1-1-1984

Concurrent programming with a focus on concurrent pascal.

Barbara H. Smolowitz

CONCURRENT PROGRAMMING WITH A FOCUS ON CONCURRENT PASCAL

by

Barbara H. Smolowitz

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

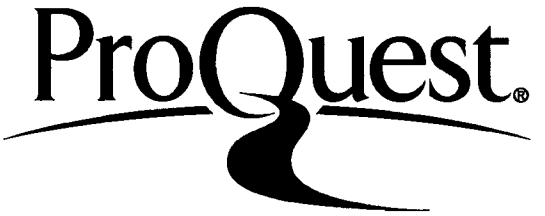
Master of Science

in

Computing Science

Lehigh University

1984



This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 10, 1984
(Date)

Professor in Charge

Head of Division

TABLE OF CONTENTS

PAGE

TABLE OF CONTENTS.....iii

LIST OF FIGURES.....iv

ABSTRACT.....1

I. INTRODUCTION.....3

II. BACKGROUND.....5

III. PROBLEMS OF CRITICAL REGION AND DEADLOCK.....11

IV. SEQUENTIAL PASCAL.....14

V. CONCURRENT PASCAL.....25

VI. OPERATING SYSTEMS AND CONCURRENT PASCAL.....51

VII. EXAMINATION OF THE SOLO OPERATING SYSTEM.....54

VIII. CONCLUSION.....71

REFERENCES.....72

APPENDIX.....73

VITA.....78

LIST OF FIGURES

FIGURE 1: Peterson's Algorithm.....27

FIGURE 2: Implementation of Peterson's Algorithm.....29

FIGURE 3: Algorithm For Critical Region Management Using Semaphores.....35

FIGURE 4: Implementation of Semaphores.....36

FIGURE 5: Implementation of a PROCESS Declaration....43

FIGURE 6: Implementation of a MONITOR Declaration....44

FIGURE 7: Implementation of a CLASS Declaration.....45

FIGURE 8: Implementation of an Initial Process.....48

ABSTRACT

CONCURRENT PROGRAMMING WITH A FOCUS ON CONCURRENT PASCAL

This thesis examines various aspects of concurrent programming. Concurrent programming is presently used to simulate concurrent processing on sequential hardware.

Concurrent processing is useful in several applications. It can be utilized to speed up computer operations and make man-machine interactions more efficient. It can also serve to more realistically model real-life situations.

Originally any concurrent programming was done at the machine or assembly language level. This programming is difficult to debug and modify. Structures added to high-level sequential languages improved the situation. Concurrent Pascal is a high-level concurrent language extended from sequential Pascal. It retains the structures of sequential Pascal while adding structures which manage the problems inherent in concurrent programming.

"Critical region" and "deadlock" are the major two problems in concurrent programming. Critical regions

are regions shared by two or more concurrent processes. Deadlock is a situation that occurs when two or more processes wait indefinitely to use a shared region.

Concurrent Pascal has been shown to be a very effective tool in the writing of operating systems for computers. An operating system is the software that manages the resources of the computer. By using Concurent Pascal, writing of an operating system is simplified.

The goals of Per Brinch Hansen, the developer of Concurrent Pascal, were simplicity, reliability, and adaptability. Simplicity results from having structures that manage the critical region and avoid deadlock through their design without the direct intervention of the programmer. Reliability is attained through comprehensive error checking by the compiler. Adaptability is achieved by using a hierarchical structure for programming in which program pieces can be studied individually. Modifications can then be made without fear of creating errors in other program pieces.

I. INTRODUCTION

Concurrent processing is a term used to describe the simultaneous processing of two or more tasks by a single computer. Actual concurrent processing can only be achieved with hardware. Concurrency, however, can be simulated with appropriate software. For this paper, the term "apparent concurrent processing" will be used when referring to simulated concurrency.

There are several levels at which some form of concurrent processing can be obtained. They range from portions of instructions within a program to whole jobs. An example of the former might be computation of a mathematical formula. In the calculation of the expression 3 * x + 7 * y / z - 5 * z, each of the terms are independent of each other and they can therefore be calculated simultaneously. The products would then be added together.

Similarly, within a program, procedures or other independent program segments can be executed simultaneously. An example of this might be a statistics package with various procedures designed to calculate different analyses on the same data.

The highest level of concurrent processing is in

the handling of whole jobs. A single computer can service several users at a given time with apparent concurrent processing as in a timesharing system.

The use of concurrent processing in operating systems will be discussed in greater depth in the following chapters. There are other areas in which concurrent processing is beneficial. Another example of the timesharing aspect of concurrent processing is in electronic mail. With electronic mail, many users send and receive mail simultaneously. There are other systems which are actually parallel processes but due to the sequential nature of computers, their simulation has been modeled sequentially. Concurrent programming allows for modeling in a more realistic fashion. Examples of this are manufacturing process control systems, train and subway scheduling, and weather forecasting.

This paper will include background information, problems involved in concurrent processing, a brief description of sequential Pascal, and the use of Concurrent Pascal for concurrent programming and specifically operating systems.

II. BACKGROUND

The various technological advances of the past forty years have led to the feasibility of concurrent programming that is discussed in this paper. In 1946, Dr. John von Neumann wrote "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" in which he proposed a novel method of programming computers. Prior to this time, "programming" computers was accomplished by hard-wiring the program into the machine. A change in program necessitated physical wiring changes. Von Neumann's idea was to store the program, in the form of numbers, along with the data. The first computer built to utilize this concept was the Electronic Delay Storage Automatic Calculator (EDSAC) built by Maurice V. Wilkes and his colleagues at Cambridge University, England in 1949. (Shelly, 1980)

Another major step in this area is described in the abstract of "PILOT - A New Multiple Computer System", written in 1959. "The PILOT data processor is a high-speed multiple computer system, more than 100 times faster than SEAC [designed in May, 1950, by the

National Bureau of Standards]. It contains three interconnected computers for rapid processing of data, and also contains multiple input-output channels for rapid transfer of data into and out of the system. All of these units operate concurrently in a coordinated fashion." (Leiner, 1959, p. 313) Each of the three computers mentioned had a specific purpose and the three were designed to run concurrently to provide the high rate of speed desired.

The first, or primary computer, was designed to handle the computations involved within the program. It had sixteen basic instructions: seven arithmetic operations, two logical processing operations, five choice operations, and two control operations. Only the two control operations, "transfer between storage units" and "regulate secondary computer", deal with program management.

Major program management was handled by the secondary comuter. This computer, independently programmed, performed procedures useful to the program executing on the primary computer. The secondary computer performed such tasks as counting iterations, sequencing the program running on the primary computer,

and manipulating the base registers in secondary storage. This computer also had sixteen basic instructions: six arithmetic operations, four choice operations, five control operations, and one logical processing operation. Working together, the primary and secondary computer were designed to handle complex sorting techniques as well as logarithmtic searches and error analyses.

The third computer was designed to independently handle the funcitons of editing, interpreting, and modifying data entering or leaving the system. It had eight basic instructions: three processing operations, three choice operations, and two control operations.

The control operations that each of the three computers could perform were the means by which they "communicated". The three computers were capable of independent execution and were programmed using the limited machine language instructions. This initial implementation of concurrency, therefore, was accomplished by a combination of hardware implemented interlocks and independently programmed computers.

In the early 1960's, the concept of running whole jobs concurrently was explored in the form of

and manipulating the base registers in secondary storage. This computer also had sixteen basic instructions: six arithmetic operations, four choice operations, five control operations, and one logical processing operation. Working together, the primary and secondary computer were designed to handle complex sorting techniques as well as logarithmtic searches and error analyses.

The third computer was designed to independently handle the funcitons of editing, interpreting, and modifying data entering or leaving the system. It had eight basic instructions: three processing operations, three choice operations, and two control operations.

The control operations that each of the three computers could perform were the means by which they "communicated". The three computers were capable of independent execution and were programmed using the limited machine language instructions. This initial implementation of concurrency, therefore, was accomplished by a combination of hardware implemented interlocks and independently programmed computers.

In the early 1960's, the concept of running whole jobs concurrently was explored in the form of

time-sharing. While the PILOT project examined using different parts of the hardware within the system to perform tasks of one job concurrently, the next step was better utilization of the system by interacting the computer with more than a single user at one time. Man-machine interaction is extremely slow in comparison with the computational speed of the computer. Better utilization of the equipment was the impetus behind the concept of timesharing.

As programs became more complex, debugging the programs became more time consuming. With batch processing, the delays between discovering a bug and trying a correction could become interminable. One solution would have been to allow the programmer dedicated access to the computer for debugging. However, this would have been wasteful of computer time. Several other problems were inherent in this solution.

In "An Experimental Time-Sharing System", the solution of having several users at terminals which interact with the computer is discussed.

> "To solve these interaction problems we would like to have a computer made simultaneously available to many users in a

manner somewhat like a telephone exchange. Each user would be able to use a console at his own pace and without concern for the activity of others using the system. This console could as a minimum be merely a typewriter but more ideally would contain an incrementally modifiable self-sustaining display. In any case, data transmission requirements should be such that it would be no major obstacle to have remote installation from the computer proper.

"The basic technique for a time-sharing system is to have many persons simultaneously using the computer through typewriter consoles with a time-sharing supervisor program sequentially running each user program in a short burst or quantum of computation. This sequence, which in the most straightforward case is a simple round-robin, should occur often enough so that each user program which is kept in the high-speed memory is run for a quantum at least once during each approximate human reaction time (~.2 seconds). In this way, each user sees a computer fully responsive to even single key strokes each of which may require only trivial computation; in the non-trivial cases, the user sees a gradual reduction of the response time which is proportional to the complexity of the response calculation, the slowness of the computer, and the total number of active users. It should be clear, however, that if there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer speed. During the period of high interaction rates while debugging programs, this should not be a hindrance since ordinarily the required amount of computation needed for each debugging computer response is small compared to the ultimate production need." (Corbato, 1962, pp. 335-336)

While several problems were noted by the authors, solutions were suggested. By the mid-1960's, operating systems were designed to handle this type of apparent concurrent processing.

At approximately the same time, assembly languages were developed which could handle the programming of concurrent tasks within a program, i.e. programmed multiprocessing. The incorporation of programmed multiprocessing gives sequential machines the ability to perform apparent concurrent processing. The programmed multiprocessing was handled through such commands as FORK and JOIN. The FORK command begins two or more parallel processes and those processes are ended and the single parent process continued at the JOIN. Melvin E. Conway wrote in his conclusion of "A Multiprocessor System Design" in 1963, that the effort should be made to incorporate such concurrent concepts in "common publication languages, for example, ALGOL." (p. 146)

While several problems were noted by the authors, solutions were suggested. By the mid-1960's, operating systems were designed to handle this type of apparent concurrent processing.

At approximately the same time, assembly languages were developed which could handle the programming of concurrent tasks within a program, i.e. programmed multiprocessing. The incorporation of programmed multiprocessing gives sequential machines the ability to perform apparent concurrent processing. The programmed multiprocessing was handled through such commands as FORK and JOIN. The FORK command begins two or more parallel processes and those processes are ended and the single parent process continued at the JOIN. Melvin E. Conway wrote in his conclusion of "A Multiprocessor System Design" in 1963, that the effort should be made to incorporate such concurrent concepts in "common publication languages, for example, ALGOL." (p. 146)

III. PROBLEMS OF CRITICAL REGION AND DEADLOCK

From the beginning it was recognized that there were two major problems in concurrent processing. The first is termed "deadlock". When concurrent processes use shared resources, there is the problem that two or more processes will wait to use the shared resource indefinitely, creating a deadlock. If both processes are equivalent and are given a part of the shared resource, neither process may have enough of the resource to complete its process, and therefore neither process can continue. Per Brinch Hansen defines "deadlock" as a "situation in which two or more processes are waiting indefinitely for events that will never occur." (1973, p. 336)

An example of this would be the problem of the banker with a fixed number of monetary units to loan to several customers. He wishes to satisfy the maximum number of customers whose individual requests do not exceed the fixed amount he has to lend. The customers may be given only part of the amount requested at any given time, but they will not repay the loan until they have received the entire amount requested. More

specifically, suppose the banker has 1000 monetary units to lend to customers Custa, Custb, Custc, and Custd. The requests are as follows:

Custa	375
Custb	582
Custc	260
Custd	386

The banker has several options. One option is for him to give Custa and Custb their full amounts and Custc and Custd a very small portion of their requests. Custc and Custd would then receive the remainder of their requests from the monetary units repaid by Custa and Custb. If, however, he gives each customer 250 monetary units, he will have a deadlock situation. He will be unable to lend anyone his full request and therefore none of the customers will repay his loan.

Another example of deadlock would be two processes (X and Y) that share two files A and B. Process X reads from file A and writes to file B, while process Y reads from file B and writes to file A. Process X will not give up file A until it has written to file B and process Y will not give up file B until it has written to file A. Initially process X is given file A and process Y is given file B. A deadlock occurs because neither process can terminate. Process X waits

indefinitely for B, and process Y waits indefinitely for A.

The second problem that arises is with critical regions. Critical regions are regions within a system (or program) which are shared by two or more processes but should be accessed by only one process at a time. Examples might be input/output devices or variables common to at least two concurrent processes. Hansen suggests three criteria for critical regions as follows:

(1) No more than one process can be allowed access to the critical region at any given time.

(2) Any process which has access to the critical region must finish execution within and exit the critical region within a finite amount of time.

(3) Any process that requests access to the critical region may not be blocked from the critical region indefinitely. (Brinch Hansen, 1973)

There are various methods for managing these problems in concurrent programming. They will be discussed further in the chapter on Concurrent Pascal.

IV. SEQUENTIAL PASCAL

"Pascal was introduced in 1971 by Professor Niklaus Wirth. His aim was to make available a language which would allow programming to be taught as a systematic discipline and in which the techniques of both 'scientific' and 'commercial' programming could be convincingly demonstrated. The adoption of Pascal has been rapid and widespread, to the extent that it has become the 'lingua franca' of computing science." (Findlay, 1981, p. iii)

In his own words, N. Wirth explained his justification for introducing a new language as follows:

> "The development of the language Pascal is based on two principal aims. The first is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language. The second is to develop implementations of this language which are both reliable and efficient on presently available computers.
>
> "The desire for a new language for the purpose of teaching programming is due to my dissatisfaction with the presently used major languages whose features and constructs too often cannot be explained logically and convincingly and which too often defy systematic reasoning. Along with this dissatisfaction goes my conviction that the language in which the student is taught to express his ideas profoundly influences his

> habits of thought and invention, and that the disorder governing these languages directly imposes itself onto the programming style of the students." (Jensen, 1978, p. 133)

It was, perhaps, Wirth's desire for an orderly high-level language that led to the highly structured nature of Pascal. The language is divided into data structures and instructions for how the data structures are to be manipulated. The following is a brief description of the structures in sequential Pascal. Concurrent Pascal (a description of which is found in the next chapter) is built upon these structures.

All of the data used within the program must be represented as variables. These variables must be declared as previously defined types. It is in these type declarations that a great deal of Pascal's versatility is evidenced. Once a variable is declared of a given type, it may not be given a value of another type.

There are four standard types which are predefined. These are INTEGER, REAL, BOOLEAN, and CHAR. A variable of type INTEGER may have any integer value. Arithmetic operators which would result in integer values when used with integer operands would be +, -, *, DIV, and MOD. Similarly, variables of type

REAL may have any real number value. The arithmetic operators, which result in real values when the operands are real or integer values, are +, -, *, /. There are constraints placed on maximum and minimum values by the hardware on which the software is implemented.

The data type BOOLEAN has only two values: TRUE and FALSE. These operands use the logical operators AND, OR, and NOT. The standard boolean results are obtained using these operators. TRUE and FALSE are predefined such that the value of FALSE is less than TRUE.

The last standard data type, CHAR, allows the variables declared as CHAR to have the values of a predefined set of characters that is finite and ordered. While this set is not standard, it includes the alphanumeric characters; 'A'..'Z' and '0'..'9'; the blank character; and usually various other characters such as '*', '.', '%', and '@'. The value of the characters is again implementation dependent.

Pascal also allows the user to define new types. These may be simple scalar types, subrange types, or complex structured types.

A scalar type is an ordered list of identifiers.

Once this list is declared, the identifiers become the constant values of that type. An example of this type might be the declaration for days as follows:

DAYS = (SUN, MON, TUES, WED, THURS, FRI, SAT)

A subrange type, as the name suggests, consists of the subrange of a previously declared type (with the exception of the REAL type). Two examples of this, the first a subrange of INTEGER and the second a subrange of DAYS defined above, follow.

TEMPS = 32..212

WEEKDAY = MON..FRI

The relational operators =, <, <=, >, >=, and <> apply to all of the simple data types. There are also various built-in functions, such as ORD, TRUNC, and ODD which have a value of one type as an argument and return a value of another type.

Complex structured data types consist of various simple data types (standard, user defined scalar, and user defined subrange) and a combination of one or more of four structure components. These structure components are ARRAY, RECORD, SET, and FILE.

An ARRAY consists of a collection of components of the same type. The ARRAY may be single or

multi-dimensional. A list of names or the positions on a checkerboard are examples of structures that could be represented by the ARRAY type. The ARRAY type is defined giving the ranges of the dimensions and the type of values of the components. The above examples might be declared as follows.

```
NAME = ARRAY [1..15] OF CHAR;
LISTOFNAMES = ARRAY [1..30] OF NAME
```

or

```
BOARD = ARRAY [0..7,0..7] OF BOOLEAN
```

A SET is another of the structured data types. Like an ARRAY, a SET is a collection of values of the same type. SETs differ from ARRAYs in that a variable of this type represents a subset of the powerset of the base type. The following is an example of a SET declaration:

```
SUITS = [CLUBS, DIAMONDS, HEARTS, SPADES]
```

Note that there is no order within the SET and the empty set is represented by []. The operators +, *, and - represent the set operations union, intersection, and set difference respectively. There are also relational operators for SETs. = and <> test for set equality and inequality; <= and >= test for inclusion; and IN tests for set membership.

The third structured data type is RECORD. RECORDS are the most flexible of the Pascal data types. A RECORD is a collection of components, but unlike an ARRAY, the components need not be the same. The components are called fields of the RECORD. A single RECORD may, for example, contain an ARRAY field, an INTEGER field, and a REAL field. In this example, the declaration might be as follows:

```
PERSON = RECORD
            NAME: ARRAY [1..15] OF CHAR;
            AGE:  INTEGER;
            PAY:  REAL
         END
```

In this example, NAME, AGE, and PAY are the field identifiers.

RECORDS can also be defined as variant RECORDs by using the CASE statement. This allows a given field to have different structures depending on the value of a given component (the tag field). An example of a variant RECORD declaration follows.

```
DATE = ARRAY [1..9] of CHAR;
AUTO = RECORD
          MAKE: (GM, CHRYS, FORD, AM, FOREIGN);
          YEAR: 1900..2000;
          CASE PASSEDINSPECT: BOOLEAN OF
             TRUE: (STICKERNO: INTEGER);
             FALSE: (LIGHTSFAIL: BOOLEAN;
                     BRAKESFAIL: BOOLEAN;
                     EMITSFAIL: BOOLEAN;
                     EXPDATE: DATE);
       END
```

The last of the structured data types is the type FILE. A FILE is a sequence of components which are the same. Again, a FILE is similar to an ARRAY, but there are two major differences. The length of a FILE is not fixed as it is in an ARRAY and components of a FILE can only be accessed by progressing through the FILE from the beginning. An empty FILE is a FILE with no components. Components are written to or read from FILEs. There are four operators for FILE variables. RESET returns to the beginning of the FILE for the purpose of reading from the FILE. REWRITE, likewise, returns to the beginning of the FILE for the purpose of writing to the FILE. The GET operator "gets" the next component (if it exists) from the FILE and puts it in a buffer variable, and the PUT operator "puts" the next component into the file. EOF is a built-in BOOLEAN function that becomes TRUE when the last component in the FILE is read. The procedure READ (or WRITE) is composed of an assignment and a GET (or PUT).

One type of FILE is the text FILE or FILE OF CHAR. For this type of FILE, two special procedures READLN and WRITELN are defined in terms of GET and PUT respectively. A built-in function EOLN is defined to

be TRUE only when an end-of-line marker has been reached.

The preceeding data types are all static data types. Pascal also has a dynamic data type called a POINTER (↑). With a variable of a static data type, space is allotted in memory for the value of the variable. This space is reserved during the entire execution. Space for a variable of a dynamic data type is allocated and destroyed during execution with the use of NEW and DISPOSE. POINTERs refer to the location of a value rather than actually being the location of the value.

A linked list is one example of the use of POINTERs. In a linked list each component is "linked" by a POINTER to the next component. An example of a declaration for such a linked list is:

```
NAMEPOINTER = ↑ NAMENODE;
NAMENODE = RECORD
             NAME: ARRAY [1..15] OF CHAR;
             NEXT: NAMEPOINTER
           END
```

There are four types of instructions used to manipulate the data: assignment, compound, repetitive, and conditional. The assignment statement is used to give a variable a value. It is of the form <variable> := <expression>. The second type is the compound

statement. This consists of other statements with the delimiters BEGIN and END. The statements between the BEGIN and END may be of any type and there may be any number of them.

The three types of repetitve statements are the FOR loop, the WHILE loop, and the REPEAT loop. The FOR loop performs the statements within the loop a predetermined number of times. It uses a control variable to count the iterations. The FOR loop is of the form:

FOR <variable> := <expression> TO|DOWNTO
<expression> DO <statement>

The WHILE loop performs the statement within the loop as long as a given condition is TRUE. The test for the condition appears at the beginning of the loop. The WHILE loop has the following form:

WHILE <expression> DO <statement>

The REPEAT loop is similar to the WHILE loop except that

(1) the test is performed at the end of the loop which results in the statements within the loop being executed at least once,

(2) the statements within the loop are performed until the given condition becomes TRUE, and

(3) any number of statements may be within the loop.

The REPEAT loop is of the form:

REPEAT <statement> {; <statement>} UNTIL <expression>

The last type of instructions is the conditional instruction. There are two of this type, the IF and the CASE statements. With the IF statement, of the form:

IF <expression> THEN <statement>| IF <expression> THEN <statement> ELSE <statement>

The statement following the THEN is executed only if the expression is true. If it is FALSE, and there is an ELSE, then the statement following the ELSE is executed. There is an ambiguity here which results from a statement of the form:

IF <expression> THEN IF <expression> THEN <statement> ELSE <statement>

The ambiguity is resolved by the convention that in such a case, the ELSE statement goes with the closest IF that is not already terminated (by a semicolon or closer ELSE).

The CASE statement is designed for situations which would otherwise necessitate the use of several nested IF statements. The CASE statement is of the

form:

```
CASE <expression> OF
  <case label> {, <case label>} : <statement>
  {; <case label> {, <case label>} : <statement>}
END
```

The statement associated with a particular case label is executed when the case label is the value of the expression.

The WITH statement, of the following form,

```
WITH <record variable> {, <record variable>} DO
<statement>
```

allows fields of a record to be denoted by their field identifier only. Pascal also allows the user to define PROCEDUREs and FUNCTIONs. With the exception of their headings, these have a form similar to the program itself and are used as subroutines of the program. There is also a GOTO statement in Pascal which can be used to jump the execution to another part of the program.

For a more in depth explanation of sequential Pascal, the reader is referred to PASCAL User Manual and Report (Jensen, 1978).

V. CONCURRENT PASCAL

Concurrent programming can be achieved with an extended Pascal by the addition of structures that perform the tasks of FORK and JOIN mentioned previously. COBEGIN, which has the effect of beginning two or more concurrent processes, is simulated by interleaving the statements of the concurrent processes. COEND delays continuation of the main process until all of the concurrent processes have terminated.

It is the responsibility of the programmer to handle the problems of the critical region and deadlock. There are two main methods for managing critical regions and avoiding deadlock. The first is "busy waiting" and the second is with "semaphores".

With "busy waiting", any process needing the critical region enters an indefinite loop just before entering the critical region. It exits the loop when it meets the condition that (a) it is the only process requesting access to the critical region which is currently free or (b) it is the process' turn for the critical region and the critical region is currently free.

Peterson's algorithm, shown in Figure 1, (Peterson, 1981) is an example of this type of management of critical regions. This algorithm protects a critical region while having a "fair" system of accessing the region. Each of the concurrent processes will eventually be given access to the critical region and at no time will more than one process be given access to the critical region. This is accomplished by establishing several conditions for entering the critical region. For a process to enter the critical region, the value of its conditional expression in the REPEAT loop preceeding the critical region must be TRUE. Only one process will have a set of individual conditions with a pattern of values such that the entire expression is TRUE.

THE SOLUTION FOR TWO PROCESSES.

```
(* trying protocol for P1 *)
Q1 := TRUE;
TURN := 1;
wait until NOT Q2 OR TURN = 2;
Critical Section;
(* exit protocol for P1 *)
Q1 := FALSE.

(* trying protocol for P2 *)
Q2 := TRUE;
TURN := 2;
wait until NOT Q1 OR TURN = 1;
Critical Section;
(* exit protocol for P2 *)
Q2 := FALSE
```

FIGURE 1: Peterson's Algorithm

THE SOLUTION FOR n PROCESSES.

```
(* protocols for P1 *)
FOR j := 1 TO n - 1 DO
BEGIN
  Q[i] := j;
  TURN[j] := i;
  wait until ([for all] k ≠ i, Q[k] < j) OR
TURN [j] ≠ i
END;
Critical Section;
Q[i] := 0
```

FIGURE 1 (continued)

```
(* DECLARATIONS FOR THE PROGRAM SEGMENTS *)

TYPE
  KINDTRANSACT = (CR, DEB);
  TRANSACT = RECORD
               KIND: KINDTRANSACT;
               AMOUNT: REAL;
               BRANCHNUM: INTEGER;
             END;
  ACCT = RECORD
           NAME: ARRAY [1..25] OF CHAR;
           SSNUM: INTEGER;
           NUMTRANSACTIONS: INTEGER;
           TRANSACTIONS: ARRAY [1..MAXNUM] OF
TRANSACT;
           BALANCE: REAL
         END;
  ACCOUNTS = ARRAY [1..NUMACCTS] OF ACCT;

VAR
  ACCTS: ACCOUNTS;
  B1, B2: BOOLEAN;
  TURN: INTEGER;
```

FIGURE 2: Implementation of Peterson's Algorithm

```
PROCEDURE RECORD_TRANSACTION (NUM: INTEGER; AMT:
REAL; K: KINDTRANSACT; BRNUM: INTEGER);

BEGIN
  WITH ACCTS [NUM] DO
    BEGIN
      NUMTRANSACTIONS := NUMTRANSACTIONS + 1;
      WITH TRANSACTIONS [NUMTRANSACTIONS] DO
        BEGIN
          KIND := K;
          AMOUNT := AMT;
          BRANCHNUM := BRNUM
        END;
      BALANCE := BALANCE + AMT;
    END
END;
```

FIGURE 2 (continued)

```
PROCEDURE BRANCH1;

VAR
  ACCTNUM: INTEGER;
  AMNT: REAL;
  KND: KINDTRANSACT;

BEGIN
  REPEAT
(* THE GETINFO PROCEDURE GETS THE INFORMATION
NEEDED FOR RECORDING THE DEBITS AND CREDITS.  FOR
THIS EXAMPLE WE NEED NOT BE CONCERNED WITH THE
DEFINITION OF THIS PROCEDURE. *)
    GETINFO (ACCTNUM, AMNT);
    IF AMNT > 0 THEN
      KND := CR
    ELSE
      KND := DEB;
    B1 := TRUE;
    TURN := 1;
    REPEAT UNTIL ((NOT B2) OR (TURN = 2));
    RECORD_TRANSACTION (ACCTNUM, AMNT, KND, 1);
    B1 := FALSE
  UNTIL FALSE;
END;
```

FIGURE 2 (continued)

```
PROCEDURE BRANCH1;

VAR
  ACCTNUM: INTEGER;
  AMNT: REAL;
  KND: KINDTRANSACT;

BEGIN
  REPEAT
(* THE GETINFO PROCEDURE GETS THE INFORMATION
NEEDED FOR RECORDING THE DEBITS AND CREDITS.  FOR
THIS EXAMPLE WE NEED NOT BE CONCERNED WITH THE
DEFINITION OF THIS PROCEDURE. *)
    GETINFO (ACCTNUM, AMNT);
    IF AMNT > 0 THEN
      KND := CR
    ELSE
      KND := DEB;
    B1 := TRUE;
    TURN := 1;
    REPEAT UNTIL ((NOT B2) OR (TURN = 2));
    RECORD_TRANSACTION (ACCTNUM, AMNT, KND, 1);
    B1 := FALSE
  UNTIL FALSE;
END;
```

FIGURE 2 (continued)

```
PROCEDURE BRANCH2;

VAR
  ACCTNUM: INTEGER;
  AMNT: REAL;
  KND: KINDTRANSACT;

BEGIN
  REPEAT
    GETINFO (ACCTNUM, AMNT);
    IF AMNT > 0 THEN
      KND := CR
    ELSE
      KND := DEB;
    B2 := TRUE;
    TURN := 2;
    REPEAT UNTIL ((NOT B1) OR (TURN = 1));
    RECORD_TRANSACTION (ACCTNUM, AMNT, KND, 2);
    B2 := FALSE;
  UNTIL FALSE;
END;


BEGIN (* MAIN *)
  B1 := FALSE;
  B2 := FALSE;
  TURN := 1;
  COBEGIN
    BRANCH1;
    BRANCH2;
  COEND;
END. (* MAIN *)
```

FIGURE 2 (continued)

Figure 2 is a set of program segments showing an implementation of Peterson's algorithm for two concurrent processes. In the hypothetical situation, a bank has two branches which concurrently record debits and credits. The critical region is the RECORD_TRANSACTION procedure. In this example, the critical region is managed through the variables TURN, B1, and B2.

While "busy waiting" manages the critical region, it is wasteful of CPU power. The waste arises in the constant checking in the REPEAT UNTIL ((NOT B2) or (TURN = 2)) and REPEAT UNTIL ((NOT B1) or (TURN = 1)) statements. This method of management is also cumbersome. For several concurrent processes the implementation of the algorithm becomes quite complicated.

In 1965, E. W. Dijkstra proposed using semaphores to simplify the management of critical regions. The additional structures WAIT and SIGNAL are used with the new data type, SEMAPHORE. A SEMAPHORE is an variable of type INTEGER. It is only operated upon by WAIT and SIGNAL. WAIT and SIGNAL are defined as follows: (Ben-Ari, 1982)

WAIT (s): If s > 0 then s := s - 1 else the

execution of the process that called WAIT (s) is suspended.

SIGNAL (s): If some process P has been suspended by a previous WAIT (s) on this SEMAPHORE s then wake up P else s := s + 1.

The critical region is then managed by the algorithm given in Figure 3 for n processes. It is possible for "lockout" to occur using this algorithm unless a "fair" method is designed for determining which process is woken by SIGNAL. Figure 4 shows the program segment in Figure 2 rewritten using SEMAPHOREs.

```
VAR
  S: SEMAPHORE;

PROCEDURE Pi;
BEGIN
  REPEAT
    WAIT (S);
    Critical Region (Pi);
    SIGNAL (S);
    Remote Region (Pi);
  UNTIL FALSE;
END;

BEGIN (* MAIN *)
  S := 1;
  COBEGIN
    P1;
    P2;
      .
      .
      .
    Pn
  COEND
END. (* MAIN *)
```

FIGURE 3: Algorithm For Critical Region Management Using Semaphores

```
(* SEE FIGURE 2 FOR THE TYPE DECLARATIONS AND
DECLARATION OF RECORD_TRANSACTION PROCEDURE *)

VAR
  ACCTS: ACCOUNTS;
  S: SEMAPHORE;

PROCEDURE BRANCH1;
BEGIN
  REPEAT
    GETINFO (ACCT, AMNT); (* REFER TO FIGURE 2 FOR
COMMENT ON GETINFO *)
    IF AMNT > 0 THEN
       KND := CR
    ELSE
       KND := DEB;
    WAIT (S)
    RECORD_TRANSACTION (ACCTNUM, AMNT, KND, 1);
    SIGNAL (S)
  UNTIL FALSE;
END;

PROCEDURE BRANCH2
BEGIN
  REPEAT
    GETINFO (ACCTNUM, AMNT);
    IF AMNT > 0 THEN
      KND := CR
    ELSE
      KND := DEB;
    WAIT (S);
    RECORD_TRANSACTION (ACCTNUM, AMNT, KND, 1);
    SIGNAL (S);
  UNTIL FALSE;
END;
```

FIGURE 4: Implementation of Semaphores

```
BEGIN (* MAIN *)
  S := 1;
  COBEGIN
    BRANCH1;
    BRANCH2
  COEND
END. (* MAIN *)
```

FIGURE 4 (continued)

While on the surface the problem of critical region managment appears to be solved by the use of SEMAPHOREs in a relatively straight-forward manner, Brinch Hansen (1973) points out the flaws in this reasoning.

> "If we replace this structured notation [shared regions] with semaphores, this will have grave consequences:
>
> (1) Since a semaphore can be used to solve arbitrary synchronizing problems, a compiler cannot conclude that a pair of wait and signal operations on a given semaphore initialized to one delimits a critical region, nor that a missing member of such a pair is an error. A compiler will also be unaware of the correspondence between a semaphore and the common variable it protects. In short, a compiler cannot give the programmer any assistance whatsoever in establishing critical regions correctly.
>
> (2) Since a compiler is unable to recognize critical regions, it cannot make the distinction between critical regions and disjoint processes. Consequently, it must permit the use of common variables everywhere. So a compiler can no longer give the programmer any assistance in avoiding time-dependent errors in supposedly disjoint processes."

The deadlock problem has been only partially solved. Deadlock can occur through poor management of the critical region, but it can also occur when any one (or more) of the following conditions exist.

"(1) Mutual exclusion: A resource can only be acquired by one process at a time.
(2) Non-preemptive scheduling: A resource can only be released by the process which has acquired it.
(3) Partial allocation: A process can acquire its resources piecemeal.
(4) Circular waiting: The previous conditions permit concurrent processes to acquire part of their resources and enter a state in which they wait indefinitely to acquire each other's resources."(Brinch Hansen, 1973)

Brinch Hansen (1977) outlines a hierarchical resource system to prevent deadlock. A hierarchical system consists of a sequential ordering for requesting and releasing resources. When concurrent programs are written using hierarchical ordering for system components, other benefits are realized. The major additional benefit is in program testing and correctness. Once a program component has been shown to be correct, errors in newer components cannot make older components fail because old components do not call newer components.

Brinch Hansen developed Concurrent Pascal (from 1972 - 1975) with the goal of creating a language for concurrent programs that satisfies three requirements: simplicity, reliability, and adaptability. Simplicity is achieved through the use of small, well-defined program pieces. Reliability is aided by extensive

compilation checks of type compatibility. Hierarchical structure also aids correctness testing. Adaptability comes in being able to modify existing programs. By using abstract language and small well-defined program components, modifications become easier.

Concurrent Pascal is an extension of sequential Pascal. The following is a brief description of the extended data structures and manipulation instructions in Concurrent Pascal. This information is taken from The Architecture of Concurrent Programs (Brinch Hansen, 1977).

Concurrent Pascal contains all of the data types of Pascal plus two additional data types, QUEUE and system. The majority of the manipulation instructions are the same, i.e. assignment, compound, FOR, WHILE, REPEAT, IF, CASE, and WITH. There are, however, also CYCLE statements and INIT statements in Concurrent Pascal. Concurrent Pascal also has procedure and function capabilities, but these differ slightly from sequential Pascal.

The two new data types, QUEUE and system, are called active types. Any type containing system types or QUEUEs is an active type. The remainder are passive types. QUEUE is a simple data type like CHAR, INTEGER,

BOOLEAN, REAL, subrange, and scalar types. System types are structured and consist of other component types.

There are three kinds of system types: PROCESSes, MONITORs, and CLASSes. A concurrent program is made up of these three types. A system type declaration is of the the following form:

PROCESS | MONITOR | CLASS <empty> | <parameters>; <block>

A PROCESS type consists of a data structure and a sequential statement for manipulation of that structure. Within the parameter list, the MONITORs to which the PROCESS has access are declared. A PROCESS has access only to MONITORs or CLASSes. PROCESSes do not have direct access to shared data. They must access the shared data through a MONITOR.

MONITORs consist of data structures and operations that PROCESSes can perform on these data structures. the operations are in the form of functions or procedures which the PROCESSes call. These operations manage the synchronization of the calling PROCESSes and the exchange of data among them.

A CLASS is a system component that can only be accessed by a single other system component (PROCESS,

MONITOR, or another CLASS). It consists of a data structure and operations that can be performed on the data structure (similar to a MONITOR).

Examples of PROCESS, MONITOR, and CLASS declarations are shown in Figures 5, 6, and 7 respectively. The problem of hypothetical bank with its concurrent branch recording processes is continued. For passive type declarations, see Figure 2.

```
TYPE BRANCHPROCESS = PROCESS (MANAGER:
RECORDMANAGER);

VAR
  ACCTNUM: INTEGER;
  AMNT: REAL;
  BRANCHNO: INTEGER;
  KND: KINDTRANSACT;

BEGIN
  CYCLE
    (* SEE COMMENT IN FIGURE 2 CONCERNING GETINFO.
THIS PROCEDURE REQUIRES AN ADDITIONAL PARAMETER:
BRANCHNO *)
    GETINFO (ACCTNUM, AMNT, BRANCHNO);
    IF AMNT > 0 THEN
      KND := CR
    ELSE
      KND := DEB;
    MANAGER. SEND (ACCTNUM, AMNT, KND, BRANCHNO);
  END;
END;
```

FIGURE 5: Implementation of a PROCESS Declaration

```
TYPE SENDERQUEUE = ARRAY [1..2] OF QUEUE;

TYPE RECORDMANAGER = MONITOR;

VAR
  SENDING: (ONE, TWO);
  SENDER: SENDERQUEUE;
  RECORDER: RECORD_TRANSACTION;

PROCEDURE ENTRY SEND (ACCTNUM: INTEGER; AMNT:
REAL; KND: KINDTRANSACT; BRANCHNO: INTEGER);

BEGIN
  IF BRANCHNO = 1 THEN
    BEGIN
      IF SENDING = TWO THEN DELAY (SENDER [1]);
      RECORDER.ENTER (ACCTNUM, AMNT, KND,
BRANCHNO);
      SENDING := ONE;
      CONTINUE (SENDER [2]);
    END
  ELSE
    BEGIN
      IF SENDING = ONE THEN DELAY (SENDER [2]);
      RECORDER.ENTER (ACCTNUM, AMNT, KND,
BRANCHNO);
      SENDING := TWO;
      CONTINUE (SENDER [1]);
    END;
END;

BEGIN
  SENDING := ONE;
  INIT RECORDER;
END;
```

FIGURE 6: Implementation of a MONITOR Declaration

```
TYPE RECORD_TRANSACTION = CLASS;

VAR
  ACCTS: ACCOUNTS;

PROCEDURE ENTRY (NUM: INTEGER; AMT: REAL; K:
KINDTRANSACT; BRNUM: INTEGER);

BEGIN
  WITH ACCTS [NUM] DO
    BEGIN
      NUMTRANSACTIONS := NUMTRANSACTIONS + 1;
      WITH TRANSACTIONS [NUMTRANSACTIONS] DO
        BEGIN
          KIND := K;
          AMOUNT := AMT;
          BRANCHNUM := BRNUM;
        END;
      BALANCE := BALANCE + AMT;
    END;
END;

BEGIN
(* INITIALIZE ACCTS *)
END;
```

FIGURE 7: Implementation of a CLASS Declaration

The QUEUE type is a standard type in Concurrent Pascal. It is declared within a MONITOR type and is used to delay and resume PROCESSes. There is a standard function EMPTY which has a QUEUE variable as its argument and results in a BOOLEAN value. The value is TRUE when there is no PROCESS delayed in the QUEUE. There are also two procedures defined for QUEUEs. DELAY results in the calling PROCESS losing its exclusive access to the MONITOR. Other PROCESSes can then call the MONITOR variables. CONTINUE is called by the PROCESSes returning from the MONITOR. If another PROCESS is waiting in the QUEUE, it immediately regains its exclusive access to the MONITOR variables.

As mentioned previously, there are two statements in Concurrent Pascal which are not in sequential Pascal. The first is the CYCLE statement. This statement is equivalent to:

REPEAT <statement> {; <statement>} UNTIL FALSE

It has the syntax:

CYCLE <statement> {; <statement>} END

The CYCLE statement may only be used in a PROCESS.

The INIT statement is used to initialize system components. The initial PROCESS, the outermost level

of the program, contains an INIT statement which initializes the other PROCESSes and MONITORs and defines their access rights to one another through their parameters. The INIT statement also allocates space for the system components variables. Once a system component is initialzed, its variables and parameters become permanent variables.

Routines, in Concurrent Pascal, are procedures, functions, and sequential programs. They consist of a set of parameters and a compound statement that operates on the parameters. While a system component may not refer to the variables of another system component, it may call routine entries defined within another system type. There are four types of routine entries: process entry, monitor entry, class entry, and initial statement. The last of these has been discussed previously. The initial statement does not have an identifier and is simply called using the INIT statement. Figure 8 shows an
initial process for the types declared in Figures 5-7.

```
VAR
  MANAGER: RECORDMANAGER;
  BRANCH1, BRANCH2: BRANCHPROCESS;

BEGIN
  INIT
    MANAGER,
    BRANCH1 (MANAGER),
    BRANCH2 (MANAGER);
END.
```

FIGURE 8: Implementation of an Initial Process

The other three kinds of routine entries appear in system components bearing their name. A process entry, defined within a process type, can only be called by a sequential program within a process type. It cannot be called by a system component. A monitor entry, on the other hand, can be called by any system component that wishes to operate on that monitor. Calls made simultaneously for monitor routines which operate on the same permanent variables will be handled singlely. A class entry can only be called by one system component, the system component that has access to that CLASS.

The syntax for the procedure and function routines are as follows:

PROCEDURE ENTRY | <empty> <identifier> <parameters>; <block>

FUNCTION ENTRY | <empty> <identifier> <parameters>: <identifier>; <block>

A sequential program routine is controlled by a job PROCESS. The parameters of the program must be of passive types and the rightmost parameter represents the variable in which the compiled program code is stored. The program may call other routines defined within the job PROCESS as long as these are lised following ENTRY in the program definition. The syntax

for a sequential program routine is as follows:

PROGRAM <identifier> <parameters>
<access rights> | <empty>

where <access rights> has the following syntax:

; ENTRY <identifiers>

The use of the MONITOR, PROCESS, and CLASS, as defined in Concurrent Pascal, removes the necessity for the programmer to manage the problems of critical region and deadlock. This management is built into the interaction of these data structures. The limited accessing among the data structures and their "one-way" nature also allows for greater compiler checking. This aids in ensuring program correctness.

This characteristic of Concurrent Pascal facilitates the writing of operating systems as will be discussed in the next two chapters.

VI. OPERATING SYSTEMS AND CONCURRENT PASCAL

An operating system is a software system designed to manage the sharing of computer resources. As mentioned previously, the sharing of resources can be by several users as in a time-sharing system. An operating system is also necessary for a single user to efficiently use a computer system. The problem of managing a system for several users is, therefore, an extension of the problem of managing the system for a single user.

A great deal of efficiency can be gained for a single user system by running computer processes concurrently. Ben-Ari (1982) gives the example of a computer that can execute one million instructions per second. This computer is connected to a card reader which reads 300 cards per minute. While one card is read (1/5 of a second), 200,000 instructions could be executed. A large percentage of the time the CPU will be idle if the card reading process and CPU execution take place sequentially.

In the 1960's autonomous peripheral devices were designed which could operate independent of the CPU. This meant that a computer could execute one program

while reading in a second program and possibly print out a third program. The problem arose, though, of synchronizing the CPU and the peripheral devices.

One method devised to handle the synchronization problem was the interrupt concept. With this method, a peripheral device sends a signal to a register connected to the CPU. When the signal is received, the CPU stops executing the current program and can then switch to a program that is waiting for the peripheral device. The program managing the action between the peripheral device and the CPU is the operating system.

The same concept used to permit concurrent operation of the peripheral devices and the CPU could be used to manage a system with several users.

Most operating systems are written in low-level languages. These programs are large and unwieldly. Several problems arise with these systems. Because of their size these programs are difficult to understand and modify. They are also prone to time-dependent errors. This makes the system unreliable and prone to crashing. Once an error has occurred, it is difficult to locate the problem.

Concurrent Pascal is an effective tool for writing operating system programs. Its structure is such that

shared resources are managed by independent components. It also allows for systematic testing of the system through hierarchical design. The Solo Operating System, which will be examined in detail in the next chapter, was written in Concurrent Pascal. Its author, Per Brinch Hansen, reported that it took approximately two man-years to develop the entire system. He estimates that it would have taken twenty to thirty man-years to develop the same system in machine language. (Brinch Hansen, 1977).

VII. EXAMINATION OF THE SOLO OPERATING SYSTEM

This chapter examines the Solo Operating System written by Per Brinch Hansen (1977). The purpose of this analysis is to show how the system was constructed using the concurrent structures of Concurrent Pascal. This examination will also show how the system was developed using a hierarchical structure.

The Solo Operating System was the first operating system written in Concurrent Pascal. It was implemented on the PDP 11/45 computer and was in use in May, 1975. It is unusual in that it is written almost entirely in Concurrent Pascal with only a small percentage of machine language code. Protection of the system is achieved through extensive compile-time checks of type compatability and access rights instead of execution-time checking with hardware mechanisms.

The operating system manages the processing of programs, written in sequential or Concurrent Pascal, for a single user. The user is able to edit, compile, and store these programs. The user intereacts with the computer through the use of a console. Through the console, the user can access a card reader, tape and disk devices, and a printer. The handling of these

functions is managed through concurrent processes in the operating system.

The main body of the operating system program is the INITIAL PROCESS (Brinch Hansen, 1977, pp. 140 - 141). This process, when executed, initializes six PROCESSes of five PROCESS types and fourteen MONITORs of seven MONITOR types. This INITIAL PROCESS has access only to those PROCESSes and MONITORs. Once it terminates execution, these structures remain as permanent variables. It is this INITIAL PROCESS that begins all of the concurrent processes necessary for the operating system.

The Appendix shows the hierarchical structure of the remainder of the program. If the program is considered in terms of "bottom up" design, the highest layer (that layer which no other components access) consists of the five other PROCESSes. These PROCESSes then have access to various MONITORs and CLASSes, as shown, which are either declared as parameters or variables within the PROCESS declaration. That layer of MOITORs and CLASSes then have access to MONITORs and CLASSes in a similar manner, and so on. The lowest layer of active types are those MONITORs and CLASSes that do not declare any other active types as

parameters or variables. They therefore do not have access to any other active types.

There are six CLASSes and MONITORs that do not access any other CLASSes or MONIORs. These are FIFO CLASS, TYPEWRITER CLASS, LINEBUFFER MONITOR, PAGEBUFFER MONITOR, ARGBUFFER MONITOR, and PROGSTACK MONITOR.

The FIFO CLASS (Brinch Hansen, 1977, p. 103) is used to manage a fifo (first in, first out) QUEUE. It consists of four ENTRY functions: ARRIVAL, DEPARTURE, EMPTY, and FULL. It is through these functions that this CLASS is accessed. The functions ARRIVAL and DEPARTURE are INTEGER functions and return the values at which the next QUEUE element can take or leave from respectively. The functions EMPTY and FULL return BOOLEAN values depending on the value of the INTEGER variable length. A value of 0 for length would return a value of TRUE for EMPTY and a value of limit (a parameter value for the size of the QUEUE) would return a value of TRUE for FULL. A variable of type FIFO CLASS is initialized with the head and tail variables having a value of 1 and a length of 0.

The TYPEWRITER CLASS (Brinch Hansen, 1977, pp. 107 - 108) is used to transfer a line of text to or from the console. An IO procedure is used to delay the

calling process while a single character is transferred. This type consists of two ENTRY procedures WRITE and READ. The WRITE procedure consists mainly of a REPEAT loop that calls a WRITECHAR procedure until an entire line has been written to the console (using the IO procedure). The READ procedure begins by ringing the bell on the console. The remainder of the procedure is consists mainly of a REPEAT loop. In the REPEAT loop, a single character is read from the console until an entire line is read. The end of line is determined by a linefeed character or by reaching the limit for the line array. Within the loop a test is made for either a "control c" character or a "control l" character. If a "control c" is read, a "?" is written on the console and the index of the line array is decremented by 1. If a "control l" is read, a linefeed character followed by a "?" are written on the console.

The TYPERESOURCE MONITOR (Brinch Hansen, 1977, pp. 105 - 106) is used to gain exclusive access to the console. It consists of two ENTRY procedures: REQUEST and RELEASE. This type uses the FIFO CLASS to manage a QUEUE. The REQUEST procedure tests whether or not another process is currently using the console. If it

is, the process requesting access is placed on the QUEUE. The process accessing the console is then identified on the console. The RELEASE procedure checks the QUEUE to see if any processes are currently waiting to use the console. If the QUEUE is empty, then the console becomes free. Otherwise, the next process is taken off of the QUEUE and allowed to continue. The main body of this declaration initializes the FIFO CLASS variable in addition to initializing its passive type variables.

A TYPERESOURCE parameter and a TYPEWRITER variable are accessed by a variable of the TERMINAL CLASS type (Brinch Hansen, 1977, p. 109). This type uses the previous two types to gain exclusive access to the console, to identify its calling process, and to transfer the line of text either to or from the console. Two ENTRY procedures are used to accomplish this: READ and WRITE. The READ procedure requests access to the console through a TYPERESOURCE parameter. If the process requesting the console is different than the one that most recently accessed the console previously, the process name is written on the console. The line of text is then read from the console and access to the console is released. The write procedure

differs only in that instead of reading a line of text from the console, it writes a line of text on the console. The main body of this declaration initializes the TYPEWRITER CLASS variable.

The RESOURCE MONITOR (Brinch Hansen, 1977, pp. 104 - 105) type is very similar to the TYPERESOURCE MONITOR. It has two ENTRY procedures, REQUEST and RELEASE, which perform like those described above. This MONITOR, however, gives a process exclusive access to any of the computers resources as opposed to only the console. It therefore does not need to inform the resource as to which process has accessed it. It simply tests to see if the resource is available and delays or continues the processes accordingly. For this declaration, another active declaration is needed. This is for an ARRAY of QUEUE as follows:

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

The main body of this declaration initializes the FIFO CLASS variable and initializes the BOOLEAN variable to TRUE.

A single character is written onto or read from a TERMINAL CLASS parameter by a variable of the

TERMINALSTREAM CLASS type (Brinch Hansen, 1977, pp. 110 - 111). This type consists of three ENTRY procedures: READ, WRITE, and RESET, and a procedure used only by variables of that CLASS type. The local procedure is an initialization procedure used to initialize the header variable. The READ and WRITE procedures are used to read and write (respectively) a character to a variable of type TERMINAL CLASS. In the READ procedure the end of a line has been reached then the TERMINAL CLASS variable procedure READ is called and the count is reset to 0. If it is not the end of the line, then the next character from the text line array is assigned to the variable parameter c. The WRITE procedure executes in a similar manner. It increases the count and then stores a single character in an array of type line. When the end of the line is reached, the TERMINAL CLASS variable procedure WRITE is called and the text line array is passed to it. The procedure RESET is used to reinitialize the line of text. The main body of this declaration is a procedure call for the INITIALIZE procedure.

There are three buffer type MONITORs used in this program: ARGBUFFER, LINEBUFFER, and PAGEBUFFER (Brinch Hansen, 1977, pp. 125 - 126). They are different only

in the type of the buffer used. There are two ENTRY procedures: READ and WRITE. The READ procedure tests to see if the buffer is full. If it is, the message is assigned to a text variable and full is then assigned the value FALSE. The sending process then continues. If the buffer is not full, the receiving process is delayed before completing the procedure. The WRITE procedure is similar only that the operations are in reverse. The PAGEBUFFER MONITOR type also checks for the end of the file.

There are several CLASSes and MONITORs pertaining to disk use. The first, the DISK CLASS type (Brinch Hansen, 1977, pp. 112 - 113), transfers a page to or from a disk device. It also accesses the console to report a disk failure and to communicate with the operator concerning this error. This type consists of three procedures, two of which are ENTRY procedures. The TRANSFER procedure, which is local to this CLASS, either reads or writes a page from or to the disk. The page is identified by its absolute page address. Whether the procedure reads or writes, using a TERMINAL CLASS type variable is determined by a parameter. The page address is also passed as a parameter. The IO procedure is used by this TRANSFER procedure as it was

in the TYPEWRITER CLASS type. The two remaining procedures: READ and WRITE, simply have calls to TRANSFER. The only difference between the two is in one of the parameters. The READ procedure passes input as a parameter and the WRITE procedure passes output. The type page is a universal type. This allows the DISK CLASS to transfer pages of different types.

The DISK CLASS type is accessed by the DISKFILE CLASS type (Brinch Hansen, 1977, pp. 114 - 115). The purpose of this type is to make it possible for a process to access a disk file. If a disk failure occurs, the TYPERESOURCE CLASS parameter is accessed to communicate exclusively with the console. This type has a BOOLEAN function INCLUDES which is TRUE only if a given page number is within the proper range and a file is to be accessible. There are also four ENTRY procedures: OPEN, CLOSE, READ, and WRITE. The READ and WRITE procedures use the DISK CLASS type variable to transfer a page from or to a disk. The OPEN procedure assigns a page map to a file and makes it accessible. The CLOSE procedure makes the file inaccessible and resets the length of the file to 0. The main body of the type declaration sets the length to 0, the accessability variable to FALSE, and initializes the

DISK CLASS variable. It should also be noted that the variable length in this declaration is an ENTRY variable. This allows it to be used outside the CLASS. Its value, however, can only be changed within the CLASS.

The DISKTABLE CLASS type (Brinch Hansen, 1977, pp. 116 - 117) uses both a TYPERESOURCE type parameter and a DISKFILE type variable. The TYPERESOURCE parameter is again accessed to report disk failure as mentioned above. It uses the DISKFILE to gain access to locate a catalog on a disk. The main body of the declaration cosists of initializing the DISKFILE variable, accessing the DISKFILE procedure OPEN, and initializing the local variables. The one ENTRY procedure in this declaration, procedure ENTRY READ, uses the DISKFILE to read an entry at a given location in the catalog.

Catalog lookup is managed by the DISKCATALOG MONITOR type (Brinch Hansen, 1977, pp. 117 - 118). A TYPERESOURCE parameter is used as mentioned above for disk failure. A RESOURCE type parameter is used to gain exclusive access to the disk. This type also uses a DISKTABLE variable to search for a file identifier. There is a local function HASH which returns a value for the hash key. There is also one ENTRY procedure,

LOOKUP. The LOOKUP porcedure is a search procedure using the hash key. A variable BOOLEAN parameter returns the appropriate value indicating if the identifier was found. If the identifier was found, the procedure also returns the file attributes. The body of the declaration initializes the DISKTABLE variable.

The last of the disk accessing CLASSes and MONITORs is the DATAFILE CLASS type (Brinch Hansen, 1977, pp. 119 - 121). It is with this CLASS that a process accesses a file of a given identifier name. It accesses a parameter of type RESOURCE to gain access to the disk and a parameter of type DISKCATALOG to look up the file. A parameter of type TYPERESOURCE is used to access the console to report disk failure. A variable of type DISKFILE is used to open and close files. There are four ENTRY procedures: OPEN, CLOSE, READ, and WRITE. The READ and WRITE procedures simply request access to the disk using the RESOURCE parameter, read or write to the file using the DISKFILE variable, and release the disk again using the RESOURCE parameter. The CLOSE procedure closes a file using the DISKFILE procedure CLOSE and reinitializes the local variables. The OPEN procedure accesses the DISKCATALOG parameter to perform a lookup. If the file is found, then the

procedure requests use of the disk through the RESOURCE parameter, opens the file using the DISKFILE procedure open, resets the length variable, and releases the disk through the RESOURCE parameter. The main body of this type initializes the DISKFILE variable and the local variables.

The PROGFILE CLASS type (Brinch Hansen, 1977, p. 122) is used to transfer a sequential Pascal program from disk into core. It accesses a TYPERESOURCE parameter to communicate with the console in the case of disk failure, a RESOURCE parameter to gain exclusive access to the disk, and a DISKCATALOG parameter to lookup the file on the disk. A DISKFILE variable is used to read the program from the file. This type consists of a single ENTRY procedure, OPEN. After the file is looked up, tests are performed to make sure it is found and and that the file contains sequential code. If both of these conditions are satisfied then the disk is requested, the file is opened, and the program is read. Another test is made to ensure that the length of the file does not exceed the space allotted in core. The main body initializes the variable of type DISKFILE.

The PROGSTACK type (Brinch Hansen, 1977, pp. 123 -

124) is a MONITOR used to manage the nested calls of programs from one to another. It maintains a Lifo (last in, first out) stack. Two BOOLEAN ENTRY functions, SPACE and ANY, are used to determine if the stack has run out of space or is empty (respectively). There are also three ENTRY procedures: PUSH, POP, and GET. PUSH is used to put an identifier on the stack. The POP procedure, in addition to removing an identifier from the stack, returns the attributes of the termination of the program. The GET procedure identifies the program at the top of the stack. The main body of this type initializes the top of the stack to 0. No other CLASSes or MONITORs are accessed by this type.

PROCESSes communicate with each other through access to the CHARSTREAM CLASS (Brinch Hansen, 1977, pp. 126 - 127). Messages are passed character by character and a PAGEBUFFER parameter is used to send and receive a page of characters. There are four ENTRY procedures: INITREAD, INITWRITE, READ, and WRITE. The INITREAD and INITWRITE open the CHARSTREAM for reading and writing respectively. Once a PROCESS has opened the CHARSTREAM, it can then READ or WRITE a single character. The PAGEBUFFER MONITOR is used to manage

the reading and writing.

The remainder of the declarations are the PROCESS declarations. They are accessed only by the initialization PROCESS. There are five types of concurrent PROCESSes used: LOADERPROCESS, CARDPROCESS, PRINTERPROCESS, JOBPROCESS, and IOPROCESS.

The purpose of the LOADERPROCESS (Brinch Hansen, 1977, pp. 139 - 140) is to reinitialize the Solo operating system. The process interrupts the operating system and waits for a signal (the BEL key) from the console. It receives the signal through the IO procedure. When the signal is received, the PROCESS requests access to the disk through the RESOURCE parameter. It reloads the the system and then releases the disk.

The CARDPROCESS (Brinch Hansen, 1977, pp. 137 - 138) and PRINTERPROCESS (Brinch Hansen, 1977, pp. 138 - 139) are similar processes. The CARDPROCESS sends data from a card reader to a variable of type IOPROCESS. The PRINTERPROCESS sends data from an IOPROCESS to a lineprinter. The program has only one variable of each type. This is to ensure that each of these devices is controlled by a single process. These PROCESSes use a LINEBUFFER parameter to send and receive the data to

and from the IOPROCESS. A TYPERESOURCE parameter and a variable of type TERMINAL are used to inform the console that an error has been detected. The declarations begin by initializing the TERMINAL variable. They then enter infinite loops in which the CARDPROCESS reads any of the cards in the card reader and the PRINTERPROCESS writes any data received from the IOPROCESS to the lineprinter. This is accomplished using the IO procedure. Each type uses a standard procedure WAIT to delay the process if either in the case of CARDPROCESS there are no cards to read or, in the case of PRINTERPROCESS, there is no data to be sent to the lineprinter.

The JOBPROCESS (Brinch Hansen, 1977, pp. 129 - 132) and the IOPROCESS (Brinch Hansen, 1977, pp. 133 - 136) are similar in structure. The JOBPROCESS is used to execute sequential Pascal programs which can call other sequential Pascal programs recursively. The IOPROCESS executes sequential Pascal programs that send (or receive) data to (or from) the JOBPROCESS. They both can implement interface procedures between the programs and the operating system. Each PROCESS has parameters of type TYPERESOURCE, RESOURCE, and DISKCATALOG. The results of accessing these parameters

has been explained previously. The JOBPROCESS uses two PAGEBUFFER parameters and four ARGBUFFER parameters to interact with two IOPROCESSes. Similarly, the IOPROCESS has one PAGEBUFFER parameter and two ARGBUFFER parameters to interact with the JOBPROCESS. Both PROCESSes use a PROGSTACK parameter to manage the nested program calls mentioned previously. In addition, the IOPROCESS uses a LINEBUFFER parameter to access an IO device. These PROCESSes also have variables of type TERMINAL, TERMINALSTREAM, and DATAFILE which have also previously been discussed. The PROCESSes each have a PROGFILE variable which is used to store the currently executed program and CHARSTREAM variables for communicating with each other. The declarations each contain a sequential program routine which specifies the routine entries called by the program. Each ENTRY routine is also declared. These are simple procedures and functions which access other CLASSes and MONITORS within the system. These interface routines can only be accessed by the sequential program. After initializing its variables, each PROCESS calls a CALL procedure which is local to the PROCESS. The CALL procedure loads the program from the disk into core using he PROGSTACK parameter and the

PROGFILE variable. Initially, the JOBPROCESS executes a sequential program DO which reads the users program identifier from the console. The IOPROCESS initially executes the IO program which begins the reading of cards for an input PROCESS and the writing to a lineprinter for an output PROCESS. The PROCESSes send a termination message to the console upon completion of their respective initialization procedures.

The declarations descibed here, along with the INITIAL PROCESS make up the Solo Operating System. The redundancy of the parameters and variables allows the system to check the access rights during compilation. A component can access only those components it has declared as parameters or variables. Access rights are restricted by the rules of the Concurrent Pascal Language. By using these access rights, critical regions are managed. By not allowing components to call each other recursively, deadlock is avoided.

Debugging is facilitated by bottom-up testing. For example, once the FIFO component is debugged, it will not cause errors in the TYPERESOURCE component. Any errors encountered there are specific to that component.

VIII. CONCLUSION

This thesis has examined various aspects of concurrent programming. The problems inherent in simulating concurrent processes through software were discussed and several solutions were given. While a slight extension of sequential Pascal may be sufficient to manage simulated concurrent processing, more elegant and efficient solutions result from a set of formal structures as in Concurrent Pascal. The introduction of the structures of this language also provide extended compiler error checking (through type checking), and a means for hierarchical programming. The example used throughout the chapter on Concurrent Pascal demonstrates the differences in the structures used in the three extensions of Pascal. The Solo Operating System shows Concurrent Pascal to be an effective tool for facilitating the writing of operating systems.

As concurrent programming becomes more extensive, abstract languages, like Concurrent Pascal, will provide the means for creating simple, reliable, and adaptable programs.

REFERENCES

1. Ben-Ari, M. Principles of Concurrent Programming, Prentice-Hall International, Inc., U.S.A., 1982.

2. Brinch Hansen, Per. Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

3. Brinch Hansen, Per. The Architecture of Concurrent Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

4. Conway, Melvin E., "A Multiprocessor System Design", AFIPS, Vol 24, (Fall 1963), pp. 139-146.

5. Corbato, Fernando J., Merwin-Daggett, Marjorie, and Daley, Robert C., "An Experimental Time-sharing System", AFIPS, Vol 21, (Spring 1962), pp. 335-344.

6. Findlay, William and Watt, David A. PASCAL An Introduction to Methodical Programming (2nd ed), Computer Science Press, Inc., Maryland, 1981.

7. Jensen, Kathleen and Wirth, Niklaus. PASCAL User Manual and Report (2nd ed), Springer-Verlag, U.S.A., 1978.

8. Leiner, A. L., Notz, W. A., Smith, J. L., and Weinberger, A., "PILOT - A New Multiple Computer System", J. ACM, Vol 6, No 3 (July 1959), pp. 313-335.

9. Peterson, G. L., "Myths About the Mutual Exclusion Problem", Information Processing Letters, Vol 12, No 3, (1981), pp. 115-116.

10. Shelly, Gary B. and Cashman, Thomas J. Introduction to Computers and Data Processing, Anaheim Publishing Co., U.S.A., 1980.

APPENDIX

HIERARCHICAL OUTLINE OF THE SOLO OPERATING SYSTEM

```
CARDPROCESS
  TYPERESOURCE MONITOR
    FIFO CLASS
  LINEBUFFER MONITOR
  TERMINAL CLASS
    TYPERESOURCE MONITOR
      FIFO CLASS
    TYPEWRITER CLASS
PRINTERPROCESS
  TYPERESOURCE MONITOR
    FIFO CLASS
  LINEBUFFER MONITOR
  TERMINAL CLASS
    TYPERESOURCE MONITOR
      FIFO CLASS
    TYPEWRITER CLASS
IOPROCESS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
    RESOURCE MONITOR
      FIFO CLASS
    DISKTABLE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISKFILE CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        DISK CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TERMINAL CLASS
            TYPERESOURCE MONITOR
              FIFO CLASS
            TYPEWRITER CLASS
```

```
LINEBUFFER MONITOR
PAGEBUFFER MONITOR
ARGBUFFER MONITOR
PROGSTACK MONITOR
TERMINAL CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  TYPEWRITER CLASS
TERMINALSTREAM CLASS
  TERMINAL CLASS
    TYPERESOURCE MONITOR
      FIFO CLASS
    TYPEWRITER CLASS
CHARSTREAM CLASS
  PAGEBUFFER MONITOR
DATAFILE CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
    RESOURCE MONITOR
      FIFO CLASS
    DISKTABLE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISKFILE CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        DISK CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TERMINAL CLASS
            TYPERESOURCE MONITOR
              FIFO CLASS
            TYPEWRITER CLASS   PROGFILE CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
    RESOURCE MONITOR
      FIFO CLASS
```

```
      DISKTABLE CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        DISKFILE CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          DISK CLASS
            TYPERESOURCE MONITOR
              FIFO CLASS
            TERMINAL CLASS
              TYPERESOURCE MONITOR
                FIFO CLASS
              TYPEWRITER CLASS
    DISKFILE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISK CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        TERMINAL CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TYPEWRITER CLASS
LOADERPROCESS
  RESOURCE MONITOR
    FIFO CLASS
JOBPROCESS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
    RESOUCE MONITOR
      FIFO CLASS
    DISKTABLE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISKFILE CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        DISK CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TERMINAL CLASS
            TYPERESOURCE MONITOR
```

```
          FIFO CLASS
        TYPEWRITER CLASS
PAGEBUFFER MONITOR
ARGBUFFER MONITOR
PROGSTACK MONITOR
TERMINAL CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  TYPEWRITER CLASS
TERMINALSTREAM CLASS
  TERMINAL CLASS
    TYPERESOURCE CLASS
      FIFO CLASS
    TYPEWRITER CLASS
CHARSTREAM CLASS
  PAGEBUFFER MONITOR
DATAFILE CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
    RESOURCE MONITOR
      FIFO CLASS
    DISKTABLE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISKFILE CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        DISK CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TERMINAL CLASS
            TYPERESOURCE MONITOR
              FIFO CLASS
            TYPEWRITER CLASS
PROGFILE CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  RESOURCE MONITOR
    FIFO CLASS
  DISKCATALOG MONITOR
    TYPERESOURCE MONITOR
      FIFO CLASS
```

```
  RESOURCE MONITOR
    FIFO CLASS
  DISKTABLE CLASS
    TYPERESOURCE MONITOR
      FIFO CLASS
    DISKFILE CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      DISK CLASS
        TYPERESOURCE MONITOR
          FIFO CLASS
        TERMINAL  CLASS
          TYPERESOURCE MONITOR
            FIFO CLASS
          TYPEWRITER CLASS
DISKFILE CLASS
  TYPERESOURCE MONITOR
    FIFO CLASS
  DISK CLASS
    TYPERESOURCE MONITOR
      FIFO CLASS
    TERMINAL CLASS
      TYPERESOURCE MONITOR
        FIFO CLASS
      TYPEWRITER CLASS
```

VITA

Barbara Harab Smolowitz was born January 29, 1950, in Washington, D.C. She graduated from Bethesda-Chevy Chase High School, Bethesda, Maryland in 1968. in 1972, she graduated from Carnegie-Mellon University, Pittsburgh, Pennsylvania with a B. A. Degree in Mathematics. She was then inducted into Pi Mu Epsilon, the National Honorary Mathematics Fraternity.

During the academic year 1973-74, she taught mathematics at Oak Harbor Junior High School, Oak Harbor, Washington. From 1974 until 1976, she taught college preparatory mathematics at Nazareth Senior High School, Nazareth, Pennsylvania.

In 1977 she received an M. A. Degree in Secondary Education from Lehigh University, Bethlehem, Pennsylvania. She began her studies in the Division of Computing and Information Science at Lehigh University in the fall of 1982.

She currently resides in Bridgewater, New Jersey with her husband and two daughters.