes making up these integers had been interchanged. For example the value one in the file appeared as 0100(hex) which is interpreted as 256(dec) on the Series/1. The reason for this swapping of bytes is that byte accessing (such as is used by the magnetic tape drivers) on the PDP11 takes the right hand byte first and then the left one. For example the string 'STRING' has to be stored as 'TSIRGN' on the PDP11.

There appeared to be two solutions to this problem. The first was to change the Interpreter so that when the word was accessed its bytes were swapped. The second solution was to change the actual code files so that they were in the format required by the Series/1. The first method was judged infeasible from an efficiency point of view as the effort required to swap the two bytes, although requiring only one machine instruction, would almost halve the speed of the Interpreter. The second method was

therefore utilized which provided no runtime overhead but required all code files, of which there are forty six of them, to be placed through a translator. It should be noted that the swapping of the bytes changed strings like 'TSIRGN' into 'STRING'. This is the required form on the Series/1. The translator therefore had to be able to differentiate between the integer and the string data and only change the integer data.

2.2.2 Character Set.

The second problem lay in the fact that the virtual code was created on an ASCII machine so all embedded character data was in ASCII whereas the Series/1 is an EBCDIC based machine and so expects all its data in the EBCDIC character set. The problems created by the difference in character sets may be divided into the four distinct areas. Were the code to be patched (as was done for the integer representation) then these four differences would probably have to be handled individually.

a. The String Area.

e.g. writeln('this is a string');

In this case all strings that occurred in the program are placed at the end of the program code in what is termed the string area. This area is quickly recognisable from the virtual code (i.e. 16 bit integers) and so the translator used above to swap the bytes of the 16 bit integers could just as easily translate the ASCII code in this area to the EBCDIC equivalent.

b. Single Character Constants.

e.g. Var ch: char; ch := ' ';

These single character occurrences are situated inline in the program code and although easy to fix they have had to be fixed individually by hand. This is because the load constant operation in the virtual machine does not have an associated type with the constant and so no automated translator could foreseeably know whether a character or an integer was being loaded. The number of occurrences of these constants within the programs making up SOLO was not that large to rule out hand patching of the code.

c. The Case Statement Construct.

```
e.g. Var ch: char;

     case ch of
       ...
     end;
```

In this case a branch table is produced in the code with each entry being positioned according to the ordinal number of the character within the ASCII character set. As in the previous case if this piece of code were to be patched it would have to be done manually. Unfortunately, in this case it would not be as straightforward and probably very error prone. The table would have to be lengthened in the process of altering it in order to accommodate the longer eight bit EBCDIC code. This would require the finding and patching of all branches that jumped around the case statement which is where the errors, if any, would probably be introduced.

d. Sets.

e.g. Set of char;

If the virtual code were to be changed to fix the problem of sets then several major changes would have to be made. These changes are mainly the result of the EBCDIC character set being longer than the ASCII one. All sets would have to be given twice the storage they previously required. This would upset all the locations of variables that followed a set variable in a given block. It is then necessary to modify the references of all such variables in the code to reflect their new position relative to some base register.

Whereas it was relatively easy to change the virtual code files to reverse the bytes of integers it was decided that such patching of the code to fully support the resident EBCDIC code would not be possible. The Interpreter and Kernel were therefore changed so that the system would execute using ASCII code and translate to and from EBCDIC during input-output.

2.3 Insights gained through SOLO implementation.

The problems that Powell reported were found to a lesser degree in this implementation attempt. However, these problems were far outweighed by problems due to differences in the character and integer representations. The reason Powell's problems were not paramount is probably due to the Series/1 possessing floating point hardware and a more sophisticated memory addressing system. In the same way the problems encountered in this implementation attempt were not even mentioned by Powell. Powell avoided

several major problems because the CTL Modular One, like the PDP11, is an ASCII based machine. It seems likely that transferring an operating system like SOLO to a third machine would uncover other problems peculiar to the architecture of that new machine.

This implementation of SOLO was curtailed at a stage where both the Kernel and Interpreter were completed and all that was required was to create a complete copy of the SOLO disk. The SOLO disk is the formatted file system stored on disk. The distribution tape does not come with an image of the disk and so it would have had to be created from scratch. It had become apparent by this stage though, that SOLO was not going to be a useful tool for the implementation of further operating systems. This conclusion was based on the assumption that any operating system written using SOLO as a tool would probably inherit the undesirable properties of SOLO. These undesirable properties being the large size and slowness of SOLO.

SOLO is slow because it is interpreted. It can therefore only go as fast as its interpreter can drive it. Realizing this, Brinch Hansen tried to make the code as efficient as possible by the use of threaded code to link virtual machine instructions. The threaded code reduces the overhead in going from one instruction to the next. On the PDP11 threaded code allowed Brinch Hansen to link between virtual instructions using only one machine instruction. This instruction is as follows:

```
MOV   @(Q)+,PC
```

In contrast, the Series/1 requires at least two machine instructions to accomplish the link between virtual instructions. This doubles the time required to chain from one instruction to the next. The two Series/1 instructions are as follows:

```
MVW   (Q)+,PC
BXS   (PC)        PC is a defined register
```

It is important to note that although the Interpreter was designed to be as efficient as possible its performance was severely hampered by the quality of the virtual code being executed. In general, multiple pass compilers like that used for Concurrent Pascal, can be very effective in producing efficient code. Unfortunately, no such optimization is performed with this compiler thus throwing away any gains in performance made by the Interpreter. For example, the following piece of code is but one case where optimization could be easily performed by the compiler.

```
LOCALADD   -160      Load initial address of record
FIELD       158      Add offset of field to address
```

could be optimized to

```
LOCALADD     -2      Load address of field
```

but unfortunately this was not done.

In much the same way as a computer's performance is fixed by a maximum rate of MIPS (million instructions per second), an interpretive system will have a limit to the number of virtual instructions it can execute in a given period. If optimization was performed the code could be supported with fewer virtual instructions. This would increase the throughput of the system and also reduce its size.

The following comment has been made of Concurrent Pascal:

> "Non-ambitious time-sharing systems supporting only a few terminals can easily be conceived for computers with sufficiently large main memory."
>
> [Graef et al. 79]

In this day and age although memory is generally considered a cheap commodity, on a machine like the Series/1 it must still be regarded as a cheap but scarce commodity. It was therefore thought that the large space required by the virtual code together with the space requirements of the Interpreter and Kernel would be another reason why this method of implementation would not be successful on this machine.

2.4 Feasibility of using Compiled Code.

The implementation of Concurrent Pascal via an interpretative system was abandoned because it was too large and slow. This raised the question: could any method of implementation be used successfully given the resources of the Series/1? Could a comparatively large system written in Concurrent Pascal be supported on the Series/1 and so enable this project to continue? Or would such implementation would be out of the question with the resources available.

The most likely possibility seemed to be the use of a Concurrent Pascal compiler which would produce Series/1 machine code. To test whether this method could hope to produce an efficient system it was decided to make a comparison of the original interpreted system and a machine code version. The comparison was of one of SOLO's utility programs 'DO', and although this

procedure was written in pure Pascal, any hypothesis gathered from this example could equally be applied to a Concurrent Pascal procedure. The interpreted code was taken from a disassembly of the code for the routine 'DO', while the machine code was obtained from the resident Pascal compiler on the Series/1. The comparison is contained in Appendix A and the results are contained in figure 2.1.

	Virtual Code Size (bytes)		Series/1 Code Size(bytes) minimum		maximum
A	10	(i)	-		-
B	10		4		6
C	24		6		6
D	14		8		10
E	14	(i)	-		-
F	16		6		6
G	24		12	(ii)	30
H	4		2		4
I	2	(i)	-		-
Total	90		38		62

(i) This value has been excluded from the Total.
(ii) This minimum value is without any array bounds checking.

Figure 2.1

It can be seen from figure 2.1 that were the procedure implemented in actual machine code it would require as little as one third the space that the virtual code requires. This exercise has shown that actual machine code can be generated which will be much smaller in size than the virtual code and also have the advantage that the Interpreter which requires at least 4K bytes is eliminated. The reader may well argue that had the generated code for the ENTER and EXIT routines been included in the table, the difference in size between the two methods would have been drastically reduced. This could well have been the case if inline code had been generated for these instructions.

There is no reason though, why re-entrant routines cannot be developed which perform these entry and exit functions and which are called from each of the individual entry and exit points. This would keep the code at these points to a minimum with almost no extra overhead.

From this and similar examples it was decided that if the compiler was modified to produce reasonably optimized code then Concurrent Pascal could be used to write a reasonable operating system. The code produced would be substantially smaller in size than the equivalent virtual code. The system could also be expected to perform much better because with machine code now being generated the operating system could be rated in real machine instructions per second instead of the much slower virtual instructions per second. The two pitfalls of the SOLO implementation, that of low speed and large size, would therefore be overcome.

2.5 Modification of Compiler.

In this section the conversion of the Concurrent Pascal compiler to run under a standard Pascal environment is described. The compiler was changed to produce Series/1 machine code rather than the virtual machine code it originally produced. The compiler is written in a dialect of the language Pascal, called Sequential Pascal. Neal and Wallentine [Neal and Wallentine 78] noted that although Sequential Pascal, differs quite significantly from the standard, it is still "close enough to Standard Pascal to allow the transportation (of the compiler source)".

The Concurrent Pascal compiler on the distribution tape was designed to run under the SOLO operating system. In order to do so, it had to fit into the rigid restrictions imposed by SOLO. These restrictions are on both code and data size. SOLO imposes a limit of just over ten thousand words for the program code space and sixteen thousand words for the data storage.

In order to meet these requirements the compiler had to be divided into seven passes. Each pass was called in turn by a small section of code which made up the mainline of the compiler. These eight pieces can actually be thought of as eight individual programs since the passes are called from the mainline through SOLO's RUN command. The RUN command is the method within the SOLO system of initiating a program. Communication between the passes is achieved primarily through the use of intermediate files. Each pass produces such a file which is then used as input by the next pass.

2.5.1 Portability of Compiler.

The Concurrent Pascal compiler could be supported on either the Series/1 or a Prime 750. Because of the reasons outlined below the Prime 750 was chosen.

1. With compiler sources exceeding nine thousand lines in total length both the editing and compiling facilities of the Series/1 would be heavily taxed.

2. The implementation would proceed faster on the larger Prime system. This would be because of easier access to terminals and processor (The Series/1 is totally dedicated to handling student programming tasks during most of the

day). The compiler available on the Prime is also faster than the one residing on the Series/1.

3. The Prime compiler supports extensions that the Series/1 compiler does not. One extension is that the underscore character('_') is allowed in identifiers. The second is that identifiers do not have to be unique in the first eight character positions as the Series/1 compiler requires. Both these extensions have been widely used in the coding of the Concurrent Pascal sources which would have needed substantial changes to be made to compile them on the Series/1.

Before any of the compiler could itself be compiled, quite an extensive amount of editing had to be performed. Much of this work was the result of the sources coming from a machine, the PDP11, with a very restricted character set containing only forty eight characters. A few of the symbols having to be changed are as follows:

e.g. (. had to be changed to [
.) had to be changed to]
@ had to be changed to ^

These changes generally proved very straightforward to perform and in most cases required only one editor command to fix all occurrences of that change in a source file.

Further problems were encountered though, through differences that in no way can be attributed to the restricted character set of the PDP11. The first of these to become apparent is the convention used for delimiting comments. Normally in Pascal comments are delimited through the use of braces

(i.e. {<comment>}). Several machines do not possess these characters in their character set and so resort to alternatives. The most common alternative system being the use of the compound symbols '(*' and '*)' to start and terminate comments respectively. Generally most compilers support this convention and there appears to be no valid reason why this implementation didn't. In not doing so, a lot of unnecessary work was created to correct these comments. This problem being made worse by the fact that comments are both started and terminated with the same symbol (i.e. the '"') therefore requiring each comment to be handled individually.

Another problem was the use of reserved words as identifiers. An example of this is the use of the reserved word FILE as a type identifier (e.g. TYPE FILE = 1..2;). Several small Pascal implementations like that of Sequential Pascal have limitations on the language that they will compile. While these limitations make certain reserved words unnecessary in Pascal, their use as identifiers is most unfortunate. These words should be avoided for the sake of compatibility with larger systems.

The last major item needing to be fixed in the sources prior to successful compilation was that of the I/O used. The problems lay mainly in the syntax of the GET and PUT procedures used to read and write the intermediate files. On the SOLO system they are defined as follows

```
PROCEDURE GET(F: FILE; P: INTEGER; VAR BLOCK: PAGE);
PROCEDURE PUT(F: FILE; P: INTEGER; BLOCK: PAGE);

  where   TYPE  PAGE = ARRAY[1..256] OF INTEGER;
```

The first parameter(F) specifies the actual file being used. The second parameter(P) specifies the relative block of the file to be read to or written from. In SOLO a file can be thought of as being made up of a number of blocks and the following analogy can be assumed.

```
FILE  =  (* Virtual *)  ARRAY[1..n] OF PAGE
```

The parameter P therefore corresponds, in analogy, to the array index of the above file definition. The use of this parameter could theoretically allow the file to be accessed in a random fashion. Fortunately this is not the case in the compiler and the files are only accessed sequentially. The third parameter specifies the buffer used to hold the diskpage. This notation although highly non-standard was easily changed to the following that is suitable in Standard Pascal.

```
        BLOCK := F^;  GET(F);      corresponding to the GET.
  and   F^ := BLOCK;  PUT(F);      corresponding to the PUT.

 where  F is declared of type FILE OF PAGE
                (NOTE: the use of FILE in its proper context)
```

With each of the compiler's eight sources now in a state where it could be compiled a decision had to be made as to how these would be connected to form the full compiler. It seemed apparent that each pass should be a procedure that is called in turn from the mainline. On the Prime these procedures could be of two forms, either internal or external. An internal procedure is one which is defined in the same source as the mainline of the program, while an external procedure is one where the procedure is defined in a different source. The external procedure form has the advantage that should only one procedure need to be modified then only that source has to be recompiled instead of the entire

program. The only disadvantage of using the external procedures was that they were implementation dependent and moving them to another machine would probably involve having to change them. It was felt though that the quicker turnaround for compiles and the easier editing due to smaller source files far outweighed the portability problem that these external procedures produced.

2.5.2 Output of Compiler.

With the conversion of the compiler having been made, the format of the output code had to be decided upon. Although the compiler would ultimately generate Series/1 machine code it was decided that the compiler should itself produce a machine independent code which could then be translated into the actual machine code of the Series/1. This decision was made for the following two reasons. Firstly, with the compiler still producing an intermediate code it would still be transportable. Secondly, having the code generation in this two step form would make the debugging of the compiler a much easier task.

For the choice of an intermediate language, a p-code [Nori et al. 76] type language was decided upon (See Appendix B). P-code is a symbolic language which has gained much popularity through its use in the transportation of Pascal. It was chosen for this implementation because it is very similar to the virtual code already produced making the conversion straightforward. Whereas the virtual code suffered from being too close to the machine, p-code is far more machine independent in that "Implementation details such as basic data types and the bit width of the basic storage unit are not specified" [Nelson 79].

Although the p-code type language is more machine independent than the virtual code there is still one area that poses a problem. The Prime, on which the Concurrent Pascal source is being compiled is an ASCII machine whereas the Series/1 uses EBCDIC. In much the same way as the virtual code had its problems when moving code to a machine with a different character set so does this form of code. Reviewing on the four areas that caused problems with the virtual code it is found that the first three now no longer pose any problems.

In the single character case the characters are now typed so that translation between the character sets can be performed with ease.

e.g. The statement CH := 'A';

produces the following code

```
<load address of CH>
LDCSC    65
COPYC
```

In the case of strings the translation will automatically be performed when the code is moved to the new machine.

e.g. The statement LINE := '****';

produces the following code

```
<load address of LINE>
LDCA     '****'
MOVE     4
```

Since strings are stored in the p-code code in their actual character representations, their translation is automatically achieved when the entire code is translated on moving it to the Series/1.

In the case of SET OF CHAR the compiler was changed to reflect the larger set size required by the EBCDIC machine. With this problem overcome the only other cause of worry was with regard to literals as is shown in the following example.

e.g. The statement CHARSET := ['A'];

produces the following code

```
<load address of CHARSET>
LDCSC   65
BSET
COPYS
```

This is the same as the first case above. The problem of sets is therefore reduced to the problem of single literal characters in the code.

The fourth and final area (case statements) is the one where a problem still arises. In this case the branch table generated on the Prime 750 will be different than that required on the Series/1. As with the virtual code this branch table would have to modified and lengthened to be correct on the Series/1. With p-code though, the problem is easier to fix as the code is symbolic and can be edited whereas the virtual code would probably have to be handled by tailored programs. The use of p-code is also easier because no branches would have to be patched. This is because all branches are to symbolic labels whose position is not altered through changes in the length of the code.

2.5.3 Changes to Compiler.

Other than changing the intermediate code to a similar symbolic p-code form, one relatively major change to the compiler was performed. This change was in the area of procedural entry and exit. In the original virtual code there were six distinct entry instructions and four separate exit instructions. These are as follows:

The entry instructions

```
BEGINCLASS      /* Enter class initialisation
BEGINMONITOR    /* Enter monitor initialisation
BEGINPROCESS    /* Start of a process
ENTER           /* Enter non-entry routine
ENTERCLASS      /* Enter class entry routine
ENTERMONITOR    /* Enter monitor entry routine
```

The exit instructions

```
ENDCLASS        /* End of class initialisation
ENDMONITOR      /* End of monitor initialisation
ENDPROCESS      /* End of a process
EXIT            /* Leave non-entry routine
EXITCLASS       /* Leave class entry routine
EXITMONITOR     /* Leave monitor entry routine
```

When the entry instructions were analysed, it was found that the BEGINCLASS, BEGINMONITOR, ENTERCLASS and ENTERMONITOR instructions had almost identical functionality. The main difference being the extra gating logic found in the monitor instructions. The gating logic of a monitor is the code that guarantees that the monitor will only be executed by, at most, one process at a time. Removing this gating logic allowed these four instructions to be assimilated into only one instruction. Likewise with the exit instructions, it was found that five of these six instructions could be reduced to one instruction. This resulted in this new p-code system having only three entry and two exit virtual instructions.

These five instructions are as follows:

The entry instructions

```
BEGINPROCESS      /* Start of a process
ENTER             /* Enter non-entry routine
ENTERENTRY        /* Enter entry type routine
```

The exit instructions

```
ENDPROCESS        /* End of a process
EXIT              /* Leave any routine
```

Removing the gating logic from the entry and exit instructions meant that it had to be specified in a separate instruction, the GATE command. This instruction has a parameter which specifies whether the monitor is being initialized, entered or left. The separating of this logic from the actual entry and exiting code has made the code easier to follow.

Another major change was in the format of the stack. In the original PDP11 version the stack went from the high addresses down to low memory. This meant that the PDP11's autodecrement addressing mode was used to stack the items and the autoincrement mode was used to unstack them (refer to PDP11 Processor Handbook). On the Series/1 though, there is only an autoincrement mode and no autodecrement mode (refer to Series/1 Processor Manual). Since the stacking operation occurs more often in the generated code than the unstacking one, it was decided to handle this with the builtin autoincrement mode. The unstacking operation would then have to be handled by two instructions. This change has meant that the stack is now reversed from the original version. It now starts at the low addresses and migrates up to high memory.

As well as the stack being reversed, the structure of it has also been changed. Originally this structure was in the form as depicted in figure 2.2a. This format though, had the following two distinct disadvantages:

1. The parameters were stored beneath the local base register in the stack. With the reversed stack, this now meant that they had to be addressed as a negative offset to that register. The Series/1 has some very powerful addressing modes but unfortunately these only apply to positive offsets from registers (refer Series/1 Processor Manual).

2. The stack had to be cut back in two stages. Firstly, it was cut back to the linkage area where the registers were re-established with values prior to the call. Secondly it had to be cut back to remove the parameters. Although these two tasks were done in the one virtual instruction the mechanism was thought to be quite inefficient.

Changing to the format shown in figure 2.2b has meant that much cleaner and more compact code has been generated. This has also resulted in some increased performance being achieved because now some of the specialized addressing capabilities of the Series/1 can be used.

These changes to the compiler seemed straightforward in theory to make, but in practise they proved to be very difficult to implement. Much of this difficulty must be attributed to the fact that the compiler has a multi-pass structure. Each pass is designed to get certain input in a specific order from the previous one. Changing certain things in one pass produced waves

in all the passes that followed. In this respect some of the changes made to the compiler produced some undesirable side effects (or bugs). These effects were in the form of incorrect code being produced. Since the modification of the compiler was outside the true scope of this thesis it was decided to fix this incorrect code rather than the compiler. Although fixing the compiler would have been the better alternative, several hours would have been needed to do so.

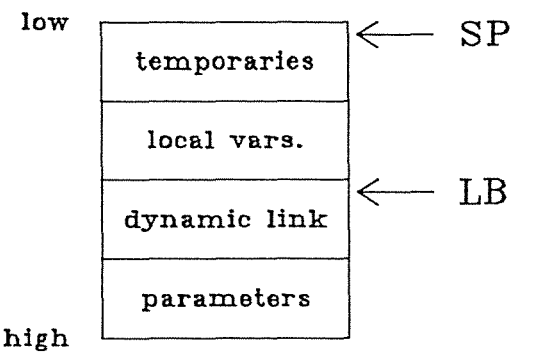


Figure 2.2a

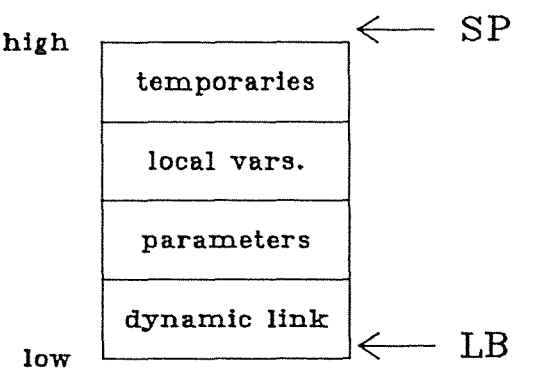


Figure 2.2b

2.5.4 The Code Translator.

The language translator is the program that converts the p-code produced by the compiler to the object code of the Series/1. Being separate from the compiler ensures that if the system were to be moved to another machine then theoretically only this program would have to be rewritten. Therefore whereas in moving SOLO the Interpreter and Kernel had to be written, here the translator and Kernel are the programs that have to be implemented. In this respect the translator has replaced the Interpreter of the original system. This isn't that surprising as both programs have a similar purpose, that of parsing and interpreting the intermediate code.

The task of the translator is essentially that of a record keeping system. The program must keep the current status of the executing environment in order to produce correct and optimized code. Since p-code is a symbolic stack machine code much of this record keeping is in the form of keeping account of the operands on the stack. Whereas in the Interpreter all stacked operands were physically stacked, in the translator these operands are only logically stacked. The actual stacking only being done when absolutely necessary. By delaying these stacking actions the translator can minimize the amount of generated code and can take full advantage of the object computer's instruction set.

In the following example the steps from source through p-code to Series/1 machine code have been shown

the statement

```
I := 1;
```

produces this p-code

```
LDLA    <offset of I>     Load address of I
LDCSI   1                 Load constant one
COPYI                     Store one in I
```

and generates this Series/1 code

```
MVWI    1,(LB,<offset of I>)
```

Although the p-code directed the translator to load both the address of I and the integer constant one, code was only generated on the COPY instruction. Optimum use of the machine's addressing capability was therefore made use of to generate only the one instruction. Had the p-code been strictly followed the following code would have been generated:

```
MVA   (LB,<offset of I>),(SP)+    Stack address of I
MVWI  1,(SP)+                     Stack constant one
ABI   -4,SP                       Cut back stack
MVW   (SP,2),(SP)*                Perform COPY
```

The differences can be readily seen. In the first case, one three word instruction was used while in the second case four instructions consuming seven words were needed.

The results of these optimizations performed by the translator are quite astounding. This being all the more surprising as these optimizations are simply register optimizations and don't involve more sophisticated techniques like data-flow analysis or loop optimization [Aho and Ullman 77]. A compiled version of the SOLO operating system, for example, occupied only three tenths of the original virtual code. The required space is further reduced because the 4 Kbyte interpreter is no longer required. If the size of the interpreter is subtracted, the compiled version is then only one quarter of the original size.

Further register optimization can easily be added to the translator although its effect must be questioned. One area where this optimization could be improved is by permitting the translator to look at the next p-code instruction (i.e. One p-code instruction lookahead). This is best explained in the following example

e.g. The statement

```
I := I + 1;
```

produces the following p-code

```
LDLA     <offset of I>     Load address of I
LDLOI    <offset of I>     Load value in I
LDCSI    1                 Load constant one
ADD I                      Add one to I
COPYI                      Store result in I
```

and generates the following Series/1 code

```
MVW    (LB,<offset of I>),W1  Load I in register
ABI    1,W1                   Add one to register
MVW    W1,(LB,<offset of I>)  Store register in I
```

The first two Series/1 instructions were generated on the receipt of the ADD instruction. Had the translator been in a position to know that the next instruction was a copy back to the variable I, then those three instructions could be replaced by the following one instruction:

```
AWI    1,(LB,<offset of I>)   Add one to location I
```

Analysis of these two pieces of code show that in many cases (where the offset of I is small) the memory size of each is identical. Further optimizations such as this were therefore never attempted as the added complexity to the translator did not seem to be offset by any substantial saving in either memory requirements or execution time. In not following these

extensions to the translator much complexity has been avoided.

Another area where this complexity was also minimized was in the generation of code. Although machine code was the target it was decided to generate assembler in the translator. This removed the difficult and often error prone task of generating actual code. A further benefit of this decision was that the output from the translator could be reviewed and verified, something very difficult to do with actual code. This has meant that another link has been placed in the compilation chain, that of the Series/1 resident assembler. The use of assembler has simplified the task of linking the Kernel code, also written in assembler, to the Concurrent Pascal code. The production of Series/1 assembler has therefore produced no repercussions other than the time required to compile the source and produce an executable code.

3 MUSCLE Implementation.

In this chapter the implementation of MUSCLE (Massey University Simple Command Language Environment) will be described. MUSCLE, a simple operating system, was written to test the effectiveness of the language Concurrent Pascal for implementing such systems. It is a multi-user interactive system which can support the editing and parsing of Pascal programs. The operating system itself does not play a big part in this project. The emphasis of the project is on how the language Concurrent Pascal may be used to produce a system such as MUSCLE. A description of MUSCLE is provided in Appendix C.

In developing systems like MUSCLE the bulk of the design effort is in three areas. The first area is the system to user interface. This interface determines how the system talks to its users. Another area is the file management system which involves both the design of the actual file structures, and that of the routines with which these structures are supported. The last area is that of the memory management system which handles the memory requirements of both the system itself and all its users. In this chapter these three areas within the MUSCLE system will be covered. This discussion will bring to light several deficiencies in Concurrent Pascal which have hindered the implementation.

3.1 User Program Interface and Organisation.

One of the first design considerations that must be made in any operating system is how it will interface with its users. The language within which the system is to be written must have a simple but efficient mechanism by which it can execute user programs. This interface mechanism must be able to perform two basic tasks. Firstly it must be able to pass control of the processor to the user program. This will occur in one of two forms: (1) when the program is first initiated, and (2) after a user program has executed some supervisor call and needs to be restarted from where it left off. Secondly it must be able to trap the user's supervisor calls and pass the request on through to the relevant operating system routine.

Unfortunately several languages designed for implementing such systems have not even considered this necessity, let alone supported it. Both Modula [Wirth 77] and Pascal Plus [Welsh and McKeag 80] fall into this category. Neither has a built in ability to call external user programs. When Welsh & McKeag [Welsh and McKeag 80] described a simple operating system written in Pascal Plus, they left the specifics of supporting user programs out and left it as a comment, as is shown below.

```
REPEAT

  { fetch, decode and execute instruction at address
    'PC', updating 'PC' and, if appropriate, setting
    'ENDOFPROGRAM' or 'ENDOFJOB' true }

UNTIL ENDOFJOB OR ENDOFPROGRAM ;
```

They suggested though, that the programs should be interpreted by the system. Thus the control of the processor never leaves the Pascal Plus environment. The interpreter now traps all problems

caused by user programs. Although this solves the problem of the system having to handle badly behaved user programs it cannot really be considered usable in general purpose systems. With respect to Modula, Wirth [Wirth 77] stated when he published it, that it was designed for small dedicated systems such as those in process control environments. In situations such as this, the entire system would be written in the language thereby removing the need and problem of calling user programs.

Fortunately though, some languages have considered the problem and have incorporated such mechanisms in their syntax. One such language is CSP/K [Holt et al. 78]. The method it uses is shown in the pseudo-PL/1 code below.

```
TRAP = CPU( <user program parameters> );
SELECT;
  WHEN (TRAP = 1) ...   /* Handle the GET request */
  WHEN (TRAP = 2) ...   /* Handle the PUT request */
  ...
END;
```

To pass control to a user program a call to the function CPU is made. When the user program executes a supervisor call the function CPU returns setting a value in the variable TRAP. The system can therefore access TRAP to determine which request the user has made. The function CPU will also return when the user program does something wrong which the hardware itself catches. In this case the value in TRAP will indicate some error condition. The parameters of CPU specify all the system parameters for the user. These include the extent of the user's stack, the current instruction address and different trap addresses that are pertinent to specific system calls.

3.1.1 Concurrent Pascal User Interface.

Fortunately like CSP/K, Concurrent Pascal also possesses such a mechanism. This is the 'PROGRAM' entry type which when called causes the processor to initiate execution of an external program. This mechanism was utilised by SOLO [Brinch Hansen 76] in the manner to be described below. The relevant section from SOLO showing this is contained in Figure 3.1.

```
TYPE   PROGFILE = CLASS

  VAR ENTRY STORE: ARRAY[1..STORELENGTH] OF PAGE;

PROCEDURE ENTRY OPEN( );
BEGIN
END (* Open *);

BEGIN
END (* Prog File *);


TYPE   JOBPROCESS  = PROCESS;

  VAR    CODE: PROGFILE;

PROGRAM JOB(PARAM: IOPARAM; STORE: PROGSTORE);
ENTRY READ, WRITE, OPEN, CLOSE;


PROCEDURE CALL( );
BEGIN
    ...
  CODE.OPEN( );
    ...
  JOB(PARAM, CODE.STORE);
END (* Call *);

PROCEDURE ENTRY READ( );
BEGIN
END (* Read *);

PROCEDURE ENTRY WRITE( );
BEGIN
END (* Write *)

BEGIN
END (* Job Process *);
```

Figure 3.1

The class 'PROGFILE' is used to read the program in from disk and to store it ready to be run. The program is stored in the entry variable 'STORE'. When the program has been loaded it is then executed by the statement

JOB(PARAM, CODE.STORE)

which passes control to the address of the second argument. The first argument is used to pass parameters to the program. Should the program want to be serviced by the operating system it will perform a supervisor call. The supervisor call automatically initiates the execution of a specific routine that was specified in the ENTRY statement. When this system routine exits the control of the processor is automatically passed back to the suspended user program, should this be desired.

This mechanism, although simple, is probably all that is required in the syntax to interface to user programs. The method used by SOLO was deemed unsatisfactory for this implementation for two reasons. The first is that the storage for the user program is set aside at compile time. This is not possible on a machine like the Series/1 where there is a shortage of memory. If the machine had a large amount of memory or supported virtual memory then this criticism wouldn't hold. Unfortunately on machines like the Series/1 all addressable memory must be physically allocated and is in short supply. The second pitfall is that the user program code is a part of the operating system data area. This imposes a large restriction on the possible size of programs able to be executed. Were the machine to have a large address space then again this criticism would be removed but with only sixteen bit addresses this just isn't the case on the Series/1.

3.1.2 Organisation of MUSCLE.

To make full use of the hardware protection on the Series/1, the memory organisation displayed in figure 3.2 was utilised.

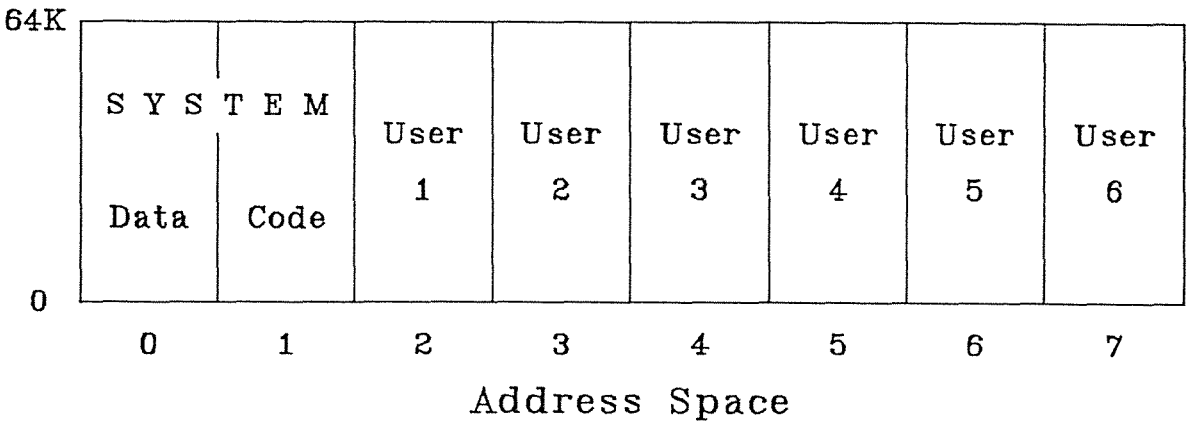


Figure 3.2

Here each user is given an address space out of which they may run. This ensures that all users are protected from one another and that the system itself is protected from corruption by the users. In SOLO the system was protected from the user by software measures. The interpreter, while executing the user program, had to check against invalid addressing through pointers. In a compiled system like MUSCLE, the protection must be hardware supported to be totally secure.

Using the structure in figure 3.2 ensures that the only restriction on size imposed on the user programs is that imposed by the machine. Of the 256 Kbytes on the machine, the operating system is given the amount of memory it requires (up to 128K). The rest is then shared among the users in a first come first served fashion. A problem arises here though in that if all users simultaneously request 64K bytes, only two could be satisfied thus leaving four requests that could not be met. This

lead to the decision that the memory must be shared as shown in figure 3.3.

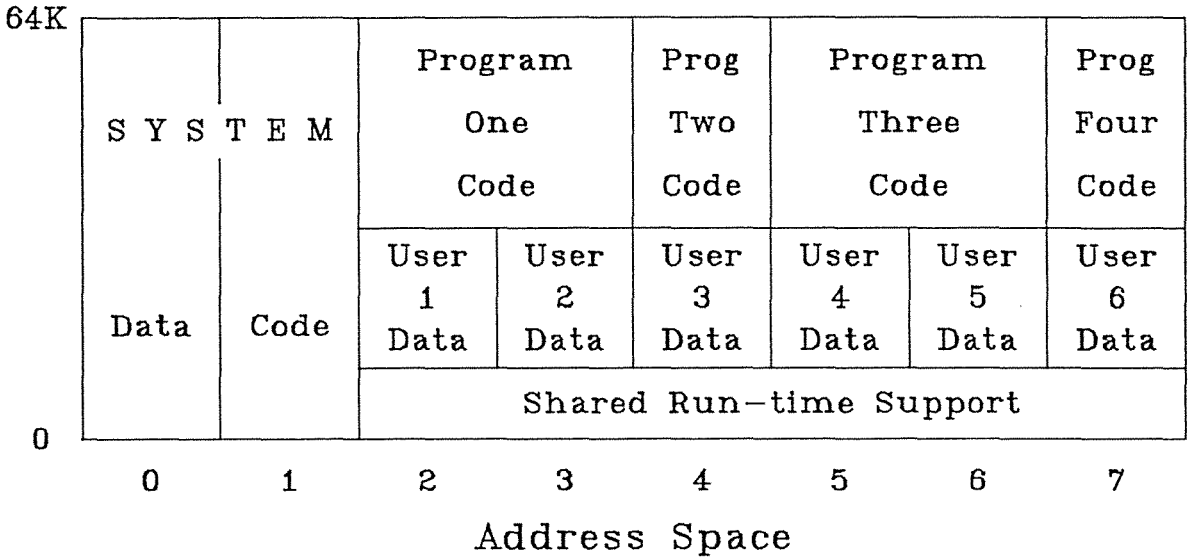


Figure 3.3

In this case the run-time support routines are loaded in at system initialisation time and shared by all users. There is therefore, only one copy in memory instead of the previous six (a saving of 10K, 2K per address space). The stack still has to be on a per-user basis as before, but the actual program code could be shared if there were two or more copies of the program being run at once. Both the support routines and program code would be write protected to prevent accidental corruption. This change resulted in two advantages:

1. It allowed the memory to go a lot further. Memory was now only being allocated for invocations of programs which were not already running.
2. It allowed programs to be loaded and saved a lot quicker. To load a program now, only the program's code and data need be read in from disk. Should the program already be

running then only the data need be read in. To save a program only the data need be stored as the code could if necessary, be restored from the original program file.

Unfortunately going to a structure where the user resides in a different address space from the system has its problems. This is mainly due to the system not having direct addressing capability of the user program. To allow the system to do this, a facility had to be provided in Concurrent Pascal which would enable the system to move data between itself and the user program. It was decided that the easiest method of achieving this within the language was by the use of the IO primitive routine. This routine is very easy to manipulate because the record type IOPARAM that it expects is user defined and may be changed to suit any call required. A new pseudo-device was therefore required and was called DATAMOVE.

DATAMOVE - This moves a block of memory across address spaces. In this case between the system dataspace and the user's program space.

The IOPARAM record for the DATAMOVE device was set up as follows

```
TYPE  IOPARAM  =   RECORD
                     DIRECTION : (TOSYSTEM, FROMSYSTEM);
                     ADDRSPACE : 2 .. 7;

                     ADDRESS   : ADDRESS_TYPE;
                     LENGTH    : INTEGER;
                   END;
```

The DIRECTION field specifies whether the block is being transferred to or from the system dataspace. ADDRSPACE determines the address space of the user program. The last two fields describe the address and length of the system buffer. To

transfer a block of text to a buffer in a system routine the following mechanism had to be used.

```
IF USER = SYSTEM
THEN BUFFER := BLOCK
ELSE DO
  PARAM.DIRECTION := TOSYSTEM;
  PARAM.USER := USER;

  IO(BLOCK, PARAM, DATAMOVE);
END;
```

The initial test (i.e. USER = SYSTEM) determines whether the transfer is between two buffers in the system dataspace or between the system and a user dataspace. If the transfer is between two system buffers then the transfer can be performed by an ordinary assignment statement. On the other hand when transferring between the system and a user the move has to be performed via the call to IO.

The need to perform this type of code to move data between the system and a user would not have had to be done in the SOLO case as the user program was directly addressable by the system. The need to do so in this case though, is not so much a criticism of the language, as a criticism of the machine's architecture. Having only a sixteen bit address space severely limits what can be done in that address space. Figure 3.4 illustrates a memory management system that would be ideal for a system language like Concurrent Pascal.

In this case the system sits at the bottom of the address space for each user, instead of being in a different address space as is the case for MUSCLE. The system code is automatically shared to minimize the amount of actual storage required. There will be

both shared and per-user data in the system. The user programs sit directly above the system thus allowing direct accessibility between the system and the user. The only requirement of this system is that the total address space must be large enough so that the inclusion of the system doesn't drastically effect the size available to the user program. This problem which is inherent to machines with small address spaces has probably been recognised by current hardware designers as the trend in modern machines is to give a large address space (in the order of megabytes).

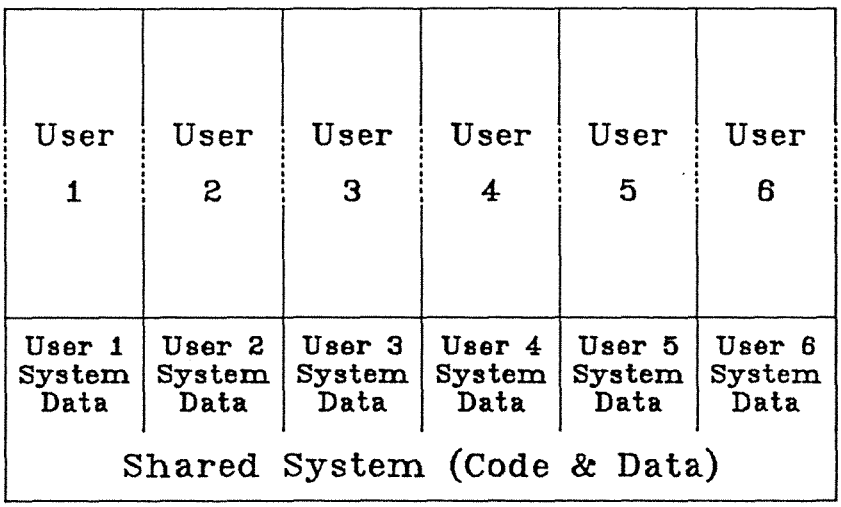


Figure 3.4

The need to move data via the IO procedure has brought to light one limitation of Concurrent Pascal. This limitation is that there is no method in the language by which an address can be obtained. This proved a problem in this case as the IO routine only allows one address to be provided, whereas two are required, the source and the destination. In order to get this mechanism to work a kluge was used. The ADDRESS field was interpreted as an offset from the global base register. This meant that the ADDRESS field was specified as an integer. The Kernel added this

offset to the base register to obtain the actual address. The value of the offset was determined by hand and then hardwired into the program as a constant. This proved a very error prone method as the offset had to be modified several times due to changes being made to the system.

What is needed in the language, is the ability to obtain and store addresses. Modula [Wirth 77] again recognised the problem and provided the ADR() function, which returns the address of the parameter in the form of an integer. A function of this type may become a problem on machines where an address is greater than the largest integer. This is because on such machines the address couldn't be represented by an integer, but on machines like the Series/1 or the PDP11 this is all that is required. Dennis Heimberger [Heimberger 78] also had this problem of requiring an address of a variable within Concurrent Pascal and he made use of the IO primitive in the following manner to obtain it.

```
IO(buffer_var, address, REAL_ADDRESS);
```

This IO call is to the pseudo-device REAL_ADDRESS. It returns the address of the variable 'buffer_var' in the variable 'address'. The type of the variable 'address' is a record which is defined by the programmer. This mechanism therefore, doesn't suffer from the possible limitations of the ADR() function. This is not as tidy a method as the ADR() function though, and suffers from the pitfall that the address is completely evaluated at runtime. Most of the calculation could be achieved at compile-time which the ADR function of Modula probably manages to do.

3.2 Memory Management System.

The memory management system of MUSCLE is unfortunately more rudimentary than was initially hoped. This is due to the Series/1 not supporting virtual memory (A discussion of how virtual memory might be supported in Concurrent Pascal is contained in section 3.5.1). The memory manager is therefore restricted to performing the following series of operations

1. Sharing the run-time support routines
2. Allocating memory for new programs
3. Releasing memory on program exit.

As has been mentioned previously, the user programs are broken up into three sections, the runtime support routines, the program's dataspace and the program's program code. The runtime support routines which contain all the Pascal I/O routines are automatically shared between all users. The program's dataspace is unique to that user, while the program area may be shared between users should they be running invocations of the same program. The status of the system with respect to memory management is maintained chiefly in the following structures:

```
TYPE   ADDRESSSPACE = ARRAY[0..MAXPAGES] OF INTEGER;
       JOBTYPE      = RECORD
                        NOINUSE: INTEGER;
                        PROG   : INTEGER;
                        PAGES  : ADDRESSSPACE
                      END;
       PROG_DATA    = RECORD
                        INUSE     : BOOLEAN;
                        FILELOADED: BOOLEAN;
                        USER_CNT  : INTEGER;
                        ADDR      : DISKPAGE;
                        PAGES     : ADDRESSSPACE
                      END;

VAR    USERS     : ARRAY[0..NOUSERS] OF JOBTYPE;
       RES_PROGS: ARRAY[0..NOUSERS] OF PROG_DATA;
```

The USERS array holds the status for all the users on the system while the RES_PROGS array holds the status for all the different programs currently running on the machine. Two invocations of the same program will use the same element of the RES_PROGS array. Since every user is running a program, there is a field in the JOBS array called PROG which is used as an index into the RES_PROGS array. The field PAGES in each array type holds the physical memory pages currently being held by that user or that program.

The following sections will describe how the primary functions of the memory management system are handled.

3.2.1 Sharing Run-Time support.

Memory is allocated for the Pascal run-time support routines at system startup. The size of these support routines is known and so their memory requirements have been hard-wired into this section of MUSCLE. The sharing of this support is performed by the routine in figure 3.5.

The procedure SHARE does three tasks. It allocates the blocks for the runtime routines. It sets the memory address registers of user one to point to these blocks. Finally it sets the PAGES field of all users to that of user one so that they also can access that memory. It can be seen that the IO routine has once again been used to call the Kernel. In this case the Kernel is called to set the machine's Memory Address Registers. The pseudo-device SETMEMORY has two parameters, the first holds the values to load into the registers and the second signifies which

set of registers to load (i.e. which user). This routine also sets these registers for one of the users. This then assures that another initialisation routine can actually load the routines into the allocated memory.

```
PROCEDURE SHARE;
  VAR  PARAM: IOPARAM;
BEGIN
  WITH JOBS[1] DO BEGIN
    FOR I := 0 TO SHAREDSIZE - 1 DO
      PAGES[I] := GETFREEBLOCK*8 + 6;

    FOR I := SHAREDSIZE TO MAXPAGES DO
      PAGES[I] := 0;

    PARAM.ARG := SYSTEM+1;
    IO(PAGES, PARAM, SETMEMORY);
  END;

  FOR I := 2 TO NOUSERS DO
    JOBS[I].PAGES := JOBS[1].PAGES;
END (* Share *);
```

Figure 3.5

3.2.2 Allocating memory for new programs.

MUSCLE recognises two types of program, the program proper and the program overlay. The program overlay is an overlay to the currently running program. When calling an overlay no stacking of the current status is done. The program proper, on the other hand, is a completely new program and when called causes the current status to be saved prior to loading the new program. When allocating memory for a program both the data and the program must be given memory. With the overlay only the program space need be allocated as it assumes that the dataspace is already allocated and in use.

The procedure that handles the request for memory has the following heading

```
PROCEDURE ENTRY ALLOCATE(    USER     : INTEGER;
                             NUMPAGES: INTEGER;
                             FILEADDR: DISKPAGE;
                             PROGONLY: BOOLEAN;
                         VAR LOADED   : BOOLEAN);
```

The first parameter specifies which user is making the request. The second specifies the number of pages that the new user program requires. FILEADDR is the address on disk where the program is stored. It is used as a unique descriptor for the program. PROGONLY determines whether the program is an overlay type program or not. The last parameter LOADED is set in the procedure ALLOCATE and returns whether the program was already in memory or not. If it has not been loaded then the calling routine will have to load it.

This procedure ALLOCATE has two important tasks. Firstly, it has to see if the program is currently being run by another user. If this is the case then it must allow this user to share the currently running version. Secondly, it has to determine how much memory the user requires and then allocate it to him. It determines whether the program is already being used by comparing the FILEADDR parameter with the ADDR field in the array RES_PROGS. The parameter LOADED is then set accordingly. The amount of new memory required is determined by analysing the variables LOADED, NUMPAGES and PROGONLY.

Although seemingly straightforward, the sharing of program memory can produce synchronization problems. The procedure ALLOCATE knows when memory has been allocated to hold a specific program,

but doesn't know when that program has been successfully loaded. Should a second process wish to make use of that same program then it will be allowed to share the same memory as the original program and told that the program has already been loaded. Were there no checks then there is the possibility that this second user could begin executing the program before the initial user has finished loading it. To guard against this problem the array RES_PROGS has a field FILELOADED which specifies whether the program has been loaded or not.

When a program is to be loaded the following code will be executed.

```
MEMORY.ALLOCATE(USER,NUMPAGES,FILEADDR,PROGONLY,LOADED);
IF NOT LOADED THEN BEGIN

  <Load the program>

  MEMORY.LOADEDPROG(FILEADDR);
END;
```

The call to ALLOCATE will set the FILELOADED variable (in the array RES_PROGS) to false. This signifies that although the memory has been allocated to the program, the program itself has not been loaded. When the program has been loaded the process calls the procedure LOADEDPROG. This procedure sets the variable FILELOADED to true, this signalling that the program has now been loaded and available for use by other users. The following section of code is contained in the procedure ALLOCATE. It is of importance to processes requesting a program that is already running in the system.

```
IF LOADED THEN
  IF NOT RES_PROGS[...].FILELOADED THEN BEGIN

    <Delay until program loaded>
  END;
```

Should they enter the ALLOCATE routine prior to the requested program being loaded then the above code will delay them until that program has been loaded. They will be continued when the first process loading the program makes the call to the procedure LOADEDPROG.

3.2.3 Releasing Memory on Program Exit.

When a user exits an overlay or program a call is once again made to the memory management system. This call serves to free the memory previously used by that module. If it was a program that was exited and not an overlay then the dataspace is returned to the free memory pool for subsequent use by others.

With respect to the program space the routine has to first check whether the program is still being used by other users. This test is made by analysing the USER_CNT field in RES_PROGS. Should no other users be using the program then the memory for the program code is also returned to the free memory pool and the program is removed from the RES_PROGS array. This is done by setting the INUSE field to false.

The call to free the memory after the program is finished is the following:

```
MEMORY.RELEASE(USER, PROGONLY);
```

The first parameter specifies the actual user involved and the

second parameter specifies whether an overlay or a program is being exited from.

3.3 File Management System.

The File management system is that part of an operating system that supports all user and system file access to secondary storage devices. It is generally recognised as being one, if not the most important piece of an operating system. This is particularly true for MUSCLE as the file system not only manages the disk, but all devices on the system. Access to these other devices is through special disk files. These files are recognised by the system as corresponding to real devices. I/O on these files therefore results in the I/O being performed on the corresponding device. This mechanism is rather like how UNIX handles its devices. It was felt that this was a useful method of handling devices as it allows the file system to control access to devices by the read/write locks associated with all files.

In this section the general structure of the File System used in MUSCLE will be described together with the quite substantial problems that were met in trying to obtain as efficient a structure as was possible. Following this, two specific areas of the File System will be covered which had their own inherent problems. These areas are the free disk space management section and the common disk buffer subsystem.

3.3.1 Structure of File System.

The internal structure of the file system is based upon the hierarchical model described by Madnick and Alsop [Madnick and Alsop 69]. The hierarchical or layered system was chosen because it lends itself to the philosophy of structured programming. In just the same way as a programmer uses a structure diagram to solve a problem y dividing it into several smaller and more easily solvable ones, the "hierarchical view of a system permits thinking about different issues at different levels, while screening out the problems arising at the other levels" [Ullman 76]. Using a high level language like Concurrent Pascal therefore, the logic for each level can be contained in a class or monitor to provide a very modular approach.

The file system as initially described by Madnick and Alsop was deemed too large and sophisticated for this implementation and so it was reduced to that depicted in figure 3.6. The original system had a six level hierarchy with the top level being termed the Access Methods (AM) level. It was this level that enabled the user to access files in different ways. For example the user could have the choice of random access versus sequential access to a file. As well as this, the AM level could support different types of files like fixed length record and the variable length record. In MUSCLE though, only one file type is supported, that of variable length lines. MUSCLE only permits sequential access to these files. This means that the AM level is not required.

The next level down is the Logical File system (LFS). The purpose of this level is to take the symbolic filename given by the user and locate the corresponding file on disk. Most of the

logic of the LFS is therefore in the area of directory searching. In MUSCLE the LFS was only called when the user wanted to either open a file, test if a file exists or else delete a file.

After the LFS comes the Basic File System (BFS) which has the task of checking the user's access right to a file. Generally this would include security checks to ensure that the user is permitted to access the requested file. In MUSCLE the security aspect of the BFS has been left out. The task of the BFS is therefore divided into two areas. The first ensures that no two users are accessing the same file at the same time. The second checks that the user only accesses the file in the mode it was opened (e.g. doesn't try to write to file open for reading). The File Organisation Strategy Modules (FOSM) determine which physical device the operation will be made on. In some systems, files are allowed to span several different devices and device types making the task of the FOSM extremely difficult. In MUSCLE though, the files are either disk based or screen based making the task of the FOSM that much easier. Should the file be a terminal based one, the FOSM passes the request straight through to the respective Device Strategy Module (DSM). In the case of the disk based file though, the FOSM has to first determine the physical disk address involved in the operation before passing the request onto the DSM.

The DSMs, for which there is one for each device type on the system, have the task of converting the commands from the FOSM to the specific IO commands required for the device by the IOCS (I/O Control System). In MUSCLE it is the DSMs which contain the IO procedure calls as the IOCS is contained in the Concurrent Pascal

Kernel.

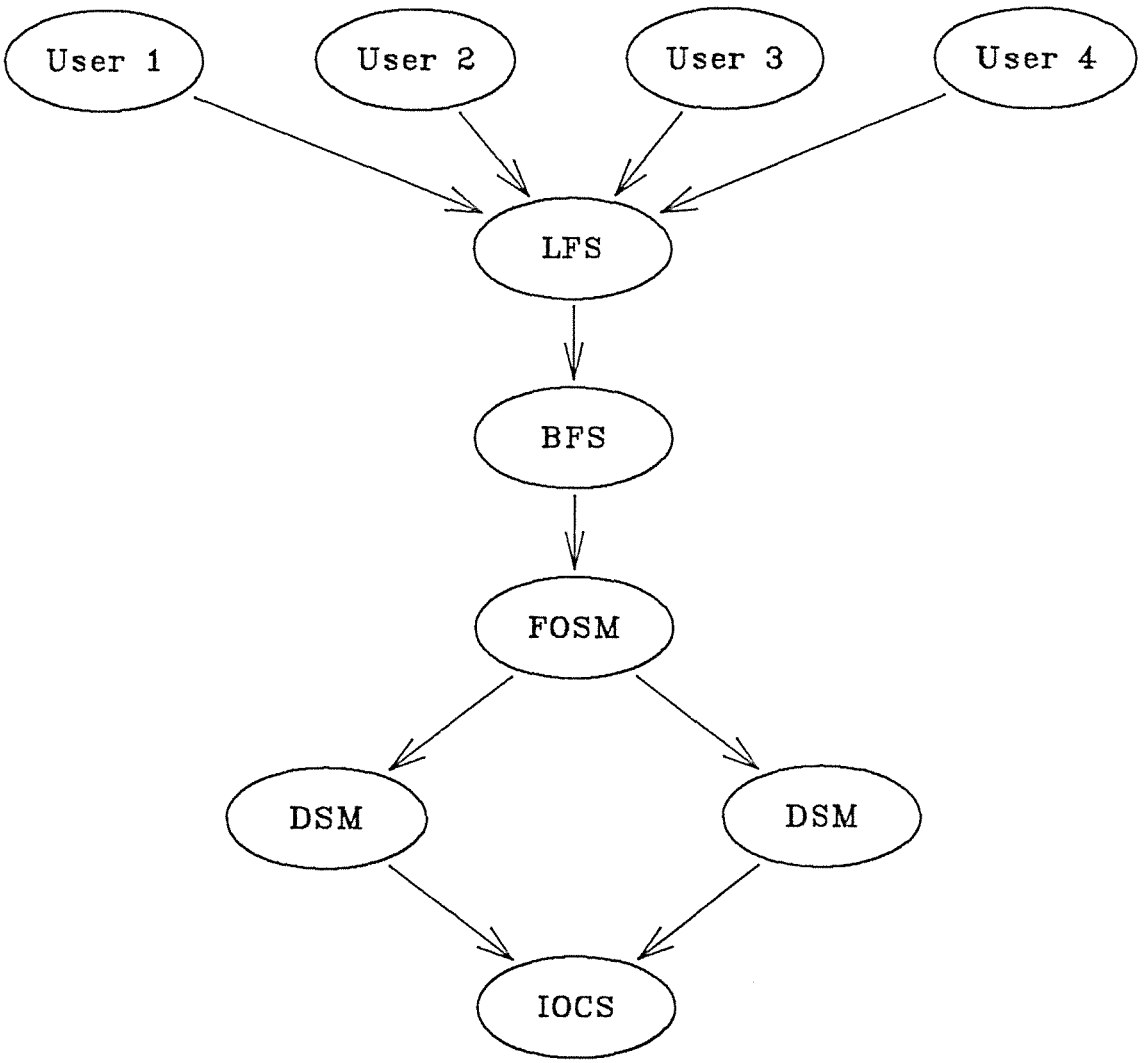


Figure 3.6

From figure 3.6 and the description above it can be seen that these components would most conveniently be represented by the following Concurrent Pascal structures. Both the LFS and the BFS should be monitors. The LFS must be one because it is this level that modifies the directories in the event of files being created or deleted. This is a task that should only be done by one person at a time. The BFS should also be a monitor as it holds all the information on the files which are open on the system. This data must be protected from simultaneous access by several users. The FOSM would probably be a class declared within the BFS as it doesn't contain any permanent data that could be

corrupted by several users accessing it at the same time. The DSMs would again be monitors, with one for every device on the machine. Were there to be two or more devices of the same type on the machine then these would be two instances of the same DSM monitor declaration. Since most devices on a computer can only handle one task at the time, having the DSM as a monitor ensures that only one user accesses a device at a time.

Unfortunately the representation suggested above contains at least three levels of monitors. This layering of monitors brings to the surface the one major problem associated with them, which is commonly referred to as the "Problem of Nested Monitors" [Lister 77]. For a further discussion on monitors and this problem please refer to section 4.3. In this section the only complication caused through the nesting of monitors that need be considered is that summed up in the following quote: "If monitors are nested, processes may be blocked unnecessarily." [Lagally 78].

Should two processes request I/O at around the same time on two different devices then theoretically they should be able to perform this I/O in parallel. In practise though, if the file system was structured as in figure 3.6 then one of the processes would be held up at the LFS (this being the outermost monitor) until the other process had completed its I/O. To overcome this problem the initial file system structure had to be modified to the form contained in figure 3.7.

In this representation both the LFS and the BFS have been changed from monitors to classes. Since classes cannot be shared between users there are now several copies of each whereas previously there was only one. Because it is necessary for both these levels to store common system wide data and to achieve mutual exclusion over other processes while accessing this data, a monitor was installed at each level to achieve this. Using this organisation the "Nested Monitor Problem" doesn't occur because, although one monitor is below the other one in the structure it is not nested as it is not called from the first one. A comparison to show the differences in performance between the two structures is contained in section 3.4.1. This shows that the original structure as it stood was unusable in multi-user environments like that in MUSCLE.

Although this structure appears to overcome the problems of the initial file system, it does suffer from the following two pitfalls.

1. Wastage of Memory.

   We now have several copies of both the LFS and BFS. This requires more memory in terms of dataspace as each instance of the class will have a certain amount of permanent data.

2. Movement away from Layered System.

   Originally all the logic and critical area was contained or clustered into one area, the monitor. It has now been spread into several different areas thereby demodularizing the structure. Although this is not a problem in itself, this is one area where monitors should be of most use but

in practise they cannot be used.

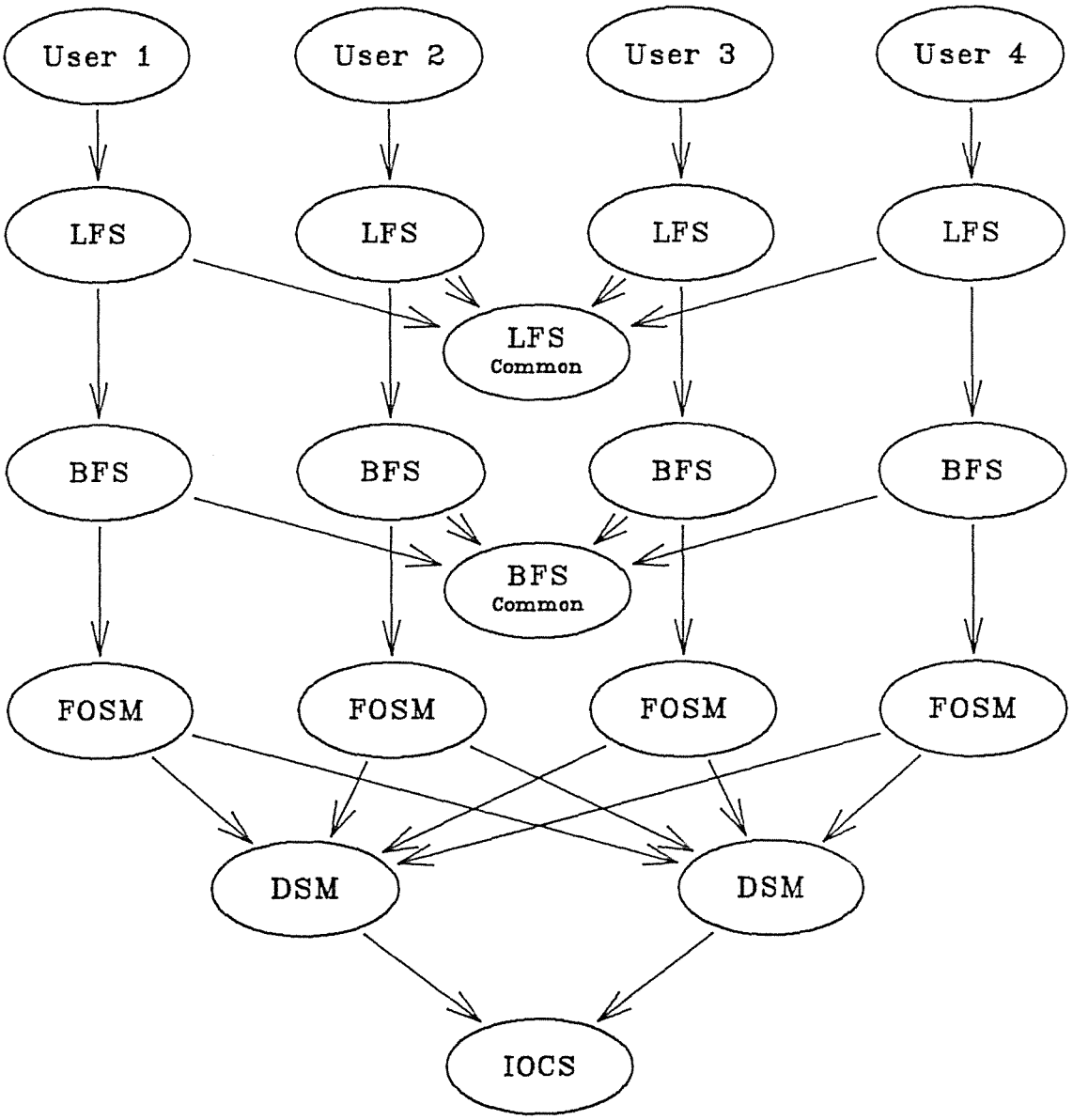


Figure 3.7

The lowest level of the structure though, doesn't suffer from the use of monitors. The DSMs exist on a one to one correspondence with the actual devices on the machine. As only one user can use a device at a time the monitor approach is particularly handy in upholding this limitation.

3.3.2 Free Block Management.

In this section the free block management system used in MUSCLE will be described. This is the part of the disk management system that maintains the free or unused parts of the disk. It can easily be seen that there are two operations that need to be supported by this system. The first is a means to procure or obtain some free disk. The second is a method whereby some disk may be returned to the free area when it is no longer required. To support this system in Concurrent Pascal the use of the monitor comes into mind as there is one resource, the free area and several users competing to use it. The initial structure of the free block manager in MUSCLE was therefore of this form:

```
TYPE  FREEBLOCKS = MONITOR

PROCEDURE ENTRY PROCURE
BEGIN
END;

PROCEDURE ENTRY DISPOSE
BEGIN
END;

BEGIN (* of Free blocks *)
END;
```

As the basic unit of the file system is the 1K byte block, this manager must maintain the free areas in terms of these blocks. It was decided to use either a link list of blocks or a bit map. The first method keeps the free area as a list of blocks forming a file. The number of blocks in the file is the number of free blocks on the device. With the bit map method though, the file only contains one bit for every page on the device. The value of the bit determines whether the page is free or not.

3.3.2.1 Link List Method.

This method has the advantage that it utilises the existing file system logic for unlinking and linking blocks into files and so no extra logic is required. For this reason therefore it was the initial method utilised in MUSCLE. Initial testing of the system was performed with the file system resident on a diskette. The slowness of the diskette, particularly in terms of seek time, quickly showed the extreme overhead that this method imposes. This overhead is due to having to link or unlink blocks from the freelist every time it is accessed.

To unlink a block from the freelist the following steps have to be performed. Assume the freelist is headed by three blocks A, B and C. The block that will be removed will be block B, as the freelist pointer referencing A should not be changed.

```
freelist  -->  A  -->  B  -->  C  -->  ...
```

```
freeblock := A -> next;              (* Saves pointer to B *)
A -> next := freeblock -> next;  (* Removes B from freelist *)
```

This sequence of steps requires two read accesses to blocks (A and B) and a modification of block A. In addition to this, block B should have its pointer set to null but this can be delayed until its chained into the required file. To unlink the block therefore two reads and one write are required and expanding this idea, it can be seen that to remove N blocks from the freelist requires (N + 1) read operations and one write. This shows then that the more blocks that can be handled at a certain time the less the overhead involved. Using this philosophy the mechanism

was changed to lessen the overhead of accessing the freelist.

In this modified version a pool of free blocks is maintained in memory. The blocks in this pool are free but are not linked into the freelist. In the event of a system crash, these blocks will be lost until some garbage collection is performed. It was felt though, that this loss of blocks was acceptable as it concerns a very small proportion of the total disk and would mean a substantial saving in overhead. When a process requires another disk block it only has to get the block pointer from the list in memory. Likewise when it releases a block it only has to add the block pointer to the list in memory.

The FREEBLOCKS monitor was therefore extended to the structure in figure 3.8. Using this structure it can be seen that the only times the freelist has to be touched is when the pool is either empty or full. This substantially cuts down the amount of overhead on the system. To handle the cases when the freelist has to be referenced it was decided to have a separate process that is dedicated to the task. In this way the accessing of the freelist is able to be carried out in parallel with standard file processing. This dedicated process is woken up via the CONTINUE statement. When it has finished its assigned task it goes back to sleep via the DELAY statement.

It can be seen that this strategy performs at its best on a steady state system. A steady state system being one where blocks are being procured and disposed of in roughly equal quantities. This stops the necessity of having to access the freelist and causes the overhead to fall to almost nothing.

```
TYPE FREEBLOCKS = MONITOR

  CONST MAXBLOCKS = 10;

  VAR   FREEBLOCKS: ARRAY[1..MAXBLOCKS] OF DISKPAGE;
        NOFREE    : INTEGER;
        MANAGER   : QUEUE;


PROCEDURE ENTRY PROCURE(VAR INDX   : DISKPAGE;
                        VAR RETCODE: INTEGER );
BEGIN
  IF NOFREE = 0
  THEN RETCODE := 15            (*  Disk must be full  *)
  ELSE BEGIN
    INDX := FREEBLOCKS[NOFREE];

    NOFREE := NOFREE - 1;
    IF NOFREE = 0
    THEN CONTINUE(MANAGER);
  END;
END (* Procure *);

PROCEDURE ENTRY DISPOSE(VAR INDX   : DISKPAGE);
BEGIN
  NOFREE := NOFREE + 1;
  FREEBLOCKS[NOFREE] := INDX;

     (* Call manager to remove blocks from array *)
  IF NOFREE = MAXBLOCKS
  THEN CONTINUE(MANAGER);
END (* Dispose *);

PROCEDURE ENTRY MANAGE
BEGIN
  CYCLE
    IF NOFREE < (MAXBLOCKS DIV 2) THEN BEGIN

       < Replenish array from freelist >
    END ELSE
    IF NOFREE > MAXBLOCKS DIV 2 THEN BEGIN

        < Remove blocks from array >
       < and chain back into freelist>
    END;

    DELAY(MANAGER);
  END;
END (* Manage *);

BEGIN
  NOFREE := 0;
END;
```

Figure 3.8

3.3.2.2 Bit Map Method.

With the bit map strategy the freelist will be maintained with a structure similar to this

```
FREELIST :  PACKED ARRAY[0 .. <no_pages>] OF BOOLEAN;
```

Each page is free if the corresponding element of the array is true. The array is specified as packed so that each element of the array will occupy only one bit. Since the freelist will be maintained in a disk file this means that with each disk page holding 8160 bits of data, the entire bit map for the device can be held on only a few actual disk pages.

Unfortunately Concurrent Pascal doesn't support PACKED types which means that each boolean value in the array will require one word instead of one bit. This was deemed unsuitable for this situation. The thing that appears to be missing from the language, is not packed types, but the ability to reference specific bits in memory. This is a necessity in several areas within an operating system. Modula recognised this problem and supports a data type BITS which is defined in Pascal as follows:

```
BITS = PACKED ARRAY [0 .. w] OF BOOLEAN;
                { where w = 15 for 16 bit machines }
```

The data type occupies a single word with each element referencing a specific bit.

Using a data structure like BITS the declaration for FREELIST now becomes

```
FREELIST :  ARRAY [ 0 .. <no_pages div 16> ] OF BITS
```

and to reference a specific page in the array the following expression will need to be used.

```
FREELIST [ PAGENO DIV 16, PAGENO MOD 16 ]
```

Although this expression is more complicated than the initial one using the PACKED structure it is much more universal and would not be implementation dependent. Packing of structures on the other hand, is generally left to the implementer of the compiler and may vary between implementations. This difference has been found by the author in two different Pascal compilers on a PRIME 750.

As mentioned previously Concurrent Pascal does not possess a structure like BITS and the only datatype that is at all applicable is that of the SET. The BITS datatype can be simulated by the following declaration.

```
BITS = SET OF 0..15;
```

Using a set of this type, there are two types of operation required. First to see if a page is free or not. This is achieved by the following test:

```
IF (PAGENO MOD 16) IN FREELIST [ PAGENO DIV 16 ] THEN ...
```

which is very similar in syntax to the Modula equivalent of

```
IF FREELIST [ PAGENO DIV 16, PAGENO MOD 16 ] THEN ...
```

The unsuitability of the SET type is born out though, when trying to add or delete a page from the FREELIST. To add a page using the standard bits structure of Modula the statement would be as

follows

```
FREELIST [ PAGENO DIV 16, PAGENO MOD 16 ] := TRUE;
```

whereas using sets the statement will be of the form

```
FREELIST[PAGENO DIV 16] := FREELIST[PAGENO DIV 16] +
                           [PAGENO MOD 16];
```

The concise nature of the first statement compared with that of the second clearly shows the superiority of the Modula BITS type over that of the set for this situation. See section 4.1.5 for further discussion on the BITS attribute.

3.3.3 Common Disk Buffers.

Common disk buffers were used in MUSCLE to cut down on the overhead of having to go to disk on every access. A physical disk page in MUSCLE contains several records or lines with only one being obtained on each call. In most cases the disk block need only be read in from disk prior to the first record being read. The successive lines are accessed from the already in memory disk block.

There are five such buffers in MUSCLE with each one contained in a monitor and accessed via the call

```
PAGES[ PAGENO ] . <operation> ( <parameter> )
```

where <operation> is one of

```
READ        /* Read a logical line
WRITE       /* Write a logical line
GETPAGE     /* Read a physical disk block
PUTPAGE     /* Write a physical disk block
```

When a process wishes to access a certain disk address, it will seek the best buffer to use. This is because a certain buffer may already contain the disk block that the process wants to access. To perform this buffer selection another monitor is used. This monitor maintains status information on all the buffers. The ideal situation would be for this monitor to itself call the prospective buffer. This unfortunately proves infeasible because of the 'Nested Monitor Problem' already mentioned in section 3.3.1. The problem is that with nested monitors only one of the buffers could be accessed at a time.

An alternative to this nesting of monitors is to have the process call each of the monitors in turn. The process must therefore execute something like the following two calls

```
BUFFER.ALLOCATE ( ADDRESS, PAGENO )
PAGES[PAGENO].<operation> ( <parameters> )
```

The first call uses the requested disk address specified in the first parameter to select the best buffer to use. The buffer number is returned in the second parameter. Although this sequence of statements seem correct there is a possible timing hole. A process executing these statements may be suspended after the call to the ALLOCATE procedure. A second process may then be able to beat it into the same PAGES monitor. In this circumstance the status information in the allocate monitor may be corrupt in that it won't possess the actual status information of the page buffers. The corruption will only be temporary though, with the system correcting itself after a certain amount of time. The only noticable effect of it is possible degradation of performance during the period of the corruption with the

actual disk being accessed more often than necessary. This bug is therefore termed a performance bug.

3.4 Performance of MUSCLE.

In this section some performance tests on different versions of MUSCLE are described. These tests have been carried out using user programs that are system bound in nature. The term system bound implies that the bulk of the execution time is spent in the operating system and not the user program. In using these types of programs the maximum difference in performance between the versions of operating system is obtained.

3.4.1 File System Structure.

In section 3.3.1 it was stated that monitors could not be used in either the Logical File System (LFS) or the Basic File System (BFS) modules of the file system. The was true because if either of these levels was a monitor then the overall system performance would be substantially retarded. This degradation is due to the limitation that only one process may enter the File System at a time.

In this section two versions of MUSCLE will be compared. The first one is the standard MUSCLE operating system which uses a class to support both the LFS and BFS levels. The second one will have some sections of the BFS in the form of a monitor thereby stopping more that one process from using those routines of the File System at the same time. It should be noted that the entire BFS couldn't be enclosed in the monitor. Doing so disabled all but one terminal from accepting commands at the same time. The actual sections of the BFS enclosed in the monitor are

only the read from disk, the write to disk and the write to screen routines. The following user program was executed under each of the systems.

```
PROGRAM TEST;

  VAR   LINE: PACKED ARRAY[1..80] OF CHAR;

BEGIN
  WHILE NOT EOF(INPUT) DO BEGIN
    READLN(LINE);

    WRITELN(LINE)
  END
END.
```

This program reads from a file on disk and writes each line of that file out to the screen. As this program is primarily I/O bound in one form or another (as well as being system bound) it should produce nearly the widest difference in performance between the two systems.

The results obtained from running this program on up to four terminals simultaneously are described in the table below.

number of terminals	Standard MUSCLE. (seconds)	MUSCLE (with monitor). (seconds)
1	75	75
2	78	150
3	81	225
4	84	291

It can be seen that having monitors at the top of the file system does stop any possibility of overlapping I/O on different devices. In this case running the program simultaneously at n different terminals required n times the elapsed time of running it on only one terminal. With the standard MUSCLE system the penalty for each additional terminal running the program was four percent of the time required to run it at one terminal. This

shows that significant gains can be made through overlapping I/O.

It should also be noted that the times for both systems were identical when the program was run on only one terminal. This shows that the overhead involved with actually entering and exiting monitors is negligible when a process doesn't have to wait.

3.4.2 Strings versus Packed Arrays of Char.

In MUSCLE, as in most operating systems, there are many places where strings have to be manipulated. These occur mainly in the top levels of the system where the user requests are handled and decoded, but are also scattered in varying areas further down in the system. In Concurrent Pascal, like standard Pascal, the only mechanism able to be used to handle such strings is that of the PACKED ARRAY of CHAR and this lack of a string type has been one area where Pascal has been criticised severely. In this section it will be shown that having to use ARRAYS to handle strings is both cumbersome and inefficient, and therefore inappropriate for real operating system purposes.

Strings, in the form of these PACKED ARRAY of CHARS, are handled in two places in MUSCLE. One area is in the process USERMACHINE (See Appendix D) where the pathname given by the user program is broken up into its component pieces. The second area is in the monitor DISKBUFFER where lines are packed and unpacked from the compressed format stored on disk.

Pathnames (e.g. A>B>C) in MUSCLE are of the general form:

{ filename > } filename

where { } denotes zero of more repetitions

where filename is a letter followed by any combination of letters or digits. To perform the task of breaking up this pathname, a procedure called GETTOKEN is used which returns the next token on the line in a variable called TOKEN. The main section of the code is therefore reduced to the following:

```
LEVEL := 0;
REPEAT
  GETTOKEN;

  IF TOKEN <> IDENTIFIER THEN BEGIN
     <Bad Pathname>
  END;

  LEVEL := LEVEL + 1;
  NAMES[LEVEL] := IDENT;

  GETTOKEN
UNTIL TOKEN <> GREATER_THAN;

IF TOKEN <> NULL THEN BEGIN
   <Bad Pathname>
END;
```

assuming the following declarations

```
TYPE NAME_TYPE = PACKED ARRAY[1..16] OF CHAR;

VAR  TOKEN : (NULL, IDENTIFIER, GREATER_THAN);
     LEVEL : INTEGER;
     IDENT : NAME_TYPE;
     NAMES : ARRAY[1..n] OF NAME_TYPE;
```

This section of code appears quite straight forward and concise but this is mainly due to all the string scanning being performed in the routine GETTOKEN which consists of an additional thirty lines of Pascal source.

Performing this same task in a language like PL/1 which does contain string types would produce something like the following code

```
LEVEL = 0;
DO WHILE (LENGTH(PATHNAME) > 0);
  IDENT = BEFORE(PATHNAME, '>');
  PATHNAME = AFTER(PATHNAME, '>');

  IF LENGTH(IDENT) = 0 !
     VERIFY(IDENT, ALPHANUMERICS)
  THEN DO;
    <Bad Pathname>
  END;

  LEVEL = LEVEL + 1;
  NAMES(LEVEL) = IDENT;
END;

IF LEVEL = 0 THEN DO;
   <Bad Pathname>
END;
```

This section of PL/1 code is very similar in both length and form to the Pascal code but is complete in that it isn't dependent upon other code in the program whereas the Pascal code is dependent upon a routine GETTOKEN. This example shows that strings in a language allow such tasks as this to be programmed in a more concise manner and generally simplify the task of programming them.

The second area where strings are used in the MUSCLE source concerns the expansion of compressed lines. Compressed lines stored on the disk must be expanded into a temporary array. This is performed by the following section of Concurrent Pascal code taken from MUSCLE.

```
LEN := ORD(BUFFER.TEXT[INDX]);
BLANKS := ORD(BUFFER.TEXT[INDX+1]);
INDX := INDX + 2;

FOR CNT := 0 TO BLANKS-1 DO
  TEMP[CNT] := ' ';

FOR CNT := BLANKS TO BLANKS+LEN-1 DO BEGIN
  TEMP[CNT] := BUFFER.TEXT[INDX];
  INDX := INDX + 1;
END;

FOR CNT := BLANKS+LEN TO 255 DO
  TEMP[CNT] := ' ';
```

This code was very expensive in terms of process time and was replaced by some assembler code. The assembler code is invoked through an IO call to a pseudo-device called EXPAND_LINE. The equivalent source in Concurrent Pascal was therefore reduced to the following.

```
PARAM.BUFFER_OFFSET := 14;
PARAM.BUFFER_LENGTH := INDX - 1;

IO(TEMP, PARAM, EXPAND_LINE);
INDX := INDX + ORD(BUFFER.TEXT[INDX]) + 2;
```

Here the first two statements specify both the location of BUFFER and the start position that the current line occupies within the BUFFER. The actual IO call then moves this line into the temporary array TEMP.

To compare these two methods the following user program was run under MUSCLE.

```
PROGRAM SPEEDTEST;

BEGIN
  WHILE NOT EOF(INPUT) DO
    READLN(INPUT);
END.
```

This program reads a 3000 line disk file from the internal file

INPUT. It was run under three versions of MUSCLE, the first two contain the Pascal code while the third uses the assembler code via the device EXPAND_LINE. The difference between the first two is that one was compiled with array bounds checking turned on in the operating system while the other one was compiled without it. Having both these versions should determine whether the bulk of the overhead of using arrays of char is in the mechanism itself or if in fact having the additional checking code plays a large factor in the overhead.

The following results were obtained:

number of terminals	Pascal(check on) (seconds)	Pascal(no check) (seconds)	I/O routine (seconds)
1	39	34	15
2	75	65	27
3	113	97	41
4	151	130	55

These show that having the operating system compiled with checking turned on places an additional ten percent overhead on the system but replacing the Pascal type string handling with the IO routine allows savings of up to sixty-five percent in the elapsed time to be made. This shows that although the bounds checking code does impose some overhead, it is far outweighed by the mechanism itself.

Were strings provided in the language then it could be expected that these would provide a much more efficient mechanism than the packed arrays of char. They should prove better than the arrays as they allow the use of string handling machine instructions which are much faster than the byte by byte operations that Pascal uses. The use of strings though, wouldn't be as good as

the IO routine. This is because this routine was tailor made for the purpose and takes advantage of shortcuts. Generalized string routines could not be as efficient as special purpose ones such as this.

3.5 Beyond MUSCLE.

MUSCLE has no pretentions of being a fully fledged operating system. It is suggested though, that with more appropriate hardware, the fundamental structures of MUSCLE could be expanded into something approaching that of a large operating system. Of the three areas already discussed in this chapter, both the user interface and the file system structure employed in MUSCLE appear adequate for a larger system. The memory management system is the only section that is deemed unsuitable for a larger system.

In this section a discussion on how Concurrent Pascal could be used to implement a memory management system for a large system will be made. This system which supports virtual memory will be based on a demand paging system as this is commonly used on large systems today.

3.5.1 Supporting a Virtual Memory System.

The main difference between the paged system and the one employed by MUSCLE is that with the paged system, memory is allocated on demand whereas MUSCLE allocates it prior to use. With the paged system a page fault will occur when a process tries to access memory that has not yet been allocated. The system will then grab a block or page of memory and give it to the user. In MUSCLE the memory is allocated to the user prior to his requiring it. This difference is quite substantial in its requirements on

the operating system. In MUSCLE the request for memory is via an ordinary supervisor call and the linking mechanism in this case is quite straightforward. With the paging system though, the request is initiated by a machine interrupt which will be trapped by the Kernel. The problem in this case is how the Kernel calls the appropriate system routine.

In MUSCLE the user must ask for memory prior to using it and the release it after use. In a paging system the user has only to ask for it. In meeting one user's request for memory the system may have to take it from another user. Should this first user then try to access it, he will generate a page fault and the system must find another block to replace the lost one. The memory management system for the paged system will therefore only consist of one procedure which allocates a block of memory. The declaration for this procedure will probably be of the form

```
PROCEDURE ENTRY ALLOCATE_PAGE(USER    : USER_TYPE;
                              ADDRESS: ADDRESS_TYPE);
```

The parameter USER specifies the process getting the page fault and ADDRESS determines the actual virtual address causing the fault.

There appear to be two ways the system could be written in Concurrent Pascal to support the paged system. The first is to have the user handle his own faults in much the same way as the user in MUSCLE allocates and releases the memory he requires. The other method is to have a dedicated process in the operating system that detects and handles the fault. The following two subsections will give a description of each method in turn.

3.5.1.1 User handling his own Page Faults.

In this mechanism the process dedicated to servicing the user will handle the interrupt. In the example of MUSCLE this process would be an invocation of the process type USERMACHINE (See Appendix D). The main problem with this method is how the Kernel passes control onto the Concurrent Pascal process after trapping the page fault. Ideally it would be like a ordinary supervisor call and with this in mind the following mechanism is suggested.

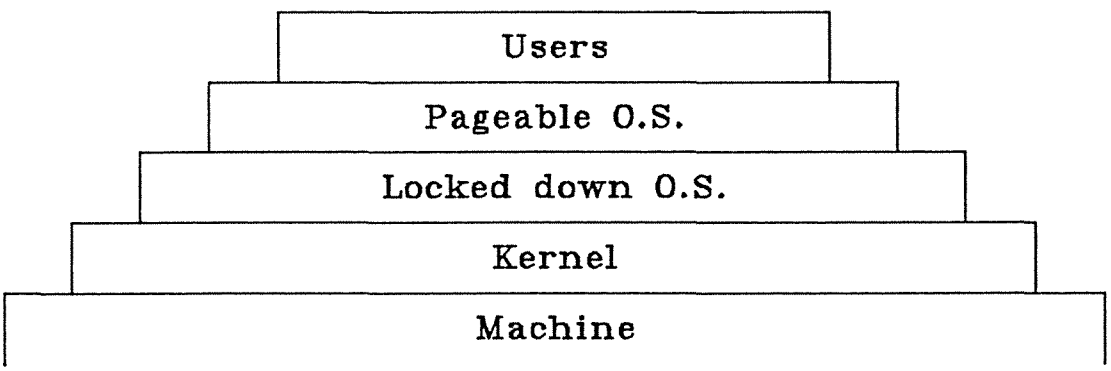
```
PROGRAM JOB (...)
ENTRY FAULT_HANDLER, ...;
```

Here the first procedure in the program ENTRY construct is known by the Kernel to be the fault handling routine for that user. The only time that procedure will be called is therefore as a result of a page fault. Once the FAULT_HANDLER routine has control the rest of the logic should be quite straightforward and easily handled by the system language.

It can be seen that this mechanism will only support page faults that occur while executing a user program. This was intentional to stop the possibility of a recursive page fault. A recursive page fault is one where the actual fault handler which is handling a current page fault causes another page fault. To stop this from happening in this example the entire operating system will have to be locked down and only the user memory paged. The term locked down means that it is always resident in memory. Should it be infeasible to have the entire operating system in memory then there must be a mechanism by which the memory management system will know which part of the operating system is

locked down and which is pageable.

To overcome this problem a two level system is suggested, one level (the lowest) contains the locked down memory and the other has all the pageable memory. The lowest level will contain all the memory management routines and everything they call, together with other routines the implementer specifically wants to be permanently resident in memory. To support such a configuration it would be best to have two Concurrent Pascal programs. Each program would support one level of the operating system. On system startup the Kernel would load the lower level program into memory. This lower level program would then call the second Concurrent Pascal program which in turn would call the user programs. This would create the hierarchy that is depicted in the diagram below.



Unfortunately, Concurrent Pascal has been severely criticised for its inability to load and execute other Concurrent Pascal programs. This leads the author to the opinion that paging the operating system by using this mechanism is infeasible with the language Concurrent Pascal as it exists at the moment.

3.5.1.2 Dedicated Processes handling Page Faults.

In this case a process is dedicated to trapping and then handling the page faults. Just as in the previous method, the problem is how the Kernel passes control to this process after trapping the fault. The obvious solution to this is once again the use of the IO primitive. Here the process will make an IO call to a pseudo-device and be suspended until a fault is detected. The process is then allowed to return with the necessary status information contained in the IO parameter.

A paging system using this mechanism has in fact already been produced in Concurrent Pascal [Graef et al. 79]. In this implementation the swapper process managing the virtual memory uses two of these pseudo-devices. One of these devices accesses the swap queue and the other, the page tables. Both devices are accessed by the IO call. Although this example shows that Concurrent Pascal can support the bulk of the logic needed for managing virtual memory the implementers must still have had to do some very messy operations in order to get things working. One point that the article didn't include was how the swapper knew which pages it could swap out and which pages it couldn't. When a program pages itself the problem of knowing which part of itself it may page and which part must not be paged appears to be a very perplexing problem.

Chapter 4

Analysis of Concurrent Pascal

4 Critical Analysis of Concurrent Pascal.

In this section the features of Concurrent Pascal will be compared with those facilities required by the implementation of multi-user operating systems. In the implementation of MUSCLE, several deficiencies in the language were found. This discussion will review these and suggest alternative facilities. The feasibility of these alternate facilities will also be considered. The aim of this discussion will be to suggest building blocks for a new system language. This language, if implemented, would hopefully overcome many of the problems found with Concurrent Pascal and provide a more useful language with which to implement such systems.

4.1 Basic Data Types.

The language Pascal offers a number of basic data types. These simple data types are supported by the language and are used by the programmer as building blocks to produce more sophisticated data structures. Concurrent Pascal supports all these basic data types except for the pointer type. In addition to these it supports others which are primarily concerned with the concurrent properties of the language. This section discusses some of these basic data types that Concurrent Pascal supports. Also in this section some additional data types not supported by the language will be described. Discussion of these non-supported data types will show limitations in the support offered by the Concurrent Pascal language.

4.1.1 The Type REAL.

The use of real arithmetic in operating systems is negligible if not non-existent. No real arithmetic was used in the implementation of MUSCLE and it was only used once or twice in the entire SOLO system. The supporting of it in Concurrent Pascal is therefore quite unnecessary. Prael [Prael 82] concurs with this opinion of the real and stated that the real data type is "excess baggage which merely complicates the language implementation and maintenance without sufficient return".

On the other hand, the system programmer often has to work to double precision in integer arithmetic. It is suggested therefore that languages like Concurrent Pascal should support several integer types. These different types would match those supported by the machine. In Ada [Pyle 81] for example, there are three standard integer types, SHORT_INTEGER, INTEGER and LONG_INTEGER. Although these types are implementation dependent and may pose portability problems they offer a lot of flexibility to the programmer.

4.1.2 The String Type.

It was suggested in section 3.4.2 that Concurrent Pascal, like Pascal, requires strings. It was shown in that section that the use of packed arrays of char was both inconvenient and inefficient. Other people [Sale 79] [Boswell et al. 83] have recognised that Pascal requires a string type and have suggested extensions to the language to incorporate such a type. Prael [Prael 82] was concerned that the lack of a string type when implementing operating systems was more than an inconvenience and stated that "inconveniences of this type usually translate into

frequently recurring bugs and incompatible system components". Packed arrays of char may sometimes be useful, however. It is therefore suggested, that any string extension in a Pascal based language should be accessible as both a string and a packed array of char.

4.1.3 The Pointer Type.

The implementation of MUSCLE has shown that Concurrent Pascal can be used to support a system with a hierarchical or layered organisation. This implementation has shown up one limitation of the language in this respect. When each layer of the system is contained in a class or monitor, the logic of a given level can only access data belonging to that level. In a true hierarchic system, a given level should have full access to all code and data not only on that current level but all levels beneath it. In Concurrent Pascal only the code of lower levels is accessible. Data of lower levels is not accessible.

This inability to access data of lower layers forces the programmer to take one of two routes in the implementation. These are either to take copies of the relevant data or else move the logic accessing that data down to the level where the data is declared. Both these alternatives are undesirable. The first method is costly because it imposes the overhead of copying data backwards and forwards between the levels. The second method is messy because it forces the programmer to move away from the layered scheme to that of a flat or one level system.

There is a definite need for a given level to access data from lower levels. The remaining question is: how this may be

achieved? Two existing facilities of Pascal could be used to make this possible. The first facility consists of allowing global data to be declared in the language. This would mean that all the levels could access the data directly. This mechanism has the disadvantage that data is not protected from unauthorized access, as was shown in section 1.1. It is suggested therefore, that the use of global data is not suitable for such applications.

The second facility that could be used is that of the pointer type. The lack of a pointer type has already been noted by Ravn [Ravn 82] and so it is generally accepted that Concurrent Pascal is indeed deficient in this respect. Two types of pointer come to mind that could be used. These two types are the capability type pointer [McKendry and Campbell 80] and the Pascal type pointer. The capability type pointers contain builtin restrictions which stop concurrent access to the same data structure by several processes. They also prohibit processes from accessing data structures which they don't currently have rights to. The Pascal type pointer on the other hand doesn't have these builtin restrictions. This type of pointer is designed to be used by only one process.

There is a definite need for the capability type pointers in the language. The ordinary Pascal pointer type would also be useful in the language for intra-process circumstances and so it is suggested that the language should support both pointer constructs. The Pascal pointer type would have the restriction that it is not able to be passed to, or used in, monitors. This restriction guarantees that the Pascal type pointer is not passed

between Processes.

4.1.4 The type QUEUE.

The standard type QUEUE in Concurrent Pascal is used to synchronise processes in monitors. The synchronisation is achieved by suspending the processes on queues until a certain condition is met. When the condition becomes true they are then allowed to continue. Unfortunately a limitation of the language is that only one process at a time can be waiting on a queue. In many instances though, several processes may need to be suspended. This necessitates the use of arrays of queues. Having to use arrays of this type requires extra logic which unnecessarily increases the length of the program. To show the extra logic required a section of SOLO [Brinch Hansen 76] shown in figure 4.1 will be used.

```
CONST PROCESSCOUNT = 7;
TYPE PROCESSQUEUE = ARRAY [1..PROCESSCOUNT] OF QUEUE;


TYPE RESOURCE =
MONITOR

VAR FREE: BOOLEAN; Q: PROCESSQUEUE; NEXT: FIFO;

PROCEDURE ENTRY REQUEST;
BEGIN
IF FREE THEN FREE:= FALSE
  ELSE DELAY(Q[NEXT.ARRIVAL]);
END;

PROCEDURE ENTRY RELEASE;
BEGIN
IF NEXT.EMPTY THEN FREE:= TRUE
  ELSE CONTINUE(Q[NEXT.DEPARTURE]);
END;

BEGIN FREE:= TRUE; INIT NEXT(PROCESSCOUNT) END;
```

Figure 4.1

In this example a class of type FIFO is used to select the

element of the array on which the process will be delayed. This means that every time a process is delayed or continued a function call has to be made as well as the call to the kernel which performs the actual process exchange.

The restriction of one process queues has been dropped in many languages that followed Concurrent Pascal. For example, both the SIGNAL type of Modula [Wirth 77] and the CONDITION type of Pascal Plus [Welsh and McKeag 80] are modelled on the QUEUE type of Concurrent Pascal. Both allow several processes to be simultaneously suspended on them. The equivalent monitor in Pascal Plus of the one in figure 4.1 would therefore be of the form in figure 4.2. Although this version doesn't appear all that much simpler than the Concurrent Pascal one, it does away with the need for the class FIFO. In doing so it should have less runtime overhead than the Concurrent Pascal one. The greater flexibility and power derived from the CONDITION and SIGNAL type variables thereby suggest they are far more convenient and useful to the system programmer.

4.1.5 Accessing at Bit Level.

In section 3.3.2 it was noted that Concurrent Pascal was missing the ability to address down to the bit level. This facility is commonly required at the system level, especially when interfacing with devices. It is also useful as a means of maintaining data in a compact form. The PACKED attribute of pascal is deemed inappropriate for this type of programming as the actual packing of such structures is left entirely to the compiler. At the system level the programmer must have total control on how the data is stored.

```
MONITOR MODULE RESOURCE;

VAR FREE: BOOLEAN; AVAILABLE: CONDITION;

PROCEDURE *REQUEST;
BEGIN
  IF FREE THEN FREE := FALSE
    ELSE AVAILABLE.WAIT
END;

PROCEDURE *RELEASE;
BEGIN
  IF AVAILABLE.LENGTH = 0 THEN FREE := TRUE
    ELSE AVAILABLE.SIGNAL
END;

BEGIN FREE := TRUE END;
```

Figure 4.2

Modula [Wirth 77] recognised the problem and in doing so supports the BITS type as already mentioned in section 3.3.2.2. Unfortunately the BITS type is itself deficient in that it doesn't permit the programmer to specify objects, whose components are larger than one bit. The language PL/1 [Bates and Douglas 75] provides more powerful attributes in the form of the FIXED BIN and BIT types. These allow the programmer to specify the size of the variable in terms of a certain number of bits but is probably inefficient when accessing these structures. It is suggested therefore, that the following compromise between these two approaches be made. This compromise should produce both an efficient and a flexible data structuring tool.

```
TYPE    STRUCTURE = WORD RECORD
                      ....
                    END;
```

The prefix WORD is used to specify that the size of the associated record is one machine word. The compiler will check to ensure that this is in fact true. Within this structure would

be variables of the type BIT(n). This type being equivalent to the integer subrange 0 .. 2**n-1 which occupies n bits.

```
e.g.        STRUCT  =  WORD RECORD
                         A : BIT(3);
                         B : BIT(5);
                         C : BIT(8)
                       END;
```

The type BIT(n) should only be allowed to be used in a WORD record which ensures that the programmer will not misinterpret the storage representation of a data structure. This restriction may be relaxed in some cases though, to allow declarations like the following:

```
ARRAY[0..m*(16 div n)-1] of BIT(n) {for m > 0 and n in [1,2,4,8]}
```

In this case, an integral number of the basic units will fit into a word and also the entire length of the array is also an integral number of words. The Modula type BITS could therefore be simulated by the following declaration:

```
TYPE   BITS = ARRAY[0..15] OF BIT(1);
```

4.2 Procedure Parameters.

Concurrent Pascal supports two types of parameters, the variable parameter and the constant parameter. The variable parameter is identical to the variable parameter of Pascal and is passed by reference [Aho and Ullman 77]. The constant parameter on the other hand differs substantially from the value parameter of Pascal. Concurrent Pascal's constant parameters cannot be altered at all in the procedure whereas Pascal's value parameters may be changed. Changing the value parameter though, doesn't alter the corresponding actual parameter. This restriction in

the constant parameter was encountered time and time again in the implementation of MUSCLE and resulted in local copies of the parameter having to be made. These copies were then used and able to be subsequently changed. The problems caused by the constant parameter of Concurrent Pascal shows that it is therefore not a suitable replacement to Pascal's value parameter.

It is suggested therefore, that there should be three types of parameters permitted in the language. The first type is the 'var' type of both Pascal and Concurrent Pascal. This will allow the procedure to modify the actual parameter. The second type required is the value parameter of Pascal. In this case a copy of the actual parameter is made. The procedure may then access and possibly change only a local copy of the original parameter. Copying parameters can be very expensive especially when large arrays or records are concerned and for this reason a third type is suggested. This third type, prefixed by 'const', will be the constant type of Concurrent Pascal. This parameter will have the restriction that it cannot be modified in the procedure. As access to these parameters is read only there will be no reason to actually take a copy of the parameter. This protects the actual parameter from modification and doesn't incur the overhead of having to copy it.

4.3 The Monitor.

From its initial exposure in Concurrent Pascal the monitor has subsequently appeared in several other languages including Modula, Pascal Plus and CSP/k. The wide acceptance of the construct shows that it is successful in its design. Monitors successfully control access to shared data. The main difficulty

with monitors occur when they are nested. In this case the language syntax allows the nesting but the semantic support is missing. This lack of semantic support shows that the monitor is still very much in its infancy.

The problem of nesting monitors was first reported by Lister [Lister 77]. He was concerned that if a process called a second monitor while in a monitor, deadlock may occur should that process execute a delay procedure in the second monitor. The problem in this case is that on execution of the delay instruction, exclusion is released on the second nested monitor but not on the first one. As the first monitor is still held, no other process can enter the nested monitor via the first one to signal the waiting process. In fact any such process trying to do so, would be delayed indefinitely at the gate of the first one thereby creating the deadlock.

The problem of deadlock, although probably the most disastrous, is only one of the problems caused through the nesting of monitors. The problem which most often occurs is that of unnecessarily delaying processes. This delay to processes is caused when they have to wait at a monitor which is being unduly held by another process. This was noted by Haddon [Haddon 77], who suggested alternative approaches to handle the nesting of monitors which overcome this problem. One of these suggestions is to release exclusion on the current monitor prior to making a call to a second monitor. No processes would then be held up on the first monitor. Haddon was of the opinion that this approach should replace the current one and be used at all times.

After this initial criticism of the monitor by Lister, David Parnas [Parnas 78] came to its defense. He stated that Lister (and indirectly Haddon) couldn't generalize the problem of nested monitors because the action required was dependent upon the actual situation being programmed. Although this stand by Parnas has itself been criticised [Lister 78], I am in full agreement with him. Haddon wanted only one of the two approaches to be followed. It is far more desirable for the compiler to support both approaches and allow the programmer to choose between them depending on the situation.

For example, in section 3.3.3 the supporting of common disk buffers in MUSCLE is described. This is one example where the alternative mechanism as suggested by Haddon could prove effective. In this example, a monitor is used to select which one of the disk buffers is to be used by the process. The buffers are represented by an array of monitors thereby only allowing one process to access them at a time. Ideally the calling sequence would consist of a hierarchical system where the first monitor was used to select and then call the wanted buffer. This is illustrated in figure 4.3. The selection monitor has an entry routine for every entry routine in the DISKBUFFER monitor. The status information of this monitor is only accessed in the procedure SELECT. This means that when this function has exited there is theoretically no reason to maintain mutual exclusion on this monitor.

In Concurrent Pascal the mutual exclusion is not released until the entry routine itself is exited, which does not occur until the DISKBUFFER monitor has been executed. Should a second

process make a call to the BUFFER_SELECTOR monitor, it will be suspended until the first process exits. This occurs even though this second process may wish to access a totally different DISKBUFFER monitor. Using the alternative mechanism allows all the diskbuffers to be accessed at the same time whereas only one could be accessed at a time using the first form. Should a second process want to access a buffer already being used then it will be held up on the call to the second monitor and not the first one. The use of this second form allows a more hierarchical structure to be followed. It doesn't suffer from the inherent inefficiencies imposed by the first and existing form.

```
TYPE  BUFFER_RANGE =  1 .. NOBUFFERS;
      DISKBUFFERS  =  ARRAY[BUFFER_RANGE] OF DISKBUFFER;

TYPE  BUFFER_SELECTOR = MONITOR(BUFFERS: DISKBUFFERS);


  FUNCTION SELECT(..): BUFFER_RANGE;
  BEGIN
    < Logic required to select the wanted buffer >
  END;

PROCEDURE ENTRY <access> (<parameters>);
BEGIN
  BUFFERS[ SELECT(..) ].<access> ( <parameters> );
END;

BEGIN
END;
```

Figure 4.3

It has been shown there that the second alternative method is necessary for hierarchical structures. However, to support this method and not destroy the integrity of the monitor concept there appear to be three restrictions that must be met. These three restrictions are as follows:

1. If global variables of the first monitor are passed as parameters on the call to the second monitor, they must be passed by value. This means that a copy is taken of them prior to releasing the first monitor. This restriction will stop the possibility of monitor data being accessed without exclusion being held on that monitor.

2. On leaving the nested monitor the process will automatically return to the point where the first monitor was called and not back into the first monitor. Although this restriction may not always be necessary, its inclusion will prevent accidental programming errors occurring in the first monitor. These errors might occur because the process had temporarily released exclusion on the monitor.

3. Exclusion on the second monitor is requested prior to exclusion on the first being relinquished. This restriction stops a race condition occurring where a second process may beat the first process into the second monitor even though it acquired the first monitor after the first process. This condition was mentioned in section 3.3.3 and may cause performance degradation under certain circumstances.

These restrictions will probably not be suitable in all other situations and for this reason it is suggested that the programmer be allowed to choose the method currently utilized by Concurrent Pascal.

4.4 Standard Routines.

Concurrent Pascal contains several standard, or builtin routines. One purpose of these procedures (and functions) is to allow the programmer to interface with the Kernel level. The most notable procedure in this category is the IO procedure. Another use of these standard (or predeclared) routines is to support commonly needed logic. This section will discuss the addition of standard routines used specifically for this second reason. These routines aid the programmer in his implementation by providing primitive routines which he can then call.

Examples of this type of routine are the ABS and ODD functions of Pascal. If these functions were not supported in the language then something like the following statements would probably have to be used instead in the program.

```
if r < 0
then absr := -r          (* Instead of ABS(r) *)
else absr :=  r;

if (i mod 2) = 1
then oddi := true        (* Instead of ODD(i) *)
else oddi := false;
```

There are two reasons why this substitute code is not an effective replacement for the standard function itself. Firstly, it is more verbose than the function call. For this reason its underlying logic is harder to comprehend. Secondly, it is probably far less efficient than the function call. With the builtin function the compiler writer can generate the best code for the given situation. With the inline substitute code this generally will not be the case. For example, the ODD function returns the value true if the integer parameter is odd. With many Pascal implementations this truth value is obtained by

merely returning the value of the least significant (right most) bit of the parameter. This value can be obtained through an AND instruction which zeros all other bits in the word. The substitute code on the other hand, requires a division to be performed. The division instruction, which supports the MOD operator, is generally a lot slower on most machines than logical operations like the AND instruction.

This example shows that these predeclared routines are very important to the implementation as they allow both compact and quite efficient code to be generated. One limitation of predeclared routines in languages like Pascal is that there is no easy method to provide new ones that may be required in just one or two specific projects. It is suggested therefore, that the language should provide a mechanism whereby the programmer may install some predeclared routines of his own. This will then allow the programmer to make full use of the machine's architecture or instruction set, which is generally not fully utilized by the ordinary constructs in the language.

Ada [Pyle 81] is one language that does allow the programmer to define his own library routines. To make use of such routines the programmer must include the following statement at the head of his program.

WITH <lib 1>, <lib 2>;

This statement allows the program to access all the routines contained in the libraries <lib 1> and <lib 2>. The inclusion of this facility in Ada will definitely benefit the programmers using the language. Unfortunately this mechanism doesn't support

the generation of inline code for such routines. In some situations this may be desired and so it is suggested that an improved language should support a macro type library as well as the Ada type. A macro type library would be one where the routines were supported by inline code. Although the macro type library would not be transportable like the Ada type, in system applications this would be a small price to pay for the greater flexibility obtained by having both types available.

4.5 Hierarchical Operating Systems.

It was suggested in section 3.5.2 that a two level operating system would be an appropriate mechanism to handle virtual memory. Each level would be contained in a separate program with the bottom level containing all the non-pageable code and the top one containing all the pageable operating system routines. This mechanism could easily be extended to a multi-level structure. A multi-levelled structure organised in this manner is what is probably required to implement the large operating systems of today and of the future.

In MUSCLE each level of the system was represented by a class or monitor. In a large system this organisation wouldn't be feasible because of its sheer size. A large system will contain many tens of thousands of lines and these couldn't conveniently be placed in one program. It is therefore suggested that the operating system language must allow programs written in it to be divided into separately compiled modules. As well as allowing the program to be broken into these several individual pieces, the language must also allow one system written in it to be run under another system also written in it. Were this facility

available then the system being implemented could easily be structured in a hierarchical fashion.

In Concurrent Pascal neither of these facilities are supported. The first facility is not supported as there is no mechanism in the language to allow externally compiled routines to be included. This stops the programmer from breaking a Concurrent Pascal program up into separate pieces. The second feature is not supported for the following two reasons:

1. There is no method in the language to declare routines that are supported by lower system levels. This is because the language expects only the bare Kernel to be beneath it. In a hierarchical system this is not the case as the upper levels will have an enlarged Kernel. In this case the Kernel consists of the original Kernel plus all the system levels in between. Although MUSCLE was composed of only one program supported by the plain Kernel, this problem still arose. Concurrent Pascal supports several standard routines which were included for the sole reason that SOLO required them. In this same way MUSCLE also required such routines, but unfortunately the requirements of MUSCLE differed from those of SOLO. This meant that the routines required by MUSCLE were not supported by the language and had to be incorporated in the system by using some devious methods. These methods usually involving a call to a pseudo-device via the IO primitive.

2. The PROGRAM statement (See section 3.1.1) mechanism is only designed to handle a single user process. In MUSCLE there

was one instance of the USER_MACHINE process for every terminal on the system. This organisation means that the processes in the system are arranged as depicted in figure 4.4a.

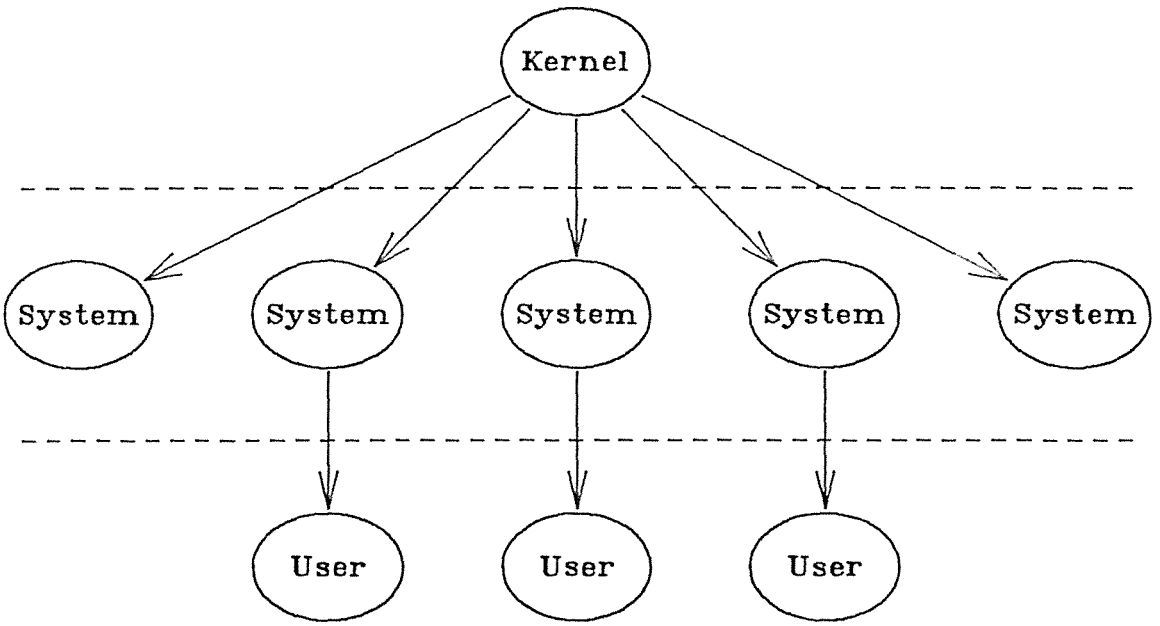


Figure 4.4a

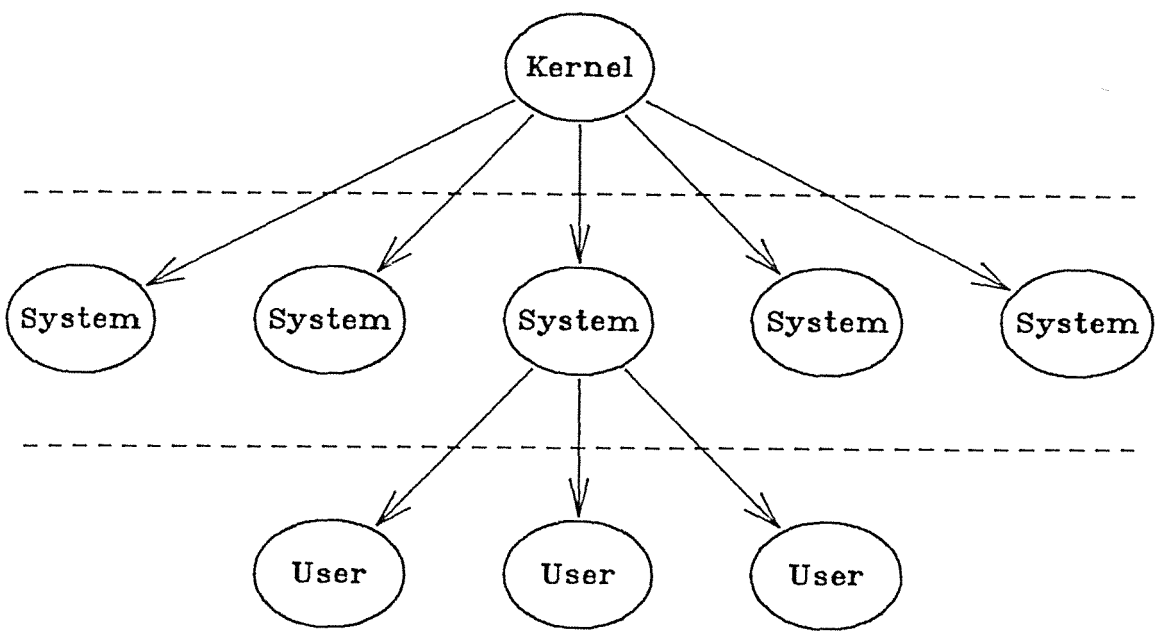


Figure 4.4b

To handle the running of Concurrent Pascal programs under a system written in the same language an organisation such as shown in figure 4.4b would have to be supported. In this case the one system process now has to be able to handle several user processes. All these user programs could theoretically execute supervisor calls on the next level down concurrently. Were this to happen then there would be several tasks running out of the same process module. This feature is definitely not allowed in Concurrent Pascal where only one process is allowed in each module at a time.

Modula-2 [Wirth 77], one of Concurrent Pascal's successors, is one language that does support external routines. To support these routines it has two declaration types, EXPORT and IMPORT. When identifiers appear in an EXPORT declaration, they will be known outside the module. This mechanism is very similar to the ENTRY type used in Concurrent Pascal. When a module wants to make use of another module's logic it includes an IMPORT declaration. For example, the declaration

```
FROM simplefiles IMPORT
   FILE, open, close, read, write;
```

allows the identifiers FILE, open, close, read and write to be accessed from the module simplefiles. The types of all these variables will be specified in an external definition module which the compiler has access to. This allows the compiler to perform full type checking on the imported variables.

This appears to be very tidy and safe mechanism to use for including external code. It does seem to suffer from one

pitfall. McCormack and Gleaves [McCormack and Gleaves 83] suggest that these external modules should be linked at run-time rather than at compile-time. Although run-time linking is a very powerful mechanism it does, in some circumstances, suffer poor performance when compared with prelinked code files. It is therefore suggested that the systems language should have two types of declarations for external routines. One would be prefixed by the word IMPORT which would allow library type routines to be included. Routines of this type would be prelinked to the module and therefore would not require any run-time linking support. The other type would be prefixed by the word ENVIRONMENT and routines of this type would be linked at run-time. This second type should therefore allow hierarchies of system programs to be assembled. The environment of each level will consist of the export variables of all the system levels, including the Kernel, beneath it.

To overcome the problem of having to handle simultaneous system calls from several user programs, two methods could be used. The first is to have the interface between the user and the system based on a message passing system. Using this mechanism will ensure that the second and successive calls will only be seen when the system is ready for them. Using a message passing mechanism at this level would probably require the entire system to be written as a message passing system. If this were not done then the user requests to each process could only be handled sequentially. This would stop any possibility of overlapping IO.

The alternative to a message passing system would be one where the the supervisor call would initiate a new system process. In

MUSCLE there was only one process that was dedicated to handling the requests of the initiated user program. In this alternative method there would be a pool of such processes waiting to handle the user requests. Upon receipt of a supervisor call, the Kernel would initiate one of these processes. If there are no processes waiting then the user task will be suspended until such time as there is one available. A mechanism such as this could probably be added to a language like Concurrent Pascal although the syntax would probably differ significantly from that presently used.

Chapter 5

Summary and Conclusions

Initially the topic of investigation for this research project was the "Implementation of an operating system in a concurrent programming language". In the ensuing developmental stages this emphasis has changed from the implementation written in Concurrent Pascal to a study of the language itself. This shift in emphasis is the result of complications and difficulties found in both the transporting of and then the actual use of Concurrent Pascal. These problems are documented in this thesis. A better understanding of the problems encountered will help in the production of true transportable systems.

It is generally recognised that interpreted systems are more highly transportable than the equivalent compiled version. For this reason Brinch Hansen decided to implement Concurrent Pascal as an interpreted system. A problem with interpreted systems though is that they are slower than the equivalent compiled system. With this in mind Brinch Hansen designed the interpreted system to be as efficient as possible. The format of the intermediate code is one area where efficiency was of optimum importance and which led to the final form of the code. Unfortunately although this choice of the intermediate code led to a relatively fast version on the PDP11, it also led to a less transportable system. This study suggests that where efficiency and transportability are both desired there must be two forms of virtual code. The first will be like the original virtual code which allows fast interpreting but is relatively machine

dependent. The other will be a form such as that described in Appendix B that is primarily designed for transportability. Also contained in the system would be a program that could take the transportable form of the code and translate it to the corresponding run form suitable for the machine it was converted on. This conversion program should be written in a standard language like Pascal which is readily available on many machines.

Brinch Hansen made at least one other mistake when producing the transportable version of SOLO. The method of implementation (i.e. either to produce a compiled or an interpreted version) must be left up to the person transporting the system. With Concurrent Pascal there was no easy path to follow to produce the compiled version thus rendering this method of implementation very difficult. The source of the compiler should theoretically have been written in a standard highly transportable language like Pascal and not a special dialect like sequential Pascal. In its current form the implementer has to either change the compiler to the standard language (as was done in this project) or else bootstrap the compiler by writing an interpreter that recognises the virtual code files. Bootstrapping the compiler in this example is further complicated as both the Sequential and Concurrent Pascal compilers would have to be supported. All this work has to be done before any conversion of the compiler to produce a compiled version of the system can be attempted. It is therefore strongly recommended that any transportable system must therefore be written in either a standard widely used language or at least a language which is easily supported on the target machine.

The implementation of MUSCLE has shown that Concurrent Pascal is lacking several important facilities. These deficiencies have been discussed in detail in chapters three and four of this thesis and include the following:

1. No string handling facilities.
2. No pointer type.
3. Difficulty at addressing bit data.
4. Cannot include external routines.
5. Cannot support Hierarchical Operating Systems.

This lack of facilities is easily explained by the fact that Brinch Hansen wanted to obtain a running implementation of the language. Had he designed a larger and more powerful language then its actual implementation would have been that more difficult to produce. The criticisms made of Concurrent Pascal in both this report and other publications though, have been beneficial in the concurrent programming field as a whole. These criticisms have led to other more powerful languages being created which don't suffer the problems of Concurrent Pascal. These later languages have gained much publicity so that concurrent programming languages are now firmly entrenched in the software scene. Their further development and wider use can be expected to continue in the future and this must surely produce much more maintainable and reliable system software.

Comparison between Size of Virtual and Actual Machine code.

In this appendix the interpreted code produced by the SOLO resident compilers is compared with that which could theoretically be produced by a compiler generating specific Series/1 machine code. This comparison is being performed on the procedure WRITETEXT which is contained in the program 'DO'. DO itself exists in the SOLO operating system. The procedure is as follows.

```
PROCEDURE WRITETEXT(TEXT: LINE);
  CONST NUL = '(:0:)';
  VAR I: INTEGER; C: CHAR;
BEGIN
  I := 1; C := TEXT[1];
  WHILE C <> NUL DO
  BEGIN
    DISPLAY(C);
    I := I + 1; C := TEXT[I];
  END;
END;
```

This procedure produces the following interpretative virtual code. The code has been divided into sections, each one of which represents a statement of the procedure.

```
A.    ENTER

B.    LOCALADD
      PUSHCON                    I := 1;
      COPYWORD

C.    LOCALADD
      PUSHLOC
      PUSHCON
      INDEX                      C := TEXT[1];
      PUSHBYTE
      COPYWORD
```

```
D.      PUSHLOC
        PUSHCON                          WHILE C <> NUL DO
        NEWORD
        FALSEJMP

E.        PUSHLOC
          RANGE                            DISPLAY(C);
          CALLSYS

F.        LOCALADD
          PUSHLOC
          PUSHCON
          ADDWORD                          I := I + 1;
          COPYWORD

G.        LOCALADD
          PUSHLOC
          PUSHLOC                          C := TEXT[I];
          INDEX
          PUSHBYTE
          COPYWORD

H.        JUMP                           END;

I.      EXIT
```

In this comparison the following parts A,E and I will be ignored as these are very much implementation dependent. The following is the equivalent Series/1 machine code that could be produced from a compiler doing some minimal optimization.

```
B.      MVWI    1,(LB,<offset>)

C.      MVB     (LB,<offset of TEXT[1]>),(LB,<offset of C>)

D.      MVBI    0,W1
        CB      (LB,<offset of C>),W1
        JE      <end of loop>

F.        AWI     1,(LB,<offset of I>)

G.        MVW     (LB,<offset of I>),W1
          IF (W1,LT,=1),OR,(W1,GT,=132),THEN
            B   ERRORROUTINE
          ENDIF
          AA      (LB,<offset of TEXT[1]>),W1
          MVB     (W1),(MP,<offset of C>)

H.        J       <to step D.>
```

To compare the two methods of implementation the following table has been constructed to show the relative sizes of the code produced whether it be actual machine code or the virtual code produced initially. The size of the Series/1 code generated is given as a range with the minimum and maximum values specified. It is represented in this manner as the size of an instruction maybe determined by unknown parameters, like the offset of a variable from the base register.

	Virtual Code Size (bytes)		Series/1 Code Size(bytes) minimum		maximum
A	10	(i)	-		-
B	10		4		6
C	24		6		6
D	14		8		10
E	14	(i)	-		-
F	16		6		6
G	24		12	(ii)	30
H	4		2		4
I	2	(i)	-		-
Total	90		38		62

(i) This value has been excluded from the Total.

(ii) This minimum value is without any array bounds checking.

Intermediate Language.

A symbolic form of intermediate language was chosen to make it as machine independent as possible. The actual format being very similar to Pascal's PCODE intermediate language with one instruction per physical line. The instructions themselves are also very similar in form to the original Virtual Code used in the interpreted version so as to minimise the conversion work required in the compiler.

Five basic instruction types make up the language, with the type of the instruction being designated by the first character on the line. The instruction types are as follows:

(i)	' '	Pseudo-machine instruction.
(ii)	'B'	Denotes an entry point into a block.
(iii)	'L'	Denotes a label which will be branched to.
(iv)	'P'	Denotes the size in bytes of the parameters of the block.
(v)	'S'	Gives the stack requirements of the block. Used for debugging only.

Instructions of type (ii) and (iii) have an associated integer parameter and their full composition is as follows:

'B' <integer> or 'L' <integer>

Where <integer> denotes either the block number or the label number.

Instructions of type (iv) or (v) have the following format:

```
        'P'     <blk. no>=     <value>
or      'S'     <blk. no>=     <value>
```

where <blk. no> and <value> are both integers. Here <blk. no> represents the block number and the <value> represents either the size of the parameters or the stack requirements of the block.

The pseudo machine instructions, represented by a blank in column one, use the next four characters on the line to distinguish the specific opcode. The length of the actual opcodes are anywhere from two or four characters but should they be less than four characters in length then they are blank padded out to the four character mark. Parameters may be appended to the instruction and will if present consist of single characters and/or integers. The pseudo-machine instructions are as follows:

. 1. Load and Store.

(a) Load Global Address(LDGA)

i.e. LDGA n { n = integer}

This instruction loads the effective address of GB+n onto the stack.

(b) Load Local Address(LDLA)

This is the same as LDGA except it uses the local base pointer(LB) rather than GB.

(c) Load Global(LDGL)

e.g. LDGLt n { t = character
n = integer }

This loads a global operand onto the stack from address (GB+n). The type of the operand is given by 't'.

Values of 't' are as follows:

'I' - integer
'B' - Boolean
'A' - Address
'R' - Real
'S' - Set

(d) Load Local(LDLO)

Same as LDGL except address of operand is (LB+n).

(e) Load Constant(LDCS)

e.g. LDCSt val { t = character
val = integer }

This instruction loads a constant of type t.

'I' val = integer
'R' val = real constant
'B' val = 1 (true) or 0(false)

(f) LOAD Constant Address(LDCA)

e.g. LDCA 'This is a string'

This instruction loads the address of the string that is passed as a parameter.

(g) COPY

e.g. COPYt

This instruction uses the two top of stack elements. The top of stack element will be a value of type 't' and the next to top of stack will be the address in which the top of stack is to be stored. On completion of this instruction the two top of stack elements are destroyed.

(h) MOVE

e.g. MOVE len { len = integer }

This also uses the two top of stack elements which must both be addresses. This instruction causes the string of length 'len' pointed to by the top of stack to be moved to the area pointed to be the next to top of stack.

(i) Store non-destuctive(STON)

e.g. STON n { n = integer }

This instruction is rather similar to the COPY opcode except that the next to top of stack is not distroyed on completion of this instruction. No type is given on this instruction as the top of stack must be of type ADDRESS. The destination address is evaluated from adding the constant 'n' to the next to top of stack element. The actual next to top of stack element is not changed though. This instruction is used to copy parameters specified on a INIT statement from the parent processes address space to the child processes address space.

(j) Stack Now(SNOW).

It is assumed that most translators will not stack operands unless necessary in the aim of optimizing code. This instruction forces the translator to stack all the operands presently on the stack. This instruction is used in respect to the CASE and WITH statements. In these cases the associated expressions are not physically stored anywhere by using one of the store instructions but are referenced on the stack as an offset from the local base register.

2. Addressing.

(a) Indirect(INDR)

e.g. INDRt {t = integer }

This instruction loads an operand of type 't' onto the stack, the address of which is the top of stack.

(b) Increment Address(INCA)

e.g. INCA n { n = integer }

This instruction adds the constant 'n' to the top of stack element which must be of type address.

(c) Indexing(INDX)

e.g. INDX n { n = integer }

This instruction is used for indexing into arrays. It uses the two top of stack elements and generates an address which is stacked from these two elements. The

top of stack element must be of type integer and the next to top of stack must be of type address. The address evaluated will have the following value.

value = (n * tos) + ntos

where n = constant parameter
tos = top of stack
ntos = next to top of stack

3. Integer Operations.

These arithmetic operators work on either one (unary operators) or two top of stack(binary operators) elements and leave the integer result on the stack.

(a) Binary Operators.

tos <- ntos <op> tos

operation	opcode
addition	ADDI
subtraction	SUBI
multiplication	MPYI
division	DIVI
modulus	MODI

(b) Unary Operators.

tos <- <op> tos

operation	opcode	
negation	NEGI	
predecessor	PRED	
decrement(by 1)	DECI	(same as PRED)
successor	SUCC	
increment(by 1)	INCA	(same as SUCC)
absolute	ABSI	

4. Real Operators.

Like the integer and boolean(following) operators these real operators work on either the top or top two stack elements. In this case though all the operands are of type real and the result that is stacked will also be of type real.

(a) Binary.

operation	opcode
addition	ADDR
subtraction	SUBR
multiplication	MPYR
division	DIVR

(b) Unary.

operation	opcode
negation	NEGR
absolute	ABSR

5. Boolean Operators.

(a) Binary.

operation	opcode
logical and	AND
logical or	OR

(b) Unary.

operation	opcode
logical not	NOT

6. Set Operators.

(a) Binary.

operation	opcode
intersection	INT
union	UNI
difference	DIFF

(b) Building Sets(BSET).

This operation takes the top of stack element which is of type integer and creates a set with the corresponding bit turned on and all other bits turned off.

7. Comparison Operators.

These operations use the top two elements on the stack and return a boolean value onto the stack. They operate on the two top of stack elements as follows:

tos <- ntos <op> tos

(a) Relational.

The format for this instruction type is as follows:

<op> t len { t = character, len = integer }

and the different opcodes are

LT - less than
GT - greater than
NEQ - not equal to
EQU - equal to
GEQ - greater than or equal to
LEQ - less than or equal to

The character t gives the type of the two elements to be compared and should the elements be of type array or type record(i.e. t = 'M') then the integer len will be

present which represents the length of the operands.

(b) IN operator.

This operator requires the top of stack to be of type set and the next to top of stack to be of type integer. This operator returns true if the in contained in the set otherwise false is stacked.

8. Jump and Call Opcodes.

(a) Unconditional Jump.

e.g. JUMP L <labnum>

This causes an unconditional jump to label L <labnum>.

(b) Conditional Jump.

e.g. FJMP L <labnum>

This instruction tests the boolean value that is on the top of stack and jumps to label L <labnum> if it is false otherwise it carries on to the next instruction.

(c) Case Jump(CASE).

This instruction uses the top of stack value which is of type integer and uses it to generate a jump to one of n different addresses. The n addresses directly follow this instruction.

(d) Call Instruction(CALL)

e.g. CALL B <blk. no>

This instruction initiates a call to a procedure or function. It must also save the return address to be used on completion of the procedure.

(e) Call Program(PROG)

This instruction whose format is as yet undecided will ultimately allow the system to start USER programs.

(f) Initiation(INIT)

This instruction initiates a sub-process which will execute assynchronously with the issuing process.

9. Entry and Exit.

(a) Entry Statement(ENT)

This instruction identifies the start of a procedural block. It is used to perform the stack housekeeping that may be required.

(b) Entry to Process statement(ENTP).

This instruction is similar to ENT but is used when entering a Process block.

(c) Entry to Class Statement(ENTC). This instruction is again similar to ENT but is used when entering a class or monitor block.

(d) Exit Statement(EXIT).

This instruction marks the end of a process. When it is executed the process is terminated and it's resources are deallocated.

(e) Return Statement(RETN).

This instruction marks the end of a procedural block and causes a return to the line of code where the procedure was called from.

10. Routine and Resource Handling.

(a) Base statement.

e.g. BASEt { t = character }

The parameter t can have two different values.

'L' - Load or 'R' - Restore

'L' causes the current global base pointer to be loaded with the value on top of the stack. 'R' causes the global base pointer to be restored to the previous value, this value having been stored during a mark stack operation.

(b) Gate instruction.

e.g. GATEt { t = character }

This instruction controls access to monitor routines.

The parameter t has three permissible values.

'E' - monitor entry
'I' - monitor initialisation
'L' - monitor exit

The 'I' parameter causes the monitor's gate to be

allocated and initialised. The parameter 'E' is a request from the current process to enter the monitor and the 'L' is a notification from the running process that it is leaving the monitor.

(c) Continue Statement(CONT)

This instruction is the primitive corresponding to the CONT statement in concurrent pascal, used to continue a delayed process.

(d) Wait Statement(WAIT)

This instruction is the primitive corresponding to the WAIT statement in concurrent pascal.

(e) Stop Statement(STOP)

This instruction is the primitive corresponding to the standard procedure STOP in concurrent pascal.

(f) Delay Statement(DLAY)

This instruction is the primitive corresponding to the DELAY procedure in concurrent pascal.

(g) New Statement(NEW).

i.e. NEW <blk. no>

This instruction allocates a new block of memory for a child process that is about to be started. This block of memory will be used as that process's data space.

11. Stack Handling.

(a) Function Return(FUNC)

i.e. FUNCt { t = character }

This instruction saves space on the top of the stack, in which the function will store the value it returns. The character t represents the type of the function.

(b) Marking of Stack(MKST)

This instruction marks the stack by moving the base registers prior to a procedural call.

(c) POP instruction.

e.g. POP n {n = integer }

This instruction causes n words to be popped off the stack.

12. Diagnostic Instructions.

(a) LINE instruction

i.e. LINE n {n = integer }

This signifies that we are now executing code from source line 'n'.

(b) Range Instruction

i.e. RNGE m n { m,n = integer }

This tests to validate that the top of stack element which is of type integer is within the bounds of the two parameters m and n. (i.e. m >= val >= n). An error is generated if this is not the case.

13. Conversion Instructions.

(a) Ord Function(ORD).

This instruction corresponds to the pascal primitive function ORD. It takes the top of stack, in character form, and changes it's type to that of an integer.

(b) Chr Function(CHR).

The instruction corresponds to the pascal primitive function CHR. It takes the top of stack, in integer form, and changes it's type to that of a character.

14. IO operator(IO).

This instruction corresponds to the concurrent pascal primitive procedure IO. It calls the kernel to perform the desired input-output.

System Control Language Syntax.

The following commands are recognised by MUSCLE at command level. The underlined portions of these commands are compulsory, whereas all the other components are optional and may be leftout.

1. COPYFILE (NAME = <filename> , NEWNAME = <filename>)
2. CREATEFILE (NAME = <filename>)
3. DELETEFILE (NAME = <filename>)
4. EDITFILE (NAME = <filename>)
5. LISTDIRECTORY
6. LOGIN (USER = <username>)
7. LOGOUT
8. PARSEFILE (NAME = <filename>)

Where <filename> consists of a letter followed by up to fifteen alphanumeric characters.

The command COPYFILE (may be abbreviated to COPY) copies the contents of the first file into a new file which is specified by the second filename. An error is flagged if either the first file doesn't exist or if the second one does already exists.

The command CREATEFILE (CF) creates an empty file and then calls the editor. The editor will automatically enter insert mode, thereby allowing the user to write text directly into the workfile. An error is flagged should the file already exist.

The command DELETEFILE (DF) will delete the file and return the now free disk blocks back to the free disk area. An error is flagged if the file does not exist.

The command EDITFILE (EF) tests to see if the specified file exists and then calls the editor so that the file may be edited. The editor will stay in command level so that editor commands may be used. An error is flagged if the file doesn't already exist.

The command LISTDIRECTORY (LD) lists the current directory. This displays all files that the user has created.

The command LOGIN is used by a user to initiate a terminal session. This command checks that the username given is a valid name and if so attaches the user to the directory of the same name. The structure of <username> is identical to that of <filename>. An error is flagged if the username given is not included in a list of valid usernames.

The LOGOUT (LO) command is used to terminate a terminal session. This command places the terminal in login mode, which means that the only command accepted will be that of the LOGIN command.

The command PARSEFILE (PF) is used to parse the pascal program contained in the specified file. This command will display any errors found in the program on the screen. An error is flagged if the specified filename doesn't exist.

The editor used in the system is written in Pascal. It is a memory based editor in that the entire work-file is stored in memory. The disk copy of the file is only updated when the STOP command is encountered. For a full description of the editor, refer to the textbook "Advanced Programming and Problem Solving with Pascal" [Schneider & Bruell 81].

Concurrent Pascal Source of MUSCLE.

The following appendix contains some selected sections of the Concurrent Pascal source from the MUSCLE operating system. This source comes from the following three areas of the system:

1. Screen Handling Logic
2. Memory Management Logic
3. User Interface to system

Screen Handling Logic in MUSCLE

The following three modules compose the support MUSCLES offers to screens. The terminals connected to the Series/1 require that a process is continually waiting for an interrupt from them. For this reason there is a process in MUSCLE dedicated to this task. This process is an instance of the process SCREEN_MANAGER.

When a user wishes to perform I/O on a terminal, a call to the monitor SCREEN_DRIVER is made. There is one instance of this monitor for every screen supported on the system. Output to the terminal is handled directly with a call to IO. Should input from the terminal be required though, the user process will be delayed until an interrupt is received from the terminal. In this case the a call to the SCREEN_INTERRUPTS monitor is made. If there is no pending interrupts of the right kind then the user will be delayed on a QUEUE. It is the task of the SCREEN_MANAGER to receive these interrupts and pass them on to the user processes via the monitor SCREEN_INTERRUPTS.

```
CONST  NOUSERS    =  4;

TYPE   KEY_TYPE   =  (ENTER, ATTN, PF1, PF4, PF2, PF5, PF3, PF6, NULL);

TYPE SCREEN_INTERRUPTS = MONITOR

  VAR  KEYS      : ARRAY[1..NOUSERS] OF KEY_TYPE;
       EVENTLIST: ARRAY[1..NOUSERS] OF QUEUE;
       CNT       : INTEGER;

PROCEDURE ENTRY ENTER_WAIT(USER: INTEGER);
  VAR KEY: KEY_TYPE;
BEGIN
  REPEAT
    KEY := KEYS[USER];
    KEYS[USER] := NULL;

    IF KEY = NULL
    THEN DELAY(EVENTLIST[USER]);
  UNTIL KEY = ENTER;
END (* Wait *);

PROCEDURE ENTRY KEY_WAIT(VAR KEY : KEY_TYPE;
                             USER: INTEGER );
BEGIN
  REPEAT
    KEY := KEYS[USER];
    KEYS[USER] := NULL;

    IF KEY = NULL
    THEN DELAY(EVENTLIST[USER]);
  UNTIL KEY <> NULL;
END (* Key Wait *);

PROCEDURE ENTRY POSTINTERRUPT(USER: INTEGER;
                              KEY : KEY_TYPE);
BEGIN
  KEYS[USER] := KEY;

  CONTINUE(EVENTLIST[USER])
END (* Post Interrupt *);

BEGIN
  FOR CNT := 1 TO NOUSERS DO
    KEYS[CNT] := NULL;
END (* Screen Enter Event *);
```

```
TYPE SCREEN_DRIVER = MONITOR(SCREEN_INT: SCREEN_INTERRUPTS);

  TYPE  IOPARAM =  RECORD
                     OPERATION : IOOPERATION;
                     STATUS    : IORESULT;
                     USER      : INTEGER;
                     LINELENGTH: INTEGER
                   END;

  VAR   PARAM  :  IOPARAM;


  PROCEDURE TRANSFER(VAR LINE   : LINETYPE);
  BEGIN
    WITH PARAM DO BEGIN
      LINELENGTH := 80;

      IO(LINE, PARAM, SCREEN);
      WHILE STATUS <> COMPLETE DO BEGIN
        IO(LINE, PARAM, SCREEN);
      END;
    END;
  END (* Transfer *);


PROCEDURE ENTRY READ(VAR LINE     : LINETYPE;
                         USER     : INTEGER );
BEGIN
  SCREEN_INT.ENTER_WAIT(USER);

  PARAM.USER := USER;
  PARAM.OPERATION := INPUT;

  TRANSFER(LINE);
END (* Read *);

PROCEDURE ENTRY WRITE(VAR LINE     : LINETYPE;
                          USER     : INTEGER );
BEGIN
  PARAM.USER := USER;
  PARAM.OPERATION := OUTPUT;

  TRANSFER(LINE);
END (* Write *);

BEGIN
END (* Screen Driver *);
```

```
TYPE SCREEN_MANAGER = PROCESS(SCREEN_INT: SCREEN_INTERRUPTS);

  TYPE  IOPARAM =  RECORD
                     OPERATION : IOOPERATION;
                     STATUS    : IORESULT;
                     USER      : INTEGER;
                     KEY       : KEY_TYPE
                   END;

  VAR   PARAM   :  IOPARAM;
        DUMMY   :  CHAR;


  PROCEDURE HANDLESCREENS;
  BEGIN
    PARAM.OPERATION := WAIT_INTERRUPT;

    CYCLE                    (* Continue for life of system.   *)
      IO(DUMMY, PARAM, SCREEN);    (* Wait for interrupt from any terminal *)

                                        (* Signal appropriate user *)
      SCREEN_INT.POSTINTERRUPT(PARAM).USER, PARAM.KEY)
    END;
  END (* Handle Screens *);

BEGIN
  HANDLESCREENS;
END;
```

Memory Management Section

The following routines handle the memory management system for for both the system itself and also user programs. The method used is based on multi user, variable sized partitions with no rollout. Should memory not be available then the requesting process is delayed until that memory becomes available.

The memory management is further described in section 3.2 of this thesis.

```
CONST SHAREDSIZE = 1;  (* Environment is 2K bytes long *)
      DATASIZE   = 9;  (* The data requires 18K bytes  *)
      PROGSTART  = 10; (* The program code starts after the data *)


TYPE MEMORY_MANAGER = MONITOR

  CONST  MAXPAGES= 31;  (*  Only allow a program to use up to 64K  *)
         NUMSYSPAGES = 32;
         NUMMACHPAGES= 128;

  TYPE   ADDRESSSPACE = ARRAY[0..MAXPAGES] OF INTEGER;
         INTSET       = SET OF 0..15;
         JOBTYPE      = RECORD
                          NOINUSE: INTEGER;
                          PROG   : INTEGER;
                          PAGES  : ADDRESSSPACE
                        END;
         PROG_DATA    = RECORD
                          INUSE     : BOOLEAN;
                          FILELOADED: BOOLEAN;
                          USER_CNT  : INTEGER;
                          ADDR      : DISKPAGE;
                          PAGES     : ADDRESSSPACE
                        END;
         SCHEDULEDTYPE= RECORD
                          INUSE: BOOLEAN;
                          PROC : QUEUE
                        END;
         WAITTYPE     = RECORD
                          FILEADD: DISKPAGE;
                          PROC   : QUEUE
                        END;

  VAR    JOBS     : ARRAY[0..NOUSERS] OF JOBTYPE;
         RES_PROGS: ARRAY[0..NOUSERS] OF PROG_DATA;
         SCHEDULED: ARRAY[0..NOUSERS] OF SCHEDULEDTYPE;
         WAITING  : ARRAY[0..NOUSERS] OF WAITTYPE;

         FREE     : ARRAY[0..7] OF INTSET;
         NUMFREE  : INTEGER;
         I        : INTEGER;
```

```
FUNCTION GETFREEBLOCK: INTEGER;
  VAR CNT: INTEGER;
BEGIN                    (* Find a free block, and remove it from freelist *)
  CNT := NUMSYSPAGES;
  WHILE NOT ((CNT MOD 16) IN FREE[CNT DIV 16])
  DO CNT := CNT + 1;

  FREE[CNT DIV 16] := FREE[CNT DIV 16] - [CNT MOD 16];
  NUMFREE := NUMFREE - 1;
  GETFREEBLOCK := CNT;
END (* Get Free Block *);


PROCEDURE RELEASEBLOCK(PAGE: INTEGER);
BEGIN                  (* Block is no longer needed, return to freelist *)
  FREE[PAGE DIV 16] := FREE[PAGE DIV 16] OR [PAGE MOD 16];

  NUMFREE := NUMFREE + 1;
END (* Release Block *);


PROCEDURE SHARE;
  VAR  PARAM: IOPARAM;
BEGIN                  (* This routine initially shares the runtime support *)
  WITH JOBS[1] DO BEGIN
    FOR I := 0 TO SHAREDSIZE - 1 DO
      PAGES[I] := GETFREEBLOCK*8 + 6;

    FOR I := SHAREDSIZE TO MAXPAGES DO
      PAGES[I] := 0;

    PARAM.ARG := SYSTEM+1;
    IO(PAGES, PARAM, SETMEMORY);
  END;

  FOR I := 2 TO NOUSERS DO
    JOBS[I] := JOBS[1];
END (* Share *);
```

```
PROCEDURE ENTRY ALLOCATE(     USER     : INTEGER;
                              NUMPAGES: INTEGER;
                              FILEADDR: DISKPAGE;
                              PROGONLY: BOOLEAN;
                          VAR LOADED   : BOOLEAN);
  VAR  CNT,INDX   : INTEGER;
       PARAM      : IOPARAM;
       PAGES_REQD: INTEGER;
BEGIN                        (* Allocate memory for a user's program *)
  LOADED := FALSE;           (* Assume not already loaded in memory *)

  FOR CNT := 0 TO NOUSERS DO
    WITH RES_PROGS[CNT] DO
      IF INUSE AND (ADDR = FILEADDR) THEN BEGIN  (* Already in Memory *)
        LOADED := TRUE;
        INDX := CNT;
      END;

  IF LOADED THEN BEGIN
    IF PROGONLY
    THEN PAGES_REQD := 0
    ELSE PAGES_REQD := DATASIZE;     (* Only allocate for data *)

    IF NOT RES_PROGS[INDX].FILELOADED THEN BEGIN
       WAITING[USER].FILEADD := FILEADDR;
       DELAY(WAITING[USER].PROC)
    END
  END ELSE BEGIN
    IF PROGONLY
    THEN PAGES_REQD := NUMPAGES
    ELSE PAGES_REQD := NUMPAGES + DATASIZE;

    INDX := 0;
    WHILE RES_PROGS[INDX].INUSE DO INDX := INDX + 1;

    WITH RES_PROGS[INDX] DO BEGIN
      INUSE := TRUE;
      FILELOADED := FALSE;
      USER_CNT := 1;
      ADDR := FILEADDR;
    END;
  END;
```

```
  IF PAGES_REQD > NUMFREE THEN BEGIN (* Schedule job until enough space *)
    I := 0;
    WHILE SCHEDULED[I].INUSE DO I := I + 1;

    WITH SCHEDULED[I] DO BEGIN
      INUSE := TRUE;
      DELAY(PROC);
    END;
  END;

  WITH JOBS[USER - SYSTEM] DO BEGIN
    PROG := INDX;
    NOINUSE := PROGSTART + NUMPAGES;

    FOR I := 0 TO PAGES_REQD - 1 DO BEGIN
      IF PROGONLY
      THEN CNT := PROGSTART + I
      ELSE CNT := SHAREDSIZE + I;

      IF CNT >= PROGSTART
      THEN PAGES[CNT] := GETFREEBLOCK*8 + 6   (* Program is read only *)
      ELSE PAGES[CNT] := GETFREEBLOCK*8 + 4;  (* Data is read/write   *)
    END;

    IF LOADED THEN BEGIN
      FOR CNT := PROGSTART TO (PROGSTART + NUMPAGES - 1) DO
        PAGES[CNT] := RES_PROGS[INDX].PAGES[CNT];

      RES_PROGS[INDX].USER_CNT := RES_PROGS[INDX].USER_CNT + 1;
    END ELSE RES_PROGS[INDX].PAGES := PAGES;

    FOR I := (PROGSTART + NUMPAGES) TO MAXPAGES DO
      PAGES[I] := 0;      (*  These pages are invalid  *)

    PARAM.ARG := USER;
    IO(PAGES, PARAM, SETMEMORY);
  END
END (* Allocate memory *);
```

```
  PROCEDURE ENTRY LOADEDPROG(FILEADDR: DISKPAGE);
    VAR  CNT: INTEGER;
  BEGIN
    FOR CNT := 0 TO NOUSERS DO
      WITH RES_PROGS[CNT] DO
        IF INUSE AND (ADDR = FILEADDR)
        THEN FILELOADED := TRUE;

    FOR CNT := 0 TO NOUSERS DO
      IF WAITING[CNT].FILEADD = FILEADDR THEN BEGIN
        WAITING[CNT].FILEADD := NIL;
        CONTINUE(WAITING[CNT].PROC);
      END;
  END (* Program Loaded *);


 PROCEDURE ENTRY RELEASE(USER: INTEGER; PROGONLY: BOOLEAN);
    VAR  LASTPAGE : INTEGER;
         FIRSTPAGE: INTEGER;
  BEGIN
    WITH JOBS[USER - SYSTEM] DO BEGIN
      RES_PROGS[PROG].USER_CNT := RES_PROGS[PROG].USER_CNT - 1;
      IF RES_PROGS[PROG].USER_CNT = 0 THEN BEGIN (*Program no longer in use*)
        RES_PROGS[PROG].INUSE := FALSE;
        LASTPAGE := NOINUSE - 1;
      END ELSE LASTPAGE := PROGSTART - 1;

      IF PROGONLY
      THEN FIRSTPAGE := PROGSTART
      ELSE FIRSTPAGE := SHAREDSIZE;

      FOR I := FIRSTPAGE TO LASTPAGE DO
        RELEASEBLOCK(PAGES[I] DIV 8)
    END
  END (* Release memory *);


BEGIN
  FOR I := 0 TO NOUSERS DO
    RES_PROGS[I].INUSE := FALSE;

  NUMFREE := NUMMACHPAGES - NUMSYSPAGES;
  FOR I := NUMSYSPAGES TO (NUMMACHPAGES - 1) DO
                                  (* All non-system pages initially free *)
    FREE[I DIV 16] := FREE[I DIV 16] OR [I MOD 16];

  SHARE;
END (* Memory Manager *);
```

Virtual Machine supporting the User

This class handles the interface between the user and the operating system. It does the initial checking of system calls and finding no errors passes them on to the actual processing routines.

This process supports thirteen supervisor calls. These are as follows:

OPEN	-	Open a file for I/O
READ	-	Read a logical line from a file
WRITE	-	Write a logical line to a file
GETPAGE	-	Read a physical block from a disk file
CLOSE	-	Close a file
FILEEXISTS	-	Test to see is a certain file exists
DELETEFILE	-	Delete a file
ATTACH	-	Attach to a disk directory
RUNPROG	-	Initiate a new program
LOADOLAY	-	Load a new overlay in the current program
EXITPROG	-	Exit this program and return to previous one
SETPARM	-	Load parameter value. To pass to other program
GETPARM	-	Get parameter value that was set by other prog.

```
TYPE USERMACHINE = PROCESS(ACTIVE_FILES: ACTIVE_FILETABLE;
                           MEMORY_      : MEMORY_MANAGER;
                           BUFFERS      : DISKBUFFERS;
                           BUFFER       : BUFFER_POOL;
                           FREE         : FREEBLOCKS;
                           DIRECTORY    : DISK_DIRECTORY;
                           DISPLAY      : SCREEN_DRIVER;
                           USER         : INTEGER         );

  TYPE IO_DIRN   = (TOSYSTEM, FROMSYSTEM);
       IOPARAM   = RECORD
                     BUFFER_OFFSET: INTEGER;
                     BUFFER_LENGTH: INTEGER;
                     ARG          : INTEGER;
                     DIRECTION    : IO_DIRN
                   END;

  VAR
       FILENAME     : TREENAME;

       BFILE        : BASIC_FILESYSTEM;
       LFILE        : LOGICAL_FILESYSTEM;
       PROG         : PROGRAM_FILE;
       LSB          : LEVEL_STATUS_BLOCK;

       PARAM        : IOPARAM;
       TOKEN        : (NULLTYPE, GTTYPE, IDENTTYPE);
       NAME         : NAMETYPE;
       LEN          : INTEGER;
       CHCNT        : INTEGER;
       BADNAME      : BOOLEAN;
       OPENMODE     : OPENMODES;
       OPENTYPE     : INTEGER;

       GLOBAL_DIR   : DISKADDRESS;
       CURRENT_DIR  : DISKADDRESS;

       FULLFILE     : TREE_STRUCTURE;
       LEVEL        : INTEGER;
       PARAMETER    : TREENAME;
       CODE         : INTEGER;
```

```
PROCEDURE COPYNAME(VAR NAME: TREENAME);
BEGIN                    (* This copies string parameters to system from user *)
  PARAM.DIRECTION := TOSYSTEM;
  IO(NAME, PARAM, DATAMOVE);
END (* COPY NAME *);


PROCEDURE GETTOKEN;
  VAR  CH: CHAR;
       I : INTEGER;
BEGIN                    (* This procedure gets the next token from pathname *)
  CH := FILENAME[CHCNT];

  IF CH = ' '
  THEN TOKEN := NULLTYPE          (*  EOLN reached  *)
  ELSE
  IF CH = '>' THEN BEGIN          (*  PREVIOUS PART WAS DIRECTORY *)
    TOKEN := GTTYPE;
    CHCNT := CHCNT + 1;
  END ELSE
  IF CH = '*' THEN BEGIN          (*  POINTER TO CURRENT DIRECTORY *)
    TOKEN := IDENTTYPE;
    NAME := '*               ';
    CHCNT := CHCNT + 1
  END ELSE
  IF (CH >= 'A') AND (CH <= 'Z') THEN BEGIN
    TOKEN := IDENTTYPE;
    LEN := 0;
    REPEAT
      LEN := LEN + 1;
      IF LEN <= NAMELEN
      THEN NAME[LEN] := CH
      ELSE BADNAME := TRUE;

      CHCNT := CHCNT + 1;
      IF CHCNT <= TREELEN
      THEN CH := FILENAME[CHCNT];

    UNTIL ((CH < 'A') OR (CH > 'Z')) AND ((CH < '0') OR (CH > '9'));

    FOR I := LEN+1 TO NAMELEN
    DO NAME[I] := ' ';
  END ELSE BADNAME := TRUE;
END;
```

```
FUNCTION BAD_TREENAME(LINE: TREENAME): BOOLEAN;

  (*************************************************************************)
  (*    This function breaks the treename given in line up into its        *)
  (*  individual parts. It checks the format to verify that it is valid.   *)
  (*    The valid format is as follows:                                    *)
  (*                                                                       *)
  (*                    { filename> }  filename                            *)
  (*                                                                       *)
  (*************************************************************************)

BEGIN
  CHCNT := 1;
  BADNAME := FALSE;
  LEVEL := 0;

  REPEAT
    GETTOKEN;

    IF TOKEN <> IDENTTYPE THEN BEGIN
      BADNAME := TRUE;
      TOKEN := NULLTYPE;
    END ELSE BEGIN
      LEVEL := LEVEL + 1;
      FULLFILE[LEVEL] := NAME;

      GETTOKEN
    END
  UNTIL TOKEN <> GTTYPE;

  IF TOKEN <> NULLTYPE
  THEN BADNAME := TRUE;

  BAD_TREENAME := BADNAME;
END (* Bad treename *);

FUNCTION BAD_ACTION(ACTION: INTEGER): BOOLEAN;
BEGIN    (* This procedure checks the validity of an integer parameter *)
  IF (ACTION < 0) OR (ACTION > 1)
  THEN BAD_ACTION := TRUE
  ELSE BEGIN
    OPENTYPE := ACTION MOD 4;
    CASE ACTION OF
      0: OPENMODE := READING;
      1: OPENMODE := WRITING
    END;

    BAD_ACTION := FALSE;
  END;
END (* Bad Action *);
```

```
PROGRAM JOB(LSB: LEVEL STATUS BLOCK);
ENTRY OPEN, READ, WRITE, CLOSE, RUNPROG, EXITPROG, SETPARM, GETPARM,
      FILEEXISTS, GETPAGE, DELETEFILE, ATTACH, LOADOLAY;


PROCEDURE ENTERJOB;
BEGIN
  JOB(LSB)
END (* Enter Job *);


PROCEDURE CALLPROG(FOPEN: BOOLEAN; FUNIT: INTEGER);
  VAR UNIT: INTEGER;
BEGIN
  IF FOPEN
  THEN UNIT := FUNIT
  ELSE LFILE.OPEN(GLOBAL_DIR, CURRENT_DIR, READING, FULLFILE, LEVEL,
                  UNIT, CODE);
  IF CODE = 0 THEN BEGIN
    PROG.CALL(UNIT, FULLFILE, LSB, CODE);
    IF CODE = 0 THEN BEGIN
      BFILE.CLOSE(UNIT, CODE);
      IF CODE = 0
      THEN ENTERJOB;
    END
  END
END (* Call Prog *);


PROCEDURE ENTRY OPEN(    ACTION  : INTEGER;
                     VAR NAME    : TREENAME);
  VAR FILEUNIT: INTEGER;
BEGIN              (* Handles the OPEN request from user *)
  CODE := 0;
  COPYNAME(NAME);

  IF BAD_TREENAME(FILENAME)
  THEN CODE := 1
  ELSE
  IF BAD_ACTION(ACTION)
  THEN CODE := 2
  ELSE LFILE.OPEN(GLOBAL_DIR, CURRENT_DIR, OPENMODE, FULLFILE, LEVEL,
                  FILEUNIT, CODE);

  PROG.SETREG (LSB, CODE, 0);     (* Set register zero with return code *)
  PROG.SETREG(LSB, FILEUNIT, 1); (* Place file unit into register one  *)
END (* Open *);
```

```
PROCEDURE ENTRY GETPAGE(    UNIT : INTEGER;
                        VAR BLOCK: BLOCKTYPE);
BEGIN               (* Handles the GETPAGE request from user *)
  CODE := 0;
  BFILE.GETPAGE(UNIT, USER, BLOCK, CODE);

  PROG.SETREG(LSB, CODE, 0);
END (* Get Page *);


PROCEDURE ENTRY READ(    UNIT: INTEGER;
                     VAR LINE: LINETYPE);
BEGIN               (* Handles the READ request from user *)
  CODE := 0;
  BFILE.READ(UNIT, USER, LINE, CODE);

  PROG.SETREG (LSB, CODE, 0);    (* Set register zero with return code *)
END (* Read *);


PROCEDURE ENTRY WRITE(    UNIT: INTEGER;
                      VAR LINE: LINETYPE);
BEGIN               (* Handles the WRITE request from user *)
  CODE := 0;
  BFILE.WRITE(UNIT, USER, LINE, CODE);

  PROG.SETREG (LSB, CODE, 0);    (* Set register zero with return code *)
END (* Write *);


PROCEDURE ENTRY CLOSE(    UNIT: INTEGER);
BEGIN               (* Handles the CLOSE request from user *)
  CODE := 0;
  BFILE.CLOSE(UNIT, CODE);

  PROG.SETREG(LSB, CODE, 0);    (* Set register zero with return code *)
END (* Close *);
```

```
PROCEDURE ENTRY RUNPROG(VAR PROGNAME: TREENAME);
  VAR UNIT: INTEGER;
BEGIN                 (* Handles the RUNPROG request from user *)
  CODE := 0;
  COPYNAME(PROGNAME);

  IF BAD TREENAME(FILENAME)
  THEN CODE := 1
  ELSE LFILE.OPEN(GLOBAL_DIR, CURRENT_DIR, READING, FULLFILE, LEVEL,
                  UNIT, CODE);

  IF CODE = 0 THEN BEGIN
    PROG.SAVE(LSB, CODE);

    IF CODE = 0
    THEN CALLPROG(TRUE, UNIT);
  END;

  PROG.SETREG(LSB, CODE, 0);
END (* Runprog *);


PROCEDURE ENTRY LOADOLAY(VAR PROGNAME: TREENAME);
  VAR UNIT: INTEGER;
BEGIN                 (* Handles the LOADOLAY request from user *)
  CODE := 0;
  COPYNAME(PROGNAME);

  IF BAD TREENAME(FILENAME)
  THEN CODE := 1
  ELSE LFILE.OPEN(GLOBAL_DIR, CURRENT_DIR, READING, FULLFILE, LEVEL,
                  UNIT, CODE);

  IF CODE = 0 THEN BEGIN
    LSB.IAR := 0;
    PROG.LOADOLAY(UNIT, CODE);
    IF CODE = 0
    THEN BFILE.CLOSE(UNIT, CODE)
  END;

  PROG.SETREG(LSB, CODE, 0);
END (* Load Overlay *);
```

```
PROCEDURE ENTRY EXITPROG;
  VAR UNIT: INTEGER;
BEGIN                 (* Handles the EXITPROG request from user *)
  PROG.RESTORE(LSB);          (* Restore previous program off diskette *)
  PROG.SETREG(LSB, CODE, 0); (* Signal no error *)

  ENTERJOB
END (* Exit Program *);


PROCEDURE ENTRY FILEEXISTS(VAR NAME    : TREENAME);
BEGIN                 (* Handles the FILEEXISTS request from user *)
  CODE := 0;
  COPYNAME(NAME);

  IF BAD_TREENAME(FILENAME)
  THEN CODE := 1
  ELSE LFILE.FILEEXISTS(GLOBAL_DIR, CURRENT_DIR, FULLFILE, LEVEL, CODE);

  PROG.SETREG (LSB, CODE, 0);    (* Set register zero with return code *)
END (* File Exists *);


PROCEDURE ENTRY DELETEFILE(VAR NAME    : TREENAME);
BEGIN                 (* Handles the DELETEFILE request from user *)
  CODE := 0;
  COPYNAME(NAME);

  IF BAD_TREENAME(FILENAME)
  THEN CODE := 1
  ELSE LFILE.DELETEFILE(GLOBAL_DIR, CURRENT_DIR, FULLFILE, LEVEL, CODE);

  PROG.SETREG (LSB, CODE, 0);    (* Set register zero with return code *)
END (* Delete File *);


PROCEDURE ENTRY ATTACH(VAR NAME    : TREENAME);
BEGIN                 (* Handles the ATTACH request from user *)
  CODE := 0;
  COPYNAME(NAME);

  IF BAD_TREENAME(FILENAME)
  THEN CODE := 1
  ELSE LFILE.ATTACH(GLOBAL_DIR, CURRENT_DIR, FULLFILE, LEVEL, CODE);

  PROG.SETREG (LSB, CODE, 0);    (* Set register zero with return code *)
END (* Attach to Directory *);
```

```
PROCEDURE ENTRY SETPARM(VAR PARM: TREENAME);
BEGIN                (* Handles the SETPARM request from user *)
  COPYNAME(PARM);
  PARAMETER := FILENAME;

  PROG.SETREG(LSB, 0, 0);
END (* Set Parameter *);


PROCEDURE ENTRY GETPARM(VAR PARM: TREENAME);
BEGIN                (* Handles the GETPARM request from user *)
  FILENAME := PARAMETER;

  PARAM.DIRECTION := FROMSYSTEM;
  IO(PARM, PARAM, DATAMOVE);

  PROG.SETREG(LSB, 0, 0);
END (* Get Parameter *);


BEGIN
  INIT
       BFILE(ACTIVE_FILES, BUFFERS, BUFFER, DISPLAY, FREE),
       LFILE(BFILE, FREE, DIRECTORY),
       PROG(MEMORY, BFILE, LFILE, USER);


  PARAM.BUFFER_OFFSET := 16;
  PARAM.BUFFER_LENGTH := 80;
  PARAM.ARG := USER;

  WITH CURRENT_DIR DO BEGIN
    DEV := DISK;
    PAGENO := 80
  END;
  GLOBAL_DIR := CURRENT_DIR;

  CODE := 0;
  FULLFILE[1] := 'COMMANDS        ';
  FULLFILE[2] := 'LOGIN           ';
  LEVEL := 2;
  CALLPROG(FALSE, 0);
END;
```

[Aho and Ullman 77]

Aho A, and Ullman J.

Principles of Compiler Design.

Addison-Wesley, Reading, Massachusetts(1977).

[Bates and Douglas 75]

Bates, F. and Douglas, M. L.

Programming Language/1 with Structured Programming.

Prentice-Hall, Englewood Cliffs, New Jersey(1975).

[Boswell et al. 83]

Boswell F. D., Carmody M. J. and Grove T. R.

A String Extension for Pascal.

Sigplan Notices, Vol 18 (1983), No 2 pp. 57-61.

[Brinch Hansen 75]

Brinch Hansen, P.

The Programming Language Concurrent Pascal.

IEEE Transactions on Software Engineering,

Vol SE-1 (1975), No 2.

[Brinch Hansen 76]

Brinch Hansen, P.

The Solo Operating System:

Processes, Monitors and Classes.

Software - Practice and Experience,

Vol 6 (1976), pp. 165-200.

[Brinch Hansen 77a]

Brinch Hansen, P.

The Architecture of Concurrent Programs.

Prentice-Hall, Englewood Cliffs, New Jersey(1977).

[Brinch Hansen 77b]

Brinch Hansen, P.

Experience with Modular Concurrent Programming.

IEEE Transactions on Software Engineering,

Vol SE-3 (1977), No. 2.

[Coleman et al. 79]

Coleman D, Gallimore R, Hughes J. and Powell M. S.

An Assessment of Concurrent Pascal

Software - Practice and Experience,

Vol 9 (1979), pp. 827-837.

[Graef et al. 79]

Graef N, Kretschmar H, Loehr K and Morawetz B.

How to Design and Implement Small Time-sharing

Systems using Concurrent Pascal.

Software - Practice and Experience, Vol 9 (1979), pp. 17-24.

[Haddon 77]

Haddon, B. K.

Nested Monitor Calls.

Operating Systems Review, Vol 11 (1977), No. 4 pp. 18-23.

[Heimberger 78]

Heimberger, D.

Writing Device Drivers for Concurrent Pascal.

Operating Systems Review, Vol 12 (1978), No. 4 pp. 16-33.

[Hoare 73]

Hoare, C. A. R.

A Structured Paging System.

Computer Journal, Vol 16 (1973), pp. 209-214.

[Hoare 74]

Hoare, C. A. R.

Monitors: An Operating System Structuring Concept

CACM, Vol 17 (1974), no. 10 pp. 549-557.

[Holt et al. 78]

Holt R.C., Graham G.S., Lazowska E.D. and Scott M.A.

Structured Concurrent Programming

with Operating Systems Applications.

Addison-Wesley, Reading, Massachusetts(1978).

[Jensen and Wirth 78]

Jensen K. and Wirth N.

PASCAL User Manual and Report.

Springer-Verlag, New York(1978).

[Lagally 78]

Lagally, K. (and others).

Lecture Notes in Computer Science 60.

Advanced Course in Operating Systems.

Springer-Verlag, Berlin(1978).

[Lister 77]

Lister, A. M.

The problems of nested monitor calls.

Operating Systems Review, Vol 11 (1977), No. 3 pp. 5-7.

[Lister 78]

Lister, A. M.

Operating Systems Review, Vol 12 (1978), No. 1 p. 15.

[Madnick and Alsop 69]

Madnick, S. E. and Alsop, J. W.

A modular approach to file system design.

Proc. AFIPS, Vol 34 (1969), pp. 1-14.

[McCormack and Gleaves 83]

McCormack, J. and Gleaves, R.

MODULA-2 A Worthy Successor to Pascal.

BYTE Publications, April 1983.

[McKendry and Campbell 80]

McKendry, M. S. and Campbell, R. H.

Capabilities for high level Languages.

Tech. Report UIUCDCS-R-80-1016.

University of Illinois, Illinois(1980).

[Neal and Wallentine 78]

Neal, D and Wallentine, V.

Experiences with the Portability of Concurrent Pascal.

Software - Practice and Experience,

Vol 8 (1978), pp. 341-353.

[Nelson 79]

Nelson, P.A.

A Comparison of PASCAL Intermediate Languages.

Sigplan Notices, Vol 14 (1979), No 8 pp. 208-213.

[Nori et al. 76]

Nori K.V., Ammann U., Jensen K., Nageli H.H. and Jacobi Ch.

The PASCAL <P> Compiler Implementation notes.

Revised Edition

Institut fur Informatik, Eidgenossische Technische,

Hochschule, Zurich(1976).

[Parnas 78]

Parnas, D. L.

The Non-problem of Nested Monitor Calls.

Operating Systems Review, Vol 12 (1978), No. 1 pp. 12-14.

[Powell 79]

Powell, M. S.

Experience of Transporting and Using the

SOLO Operating System.

Software - Practice and Experience,

Vol 9 (1979), pp. 561-569.

[Prael 82]

Prael, C. E.

PASCAL for Operating Systems?

A critical Examination.

Sigplan Notices, Vol 17 (1982), No 3 pp. 53-57.

[Pyle 81]

Pyle, I. C.

The ADA Programming Language.

Prentice-Hall International, Inc. London(1981).

[Ravn 82]

Ravn, A. P.

Pointer Variables in Concurrent Pascal.

Software - Practice and Experience,

Vol 12 (1982), No. 3 pp. 211-222.

[Sale 79]

Sale, A. H. J.

Strings and sequence abstractions in Pascal.

Software - Practice and Experience,

Vol 9 (1979), No. 8 pp. 671-683.

[Schneider and Bruell 81]

Schneider, G. M. and Bruell, S. C.

Advanced Programming and Problem Solving

with Pascal

John Wiley & Sons Inc., New York(1981).

[Ullman 76]

Ullman, J. D.

Fundamental Concepts of Programming Systems.

Addison-Wesley, Reading, Massachusetts(1976).

[Welsh and McKeag 80]

Welsh, J. and McKeag, M.

Structured System Programming.

Prentice-Hall, International, Inc., London.

[Wirth 77]

Wirth, N.

Modula: a Language for Modula Multiprogramming.

Software - Practice and Experience, Vol 7 (1977), pp. 3-35.

Manuals

IBM Series/1

4955 Processor and Processor Features Description.

IBM Corporation, GA34-0021-3.

Modular One User Handbook.

1.11 Information Processor.

Computer Technology Limited, 311U Issue 1.

Modular One User Manual.

1.821 Assembly Language.

Computer Technology Limited, 69/113.