1979

Concur: a High-Level Language for Concurrent Programming

Karen Anderson

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
CONCUR: A HIGH-LEVEL LANGUAGE FOR
CONCURRENT PROGRAMMING

A Thesis Submitted in Partial Fulfillment of the
Master of Science in Computer Science Degree Program.

BY: Karen K. Anderson

Approved By:

Date:
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>The Language CONCUR</td>
<td></td>
</tr>
<tr>
<td>Basic Symbols</td>
<td>4</td>
</tr>
<tr>
<td>Data Types</td>
<td>5</td>
</tr>
<tr>
<td>Data Structure</td>
<td>6</td>
</tr>
<tr>
<td>Program Structure</td>
<td>6</td>
</tr>
<tr>
<td>Constant Declarations</td>
<td>6</td>
</tr>
<tr>
<td>Variable Declarations</td>
<td>6</td>
</tr>
<tr>
<td>Process Declarations</td>
<td>7</td>
</tr>
<tr>
<td>Expressions</td>
<td>8</td>
</tr>
<tr>
<td>Assignment Statements</td>
<td>8</td>
</tr>
<tr>
<td>Process Invocations</td>
<td>9</td>
</tr>
<tr>
<td>IF Statements</td>
<td>9</td>
</tr>
<tr>
<td>LOOP Statement</td>
<td>10</td>
</tr>
<tr>
<td>Semaphore Manipulation</td>
<td>11</td>
</tr>
<tr>
<td>Input/Output</td>
<td>13</td>
</tr>
<tr>
<td>Statement Sequences</td>
<td>13</td>
</tr>
<tr>
<td>Processes</td>
<td>14</td>
</tr>
<tr>
<td>The Use of CONCUR</td>
<td></td>
</tr>
<tr>
<td>Overview</td>
<td>15</td>
</tr>
<tr>
<td>A Programming Approach</td>
<td>17</td>
</tr>
<tr>
<td>The Producer/Consumer Example</td>
<td>18</td>
</tr>
<tr>
<td>Sequential Programming in CONCUR</td>
<td>24</td>
</tr>
<tr>
<td>The Concurrent Quicksort Example</td>
<td>26</td>
</tr>
<tr>
<td>The Implementation of CONCUR</td>
<td></td>
</tr>
<tr>
<td>Overview</td>
<td>35</td>
</tr>
<tr>
<td>The Interpreter</td>
<td>36</td>
</tr>
<tr>
<td>The Machine Instruction Set</td>
<td>40</td>
</tr>
<tr>
<td>Code Generation</td>
<td>44</td>
</tr>
<tr>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>Appendix A</td>
<td></td>
</tr>
<tr>
<td>CONCUR Syntax Graphs</td>
<td>59</td>
</tr>
<tr>
<td>Appendix B</td>
<td></td>
</tr>
<tr>
<td>Sample Programs</td>
<td>65</td>
</tr>
<tr>
<td>Appendix C</td>
<td></td>
</tr>
<tr>
<td>Source Listing of CONCUR Compiler</td>
<td>99</td>
</tr>
<tr>
<td>Appendix D</td>
<td></td>
</tr>
<tr>
<td>Compiler Operating Instructions</td>
<td>151</td>
</tr>
<tr>
<td>Appendix E</td>
<td></td>
</tr>
<tr>
<td>Compilation Errors</td>
<td>153</td>
</tr>
<tr>
<td>Appendix F</td>
<td></td>
</tr>
<tr>
<td>Source Program with Object Code Listing</td>
<td>158</td>
</tr>
<tr>
<td>References</td>
<td>165</td>
</tr>
</tbody>
</table>
ABSTRACT

A language CONCUR is defined which permits the definition and initiation of asynchronous processes. The language was inspired by Modula, a language proposed by Wirth for realtime programming. CONCUR removes Modula's restrictions on the placement of process declarations and invocations in order to study the implications of process support more fully. Most of the other sophisticated features of Modula, such as modules, structure types, and procedures, have also been removed to focus attention on processes and their particular requirements. A general methodology is suggested for concurrent programming, and several sample programs are presented which demonstrate concurrent programming with CONCUR. Finally, a compiler is presented which translates CONCUR into the object language for a hypothetical machine. An interpreter for this object language is also included.
INTRODUCTION

With current technology processor speeds have approached the natural limitations of the materials from which the processors are made. Quantum reductions in program execution time through hardware breakthroughs, as have been seen in the past, cannot really be expected. Future significant increases in processing speed appear to be possible only through multiprocessing. (1)

Parallelism has been traditionally employed in hardware design to create faster equipment. More recently, parallelism has been recognized as applicable to operating systems to improve their efficiency. The last area for application of parallelism is within individual programs. A high-level language supporting the creation of concurrent processes within a program would facilitate effective parallel programming.

Such languages are beginning to emerge, two examples being Brinch Hansen's Concurrent PASCAL (2) and Wirth's Modula (3,4,5,6). This thesis presents a language called CONCUR, which was inspired primarily by Modula. CONCUR is not offered as an alternative to the production languages mentioned. It serves instead as a vehicle for exploring aspects of high-level language support for parallel processing. This study
should increase understanding of language design and compiler writing. In addition, selected concepts in operating systems and machine architecture will be explored. Finally, some of the implications of concurrent programming for programming methodology will be discussed.
THE LANGUAGE CONCUR

BASIC SYMBOLS

The basic symbols from which a CONCUR program is built are numbers, identifiers, keywords, and special symbols.

A number must be an integer. It can be signed or unsigned.

An identifier must begin with a letter and must contain only letters and digits. There is no limit to the number of characters in the identifier, but only the first twelve are used by the compiler.

Keywords are used to show explicitly the structure of a program. The keywords are reserved; i.e., they may not be used as identifiers. The following words are keywords.

<table>
<thead>
<tr>
<th>AND</th>
<th>ELSEIF</th>
<th>OF</th>
<th>THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>END</td>
<td>OR</td>
<td>VAR</td>
</tr>
<tr>
<td>BEGIN</td>
<td>EXIT</td>
<td>PROCESS</td>
<td>VSIGNAL</td>
</tr>
<tr>
<td>CONST</td>
<td>IF</td>
<td>PWAIT</td>
<td>WHEN</td>
</tr>
<tr>
<td>DO</td>
<td>INTEGER</td>
<td>READ</td>
<td>WRITE</td>
</tr>
<tr>
<td>DUMP</td>
<td>LOOP</td>
<td>SEMA</td>
<td>XOR</td>
</tr>
<tr>
<td>ELSE</td>
<td>NOT</td>
<td>SET</td>
<td></td>
</tr>
</tbody>
</table>

Special symbols serve as operators or delimiters. Some of them are composed of two characters which must appear contiguously.
ASSIGNMENT OPERATOR  :=
RELATIONAL OPERATORS  =  <>  <  <=  >  >=
ARITHMETIC OPERATORS  +  -  *  /
DELIMITERS  :  ;  ,  ( )  ( . )  ( * *)  .

Blanks, end-of-line characters, and comments are used to separate consecutive symbols but are otherwise ignored.

Comments are enclosed by (* *). They are listed but are not considered part of the program. Comments may appear between any two symbols in a program.

DATA TYPES

Two types of data are available in CONCUR: integer and semaphore.

A semaphore is a synchronization element which is manipulated by a restricted set of operators. A semaphore may be initialized by the SET operation and modified and tested by the PWAIT and V SIGNAL operations. Semaphores are explained in more detail on pages 11 through 13 and pages 16 and 17.

Since there is no Boolean type, relational operators produce integer results. True is represented as the integer one, and false is represented by the integer zero. In cases where the values cannot be closely controlled, any non-zero value is considered true, and a zero value is considered false.
DATA STRUCTURE

The only data structure permitted is the one-dimensional array. Array elements must be integers. Legal subscripts for an array range from one to the declared length. A specific element of an array can be referenced by the array name and an index enclosed in brackets ( . . ).

PROGRAM STRUCTURE

A CONCUR program consists of a block followed by a period. A block has a declaration part and a statement part. Within the declaration part symbolic constants, identifiers, and processes are defined. Within the statement part the algorithmic actions to be performed are specified. A process itself contains a block. Thus, a block is defined recursively.

CONSTANT DECLARATIONS

A constant declaration relates a symbolic name to a constant numeric value.

Examples of constant declarations are:

\begin{verbatim}
CONST LIMIT = 10;
ZERO = 0;
\end{verbatim}

VARIABLE DECLARATIONS

A variable declaration establishes the name, type, and structure of a variable; however, no initialization of the variable takes place. The program can reference that variable by its name or, in the case of an array, its name and an index.
The index may be a constant, a variable, or an expression.

Examples of variable declarations are:

```
VAR SUM: INTEGER;
LIST: ARRAY 5 OF INTEGER;
MUTEX: SEMA;
```

Example variables are:

```
SUM LIST(.I-1.) MUTEX
```

**PROCESS DECLARATIONS**

A process declaration defines a program segment called a process that can be activated by an explicit call. The declaration first names the process and describes its parameters. Then the body of the process follows as a block. Finally, the process is delimited by its name.

Examples of process declarations are:

```
PROCESS POWER (AVALUE: INTEGER, ITSPOWER: INTEGER)
```

```
BLOCK
```

```
POWER;
```

```
PROCESS WAITLOOP;
```

```
BLOCK
```

```
WAITLOOP;
```

(Processes are discussed in greater depth on page 14.)
**EXPRESSIONS**

An expression is a rule for computing values. The rule is defined in terms of operators, operands (constants or variables), and parentheses. The operators are grouped into four priority classes. The relational operators (= <> < <= > >=) have the lowest priority. The arithmetic operators + and - and the logical operators OR and XOR have the next highest priority. The arithmetic operators * and / and the logical operator AND have the next highest priority. The logical operator NOT has the highest priority. Operators of equal priority are applied left to right. Parentheses can be used to override the normal order of application and to form compound logical expressions.

Examples of expressions are:

- 2*PI*D
- PAYMENT < MINIMUMDUE
- (HOURS = 0) OR ((GROSSPAY-TAXES-DEDUCTIONS)<0)

**ASSIGNMENT STATEMENTS**

An assignment statement places the value of an expression into a variable. The variable must be of type integer or array. If the expression results in a logical value, the value will be expressed automatically as an integer. True equals one and false equals zero.

Examples of assignment statements are:

- START := 1
- COST := LIST - DISCOUNT
- AVG := (T1 + T2 + T3)/3
PROCESS_INVOCATIONS

A process is called into execution when its name appears in the statement part of a block. The name may be followed by a parameter list of constants, variables, or expressions separated by commas. All parameters are passed by value.

Examples of process invocations are:

POWER (VALUE-AVERAGE, 2) .
WAITLOOP

IF STATEMENTS

An IF statement selects actions for execution on the basis of a logical expression. Three forms of the IF are allowed. The first (IF...THEN...) only specifies actions to be performed when the expression is true. The second (IF...THEN...ELSE...) specifies actions for both the true and false conditions. The third (IF...ELSIF...THEN...) permits nesting of IF statements. The ELSIF can be repeated many times within the IF to achieve any level of nesting. Each form of the IF is terminated with the keyword END.

Examples of IF statements are:

```
IF NETPAY < 0 THEN NETPAY := 0 END
IF A > B THEN
   ABSDIF := A - B
ELSE
   ABSDIF := B - A
END
IF CREDITHRS < 12 THEN
   TUITION := 82 * CREDITHRS
ELSIF CREDITHRS > 18 THEN
   TUITION := 966 + 82 * (CREDITHRS-18)
ELSE
   TUITION := 966
END
```
LOOP STATEMENT

A loop statement is a generalized form of iteration which can serve as any of the DO-UNTIL, DO-WHILE, or DO-WHILE-DO constructs. The WHEN portion of a loop statement acts as a conditional break. Placement of the WHEN at the bottom of a loop creates a DO-UNTIL; placement at the top of a loop creates a DO-WHILE; and placement in the midst of the loop creates a DO-WHILE-DO.

A bodyless loop with a series of WHEN portions can serve as a case statement, since each WHEN may be followed by a statement sequence that is performed once before exiting the loop. Any loop statement must terminate with the keyword END.

Examples of loop statements are:

```
LOOP
\{\}
WHEN INDEX = DIMENSION EXIT
END

LOOP
WHEN LIST = EMPTY EXIT
\{\}
END

LOOP
\{\}
WHEN INPUT = ENDSIGNAL EXIT
\{\}
END
```

10
SEMAPHORE MANIPULATION

As noted previously, semaphores can be referenced by only three kinds of statements. These are SET statements, PWAIT statements, and V SIGNAL statements.

A SET statement initializes a semaphore to the value of a specified expression. In addition, it clears the wait queue for the semaphore. The programmer must be very careful to include a properly placed SET statement for each semaphore declared. If a SET is omitted, neither the semaphore's value nor the pointer to its wait queue are initialized. The arbitrary values which they would have could cause a PWAIT or V SIGNAL to produce quite bizarre behavior. The value of the semaphore determines whether processes are suspended or
released and prompts manipulation of the wait queue. Also, since any pointer other than the end-of-queue value is interpreted as a process number, the next process performing a PWAIT on the semaphore would be linked to this "arbitrary process". Similarly, a VSIGNAL could enter the arbitrary process into the ready list. If a SET is misplaced or repeated, the initialization of the pointer to the wait queue would cause any processes already waiting on the semaphore to be lost when the SET is executed. The language does not prevent such misuse of the SET statement.

Examples of SET statements are:

\[
\text{SET (MUTEX,1)} \\
\text{SET (AVAIL,NOBUFFERS-2)}
\]

A PWAIT statement follows the formal definition of the P operation. Specifically, it decrements the semaphore by 1 and then tests the resultant value. If the semaphore is greater or equal to zero, execution continues; otherwise execution is suspended and the process containing the PWAIT is inserted into the wait queue for the semaphore. Clearly, the PWAIT operation can cause a semaphore to take on negative values. If a semaphore is negative, its absolute value shows the number of processes waiting on that semaphore.

A VSIGNAL statement follows the formal definition of the V operation. That is, it increments the semaphore by 1 and then tests the resultant value. If the semaphore is greater than zero, execution continues; otherwise execution continues
after another process is deleted from the wait queue for that semaphore and made ready for execution.

Examples of \texttt{PWAIT} and \texttt{VSIGNAL} statements are:

$$\text{PWAIT(MUTEX)}$$
$$\text{VSIGNAL(AVAIL)}$$

\textbf{INPUT/OUTPUT}

Input and output operations in CONCUR are very primitive. The operations are: read a single integer value from a new record, write a single integer value on a new line, and dump the active portions of descriptor store and data store. The argument of a \texttt{READ} statement must be either an integer variable or an array item. The argument of a \texttt{WRITE} statement can be an expression. The \texttt{DUMP} takes no arguments.

Examples of input/output statements are:

$$\text{READ(MATRIX(.K.))}$$
$$\text{WRITE(RATE*TIME)}$$
$$\text{DUMP}$$

\textbf{STATEMENT SEQUENCES}

A statement sequence is a series of statements separated by semicolons. The last statement in a sequence does not require a semicolon since the last statement will always be followed by a keyword. Statements in a statement sequence are executed sequentially in the order of their occurrence.
PROCESSES

Processes exist in CONCUR to permit the programming task to be broken into subtasks. Procedures in languages such as ALGOL, PL/I, and PASCAL serve the same purpose. Unlike a procedure, however, a process has an independent character. That is, a calling process does not wait until the called process completes, and many processes may be executed concurrently. Synchronization of process execution is accomplished by using semaphores.

All constants, variables, and processes defined within a process are local to that process. Communication among processes must take place through parameters (one-way communication with call by value) and variables which are "global" to both processes.

The main block may be considered a process which is called into execution at the beginning of program execution. The process terminates when the last statement in the block is executed, but the program does not end until all activated processes are terminated.

Syntax graphs for CONCUR are shown in APPENDIX A.
THE USE OF CONCUR

OVERVIEW

The value of concurrent programming is derived from the basic premise that a task or process can often be broken into subtasks or subprocesses which can be performed independently and simultaneously. Complete independence of subprocesses is generally an unrealistic expectation, however. (Two subtasks which are in no way dependent on each other might more logically be considered separate tasks altogether.) Instead, one can expect to encounter processes whose subprocesses are what Dijkstra calls "loosely connected" (7,p.52). That is, the subprocesses are independent of each other except for infrequent periods of explicit intercommunication. "Loosely connected" further implies that no assumptions can be made about the relative speeds of the subprocesses. Successful completion of the whole task cannot depend on one subtask consistently completing before another.

Intercommunication suggests sharing some resource, such as a variable, data structure, or I/O device. A shared resource which can only be used by one process at a time is
called a "critical resource" (1,p.27). That portion of a process which refers to a critical resource is called a "critical section" (7,p.53). In order to insure mutually exclusive access to a resource, a mechanism must be provided to synchronize the execution of parallel processes. This mechanism must satisfy two criteria:

1) If one or more processes attempt to enter their critical sections at the same time, at least one will be able to enter.

2. Only one process can be in its critical section at any given time. (1)

Dijkstra has proposed a synchronization mechanism called a semaphore. (7) A semaphore is a special-purpose integer which may be accessed only through two primitive operations—the P operation and the V operation. The P operation decreases the value of the semaphore by one. If the value of the semaphore becomes negative, the process executing the P operation cannot proceed. In other words, the process must wait. A waiting process is liberated when another process performs the V operation. The V operation increases the value of the semaphore by one. If the value of the semaphore remains negative or becomes zero, there must be at least one process waiting at a P operation. Thus, the V selects one waiting process and permits it to proceed. The process performing the V operation can also continue execution.
Semaphores may be classified as binary semaphores or general semaphores. There is no syntactic difference between the two types of semaphore, but they can be distinguished by their use and their maximum values. Binary semaphores are used to enforce mutual exclusion. The maximum value of a binary semaphore is one, since there is essentially only one "permit" for which processes must compete in order to enter their critical sections. General semaphores are used to allocate more plentiful resources. The maximum value of a general semaphore is determined by the number of units (e.g. buffers, disk drives, terminals,...) available. Note that the initial value of a semaphore is also crucial. The initial value indicates how many processes, if any, will be initially granted unrestrained access to the resource associated with the semaphore.

A PROGRAMMING APPROACH

From the above discussion, a general approach to writing a concurrent program in CONCUR can be suggested. Given a task, first identify the independent or loosely connected subtasks into which it can be divided. Each subtask should be accomplished by a process call.

The next step is to determine which resources are shared among the subtasks. These are critical resources. Then, pinpoint the critical sections within each process. (To take
full advantage of concurrent execution, the critical sections should be kept as short as possible.) Mutual exclusion must be enforced for the critical sections; thus, introducing a binary semaphore, surround each critical section between a P operation (a PWAIT instruction) and a V operation (a VSIGNAL instruction) on that semaphore. A distinct semaphore will be needed for each type of critical resource. If several units of a resource are available, introduce a general semaphore for resource allocation, in addition to the binary semaphore.

Establishing the initial value of each semaphore is the final crucial step. Initial values are established using a SET instruction. Mutual exclusion semaphores should be initialized to one so that one process will be permitted to enter its critical section. Resource allocation semaphores should be initialized to the number of units originally available. For example, if a pool of six tape drives are available at the start, the general semaphore used to allocate tape drives should be initialized to six.

**THE PRODUCER/CONSUMER EXAMPLE**

A classic problem to illustrate concurrent programming and the synchronization primitives is the producer/consumer problem. The problem involves passing blocks of data from a "producer" to a "consumer". A specific example of such a problem is the task of reading a list of integers from an
input device and writing the list on an output device. In this case, the producer is a process to bring an integer into memory, and the consumer is a process to take the integer from memory and send it out. The two processes share the memory location to which the producer moves the integer and from which the consumer takes it. If a single memory location is used, however, the producer and consumer processes are not loosely connected. In fact, they are quite dependent on each other's execution. The consumer cannot move the integer from memory until the producer has placed it there, and the producer cannot place the next integer in memory until the consumer has moved the previous one. A pool of memory locations must be introduced to decrease this dependency so that concurrent operation is feasible.

In the sample program shown in Appendix B the main program performs the original task. To allow parallel processing, it accomplishes the task by invoking the producer and consumer subprocesses. Thus, the main process simply establishes and initializes the memory pool which the processes will share and then invokes them. The memory pool is implemented as a queue.
The producer process reads an integer, places it in a free memory location and links that location to the queue. It continues reading and storing until an end signal is detected.

```pascal
PROCEDURE PRODUCER;
VAR INPUTDATA: INTEGER;
BEGIN
  INDEX := 0;
  LOOP
    READ(INPUTDATA);
    INDEX := 0;
    LOOP
      INDEX := INDEX + 1
      WHEN BUFFERPTR(INDEX) = 0 DO
        IF QUEUEBEGIN = EMPTY THEN
          QUEUEBEGIN := INDEX;
          QUEUEND := INDEX
        END;
        BUFFER(QUEUEND) := INDEX;
        BUFFER(INDEX) := INPUTDATA;
        BUFFERPTR(INDEX) := EMPTY;
        QUEUEND := INDEX
      END;
    END;
    WHEN INPUTDATA = ENDSIGNAL EXIT
  END
END PRODUCER;
```
The consumer process takes an integer from the beginning of the queue and writes it out. It repeats the activity until it detects the end signal. The end signal is not written out.

```
PROCESS CONSUMER;
VAR OUTPUTDATA:INTEGER;
BEGIN
  LOOP
    INDEX := QUEUEBEGIN;
    OUTPUTDATA := BUFFER(INDEX);
    QUEUEBEGIN := BUFFERPTR(INDEX);
    BUFFERPTR(INDEX) := 0
    WHEN OUTPUTDATA = ENDSIGNAL EXIT
    WRITE(OUTPUTDATA)
  END
END CONSUMER;
```

At this point the main program and its subprocesses contain all the instructions necessary to accomplish their task; but the program as written is prone to error. The memory pool established in the main program is shared by the producer and consumer processes. Therefore, it is a critical resource and those instructions that manipulate it in each process comprise critical sections. A binary semaphore must be introduced to guarantee mutually exclusive execution of these sections. Following the naming convention of Tsichritzis and Bernstein (1), the semaphore is called MUTEX. The main program must declare the semaphore.

```...
MUTEX:SEMA;
```

There is no need to use the semaphore in the main program when initializing the memory pool, since no subprocesses are invoked until after the initialization is complete. In the
producer process, however, a \texttt{P} operation on \texttt{MUTEX} must be included to make sure that the consumer is not trying to manipulate the memory pool at the same time. A \texttt{V} operation after the critical section lets the consumer proceed, if it has been waiting.

\begin{verbatim}
READ (INPUTDATA);
PWAIT(MUTEX);
INDEX := 0;
LOOP
  .
  .
  .
END;
VSIGNAL(MUTEX);
\end{verbatim}

similar additions are also required in the consumer process to insure mutual exclusion.

\begin{verbatim}
LOOP
  PWAIT(MUTEX);
  .
  .
  .
  VSIGNAL(MUTEX)
  WHEN OUTPUTDATA=ENDSIGNAL EXIT
\end{verbatim}

Although the \texttt{MUTEX} semaphore will prevent the producer and consumer processes from manipulating the memory pool at the same time, one more restriction needs to be placed on each process. The producer should not be allowed to move an integer into memory unless an empty location is available. The consumer should not attempt to take an integer from memory unless one is there to be taken. Two general sema-
phores are needed to implement these restrictions. The semaphore AVAIL shows the number of empty memory locations, and the semaphore FULL shows the number of full memory locations. The producer must perform a P operation on AVAIL before moving the input data to memory. After moving the data, it must signal the presence of another full location by performing a V operation on FULL.

```plaintext
READ (INPUTDATA);
PWAIT (AVAIL);
PWAIT (MUTEX);
.
.
V SIGNAL (MUTEX);
V SIGNAL (FULL)
```

The inverse is true for the consumer, which must perform a P operation on FULL before attempting to take data from memory and then perform a V operation on AVAIL after taking the data.

```plaintext
LOOP
  PWAIT (FULL);
  PWAIT (MUTEX);
  .
  .
  .
  V SIGNAL (MUTEX);
  V SIGNAL (AVAIL);
```

Finally, the initial value for each semaphore must be established. Since AVAIL represents the number of empty memory locations, AVAIL should be set to the total number of
locations. FULL should be initialized to zero because there are no full locations at the start. MUTEX must be initialized to one. All the semaphores must be initialized before any processes are invoked, i.e., in the main program.

\[
\begin{align*}
\text{SET (AVAIL, NOBUFFERS);} \\
\text{SET (FULL, 0);} \\
\text{SET (MUTEX, 1);} \\
\end{align*}
\]

The producer/consumer program is now complete. We can be confident that it will perform its task regardless of the relative execution speeds of the two processes. It should also take maximum advantage of parallel execution. The complete listing of the producer/consumer program is given in Appendix B.

**SEQUENTIAL PROGRAMMING IN CONCUR**

Many tasks remain which do not lend themselves to concurrent programming. For them, the sequence of subtask performance is crucial. In such cases, when a subtask is begun, the main program must wait until the process completes before invoking the next subtask.

In many concurrent programming languages, such as MODULA and extended ALGOL for the Burroughs 6700, separate facilities exist for defining sequential procedures and concurrent processes. CONCUR, however, has only one method for process definition. A process so defined is always considered con-
A synchronization mechanism must be employed to force the main program to wait while the process executes. For this purpose, a binary semaphore will suffice. The main program follows each process call with a P operation on the binary semaphore. If the process is not complete, the P will cause the main program to wait. The main program will continue when the process performs a V operation on the semaphore. For synchronization to occur properly the binary semaphore must be initialized to zero.

Appendix B includes a listing of a sequential program. The program simply reads in a list of integers, inverts the list, and writes out the inverted list. The program uses a process to accomplish the inversion. Clearly this straightforward approach requires that the list be inverted before it is printed. Therefore, the main program must wait until the INVERT process is complete. The semaphore DONE is defined and initialized in the main program.

```
DONE:SEMA;
.
.
SET(DONE,0);
```

After calling the INVERT routine, the main program performs a P operation on DONE

```
INVERT(START,STOP);
PWAIT(DONE);
```
The INVERT process cooperates by performing a V operation on DONE when it ends.

```
VSIGNAL(DONE)
END INVERT;
```

In this way the main program will indeed wait until the INVERT process has finished, and the sequential execution required by the algorithm will be enforced.

Note that CONCUR can simulate the execution protocol of traditional languages but the inverse is not true. This shows that sequential programming is a proper subset of parallel programming.

THE CONCURRENT QUICKSORT EXAMPLE

C.A.R. Hoare has developed an internal sort algorithm called the "quicksort" (8). This algorithm basically partitions the array to be sorted around an arbitrarily-chosen pivot point so that all the elements in the array before the pivot point are less than or equal to the pivot element and all the elements in the array after the pivot point are greater than or equal to the pivot element. Once the array is partitioned in this fashion, the quicksort algorithm can be applied recursively to each of the partitions until all the segments to be sorted contain only one element. At this point the entire array has been sorted.
To implement this algorithm, a main program that controls input of the unsorted array and output of the sorted result needs only to invoke a QUICKSORT process and pass it the beginning and ending index for the array. The main program in this example reads in the array and counts the number of elements it reads. To satisfy an assumption for the partitioning procedure (discussed below), the main program appends to the array a trailer with a value that exceeds any of the actual array values. This trailer element is not included in the element count, however. The main program code follows:

```
CONST MAXSIZE=100; ENDSIG=9999;
VAR A:ARRAY MAXSIZE OF INTEGER;
    ELEMENTCNT:INTEGER;
    INDEX:INTEGER;
BEGIN (*MAIN PROGRAM*)
    ELEMENTCNT:=0;
    LOOP
        ELEMENTCNT:=ELEMENTCNT+1;
        READ (A(.ELEMENTCNT.))
        WHEN A(.ELEMENTCNT.)=ENDSIG EXIT
    END;
    ELEMENTCNT:=ELEMENTCNT-1;
    QUICKSORT(1,ELEMENTCNT);
    INDEX:=1;
    LOOP
        WRITE (A(.INDEX.));
        INDEX := INDEX+1
        WHEN INDEX > ELEMENTCNT EXIT
    END
END
```

After the quicksort process is started by main, the quicksort calls itself recursively until it passes itself a single element partition. To end the recursion, the quicksort
must first test whether it indeed has elements to sort; if not, it simply exists without calling itself again. If it has at least two elements, the quicksort invokes the partitioning procedure to divide the array. The quicksort then calls itself to sort the elements below the pivot point and calls itself again to sort the elements above the pivot point. The pivot element itself is in the appropriate position; therefore, it is not included in future sorts. Since the pivot point returned is chosen arbitrarily, one of the two resulting partitions may be empty. Thus, the quicksort must compare the pivot point to the partitioned segment's end points to avoid calling itself to sort any empty partition.

```
PROCESS QUICKSORT (M:INTEGER,N:INTEGER);
VAR I:INTEGER;
BEGIN (*QUICKSORT*)
  IF N > M THEN
    PARTITION (M,N+1);
    IF M < I THEN
      QUICKSORT (M,I-1)
    END;
    IF I < N THEN
      QUICKSORT (I+1,N)
    END
  END
END QUICKSORT;
```

The partitioning procedure assumes that the last element in the segment to be partitioned is greater than or equal to all other elements in the segment. Thus, initially, the main program appends a trailer element to the array; and, subsequently, the quicksort passes to the partitioning
procedure the index for the first element in the segment to be sorted and the index for the next element beyond the end of that segment.

There are many methods for selecting the pivot element for the partitioning procedure. The choice of pivot element can seriously influence the efficiency of the quicksort (9). The purpose of this example, however, is to explore the possible application of concurrent programming to the quicksort algorithm; therefore, the first element of the segment is chosen as the pivot element.

After selecting the pivot element, the partitioning procedure must exchange the position of this element with others in the segment until the pivot element is placed in its appropriate position. This position is the pivot point; the elements before it are less than or equal to the pivot element, and the elements after it are greater than or equal to the pivot element. Positioning of the pivot element is accomplished by the following algorithm: (1) the segment is scanned from the second element for an element that is greater than or equal to the pivot element; (2) the segment is scanned from the last element for an element that is less than or equal to the pivot element; (3) if the index of the element found in (1) is less than or equal to the index of the element found in (2), the elements are exchanged and the scanning continues; otherwise the pivot element is exchanged with the
element found in (2) and the partitioning procedure is complete. The partitioning procedure uses an exchange procedure to exchange specified elements. Its code is included with the partition code below.

```
PROCESS PARTITION (M:INTEGER,N:INTEGER);
    VAR X:INTEGER;
    PROCESS EXCHANGE (M:INTEGER,N:INTEGER);
        VAR TEMP:INTEGER;
        BEGIN (*EXCHANGE*)
            TEMP:=A(M.);
            A(M.):=A(M.);
            A(M.):=TEMP
        END EXCHANGE;
    BEGIN (*PARTITION*)
        X:=A(M.);
        I:=M;
        LOOP
            LOOP I:=I+1
                WHEN A(I.) >= X EXIT
            END;
            LOOP N:=N-1
                WHEN A(N.) <= X EXIT
            END
            WHEN I > N EXIT
        EXCHANGE (I,N)
        END;
    EXCHANGE (M,N);
    I:=N
END PARTITION;
```

The quicksort presents an excellent opportunity for concurrent programming. Although each invocation of the quicksort works on the array to be sorted, no two invocations work on the same segment or overlapping segments of the array. After the array has been partitioned, the quicksorts for the partitions of the array can proceed independently. Since all
processes in CONCUR are concurrent by default, the existing invocations of the quicksort already take advantage of this independence.

Within the quicksort, however, there are procedures which must be executed sequentially. For example, the quicksort cannot call itself recursively until the partitioning is complete. A semaphore must be declared in the quicksort to force it to wait while the partition procedure executes. The quicksort sets the semaphore to zero, invokes PARTITION, and then waits on the semaphore. The following code is added:

```
PARTDONE:SEMA;
.
.
SET (PARTDONE,0);
.
.
PWAIT(PARTDONE);
```

The partition procedure simply signals when it is complete.

```
VSignal(PARTDONE)
```

Within the partition procedure the exchange procedure must be employed sequentially. That is, further scanning of the segment or completion of the partition procedure cannot occur until an exchange is accomplished. The partition procedure must declare a semaphore on which to wait until an exchange is complete. It must set the semaphore to zero
initially and then wait on it after every call to EXCHANGE.

    EXCHDONE:SEMA;
    .
    .
    SET (EXCHDONE,0);
    .
    .
    PWAIT(EXCHDONE)
    .
    .
    PWAIT(EXCHDONE);

The exchange procedure signals when it is complete:

    VSIGNAL(EXCHDONE).

With the addition of these two semaphores most of the synchronization for the quicksort is complete. A challenging problem remains to be solved, however. Although it only invokes the quicksort process once, the main program must wait until all invocations of the quicksort process are complete. If it does not wait, it might print the array when it is only partially sorted. A simple semaphore, such as PARTDONE or EXCHDONE, is not sufficient. A mechanism must be added so that only the last quicksort process to complete will signal the main program.

In this example a counter is introduced to count the number of quicksort processes that have not completed. The main program and the quicksort itself must increment the
counter before each invocation of the quicksort. The quicksort process decrements the counter upon exiting. If the counter is decremented to zero, the process knows that it is the last quicksort and signals the main program to continue. The counter is declared in the main program along with the semaphore used to signal completion of the sort. The counter is initialized to one immediately before QUICKSORT is invoked. The semaphore is set to zero.

```
QUICKDONE:SEMA;
QUICKSIGS:INTEGER;

SET(QUICKDONE,0);
QUICKSIGS:=1;
```

In the quicksort process the counter is incremented before each invocation and decremented upon exiting the process. A count of zero prompts the quicksort to signal the main program via QUICKDONE.

```
QUICKSIGS:=QUICKSIGS+1;
QUICKSIGS:=QUICKSIGS+1;
QUICKSIGS:=QUICKSIGS-1;
IF QUICKSIGS <= 0 THEN \texttt{\textbackslash V\textbackslash \\textbackslash SIGNAL} (QUICKDONE) END
```
One last synchronization problem remains. Several copies of the quicksort may try to access the counter at the same time. To protect this critical resource, a mutual exclusion semaphore must be employed. The main program declares and initializes the semaphore.

```
CHECKSIGS:SEMA;
  :
  :
  SET (CHECKSIGS,1);
```

The main program does not need to use the semaphore when QUICKSIGS is initialized because no other processes have been invoked at that time. The quicksort process, however, must surround the increments, decrement, and test of the counter with a wait and signal on that semaphore.

```
PWAIT (CHECKSIGS);
QUICKSIGS := QUICKSIGS + 1;
VSIGNAL (CHECKSIGS);
  :
  :
  PWAIT (CHECKSIGS);
QUICKSIGS := QUICKSIGS + 1;
VSIGNAL (CHECKSIGS);
  :
  :
  PWAIT (CHECKSIGS);
QUICKSIGS := QUICKSIGS - 1;
IF QUICKSIGS <= 0 THEN VSIGNAL (QUICKDONE) END;
VSIGNAL (CHECKSIGS);
```

With the introduction of CHECKSIGS all the required synchronization mechanisms have been added to the quicksort program. A complete listing of the program as well as two sample runs are included in Appendix B.
THE IMPLEMENTATION OF CONCUR

OVERVIEW

The CONCUR compiler is a one-pass compiler written in PASCAL for the Xerox Sigma 9 computer. The compiler uses the technique of recursive descent. PASCAL was chosen as the implementation language because it is a well-structured, high-level language with recursion and facilities for programmer-defined data types. Recursive descent is a popular top-down parsing technique which can be readily programmed from a language's syntax graphs.

The compilation produces a hypothetical object language rather than Sigma 9 machine or assembly language. An interpreter is included with the compiler to execute the object code. This approach simplifies code generation since the compiler is not constrained by the hardware characteristics of the specific computer on which it is run. Indeed, such freedom improves the portability of the compiler. For this reason, many compiler writers such as Richards (10) and Waite (11) actually promote this approach for production compilers.
The scheduling algorithm used by the interpreter is a version of round robin. From the list of ready processes the interpreter sequentially selects the next process to be executed. The interpreter also generates a "random" number between 1 and 10 from the tens position of the current clock value to determine how many instructions will be performed within the selected process.

The CONCUR compiler is not presented as a "production compiler". Compiler options are almost non-existent, error indications are primitive, and error recovery is unimaginative. The compiler does, however, create a reasonable source code listing. In addition, a compilation trace, an object code list, and/or an execution trace can be produced by activating the appropriate output files.

A source listing of the CONCUR compiler is presented in Appendix C. Operating instructions for the compiler are given in Appendix D. Error numbers are listed and explained and a sample program with errors is shown in Appendix E.

THE INTERPRETER

The CONCUR interpreter executes the object code generated for the hypothetical machine. The machine consists of an accumulator, an index register, three stores (program store, descriptor store, and data store), and four pointers (a pointer to the current process being executed, a pointer to the current
instruction within that process, a pointer to the list of processes ready to execute, and a pointer to the list of free data store segments). This architecture is illustrated in Figure 1.

Figure 1
Hypothetical Machine Organization
Program store contains the machine instructions generated by the compiler. The interpreter is only permitted to read from this store. Descriptor store contains a descriptor for the main process and any other processes that have been invoked. A descriptor is a fixed-size data structure which contains the following information:

1. The current status of the descriptor or process (free, ready, waiting, or terminated).
2. The number of direct descendents of the process.
3. The static link to the process within which this process was defined.
4. The forward link to the next process in the ready or wait list.
5. The backward link to the preceding process in the ready or wait list.
6. The pointer to the beginning of this process's data area in data store.
7. The size of the data area.
8. The program counter.
9. The index register.
10. The accumulator.

The descriptors of all processes currently ready for execution are linked together in a READY list.

Data store is an array of integers which are allocated dynamically as processes are invoked and terminated. Within data store the integer data type is represented as a single
element; the semaphore data type is represented as a pair of adjacent elements; and the array data structure is represented as a group of contiguous elements. A FREE list is maintained for data store management. List manipulation and data store allocation are accomplished using versions of Knuth's first-fit and liberation algorithms (12). Since several executions of a process can proceed concurrently, each invocation of a process requires creation of a new process descriptor and allocation of a corresponding data store. In addition, because these process variables must be available to subordinate processes, the descriptor and data store cannot be released until all the subordinate processes, as well as the process itself, are terminated.

Dynamic storage allocation prevents the compiler from supplying absolute addresses in the object code. Thus, addresses of variables for the hypothetical machine are expressed as displacements from the beginning of some process data store. Since a process may refer to variables in its antecedents also, a level number must accompany each displacement to identify the specific data store to which this displacement applies. The actual starting address for the specific data store is obtained by following the static link chain up the appropriate number of levels.
THE MACHINE INSTRUCTION SET

The hypothetical machine responds to thirty-one instructions. These instructions were selected to complement the features of CONCUR. The instructions fall into five groups, each of which has a unique format.

In the descriptions below, ACC represents the accumulator, PC represents the program counter, XREG represents the index register, ADDR represents a data or program store address, and ( ) indicates its contents.

The format of Group 1 is OPCODE, INDEX FLAG, LEVEL, and DISPLACEMENT. If the index flag is set, the address represented by the level and displacement is modified by the contents of the index register. The following list gives the function of each instruction in Group 1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ACC ← ACC + (ADDR)</td>
</tr>
<tr>
<td>SUB</td>
<td>ACC ← ACC − (ADDR)</td>
</tr>
<tr>
<td>RSUB</td>
<td>ACC ← (ADDR) − ACC</td>
</tr>
<tr>
<td>MULT</td>
<td>ACC ← ACC * (ADDR)</td>
</tr>
<tr>
<td>DIV</td>
<td>ACC ← ACC ÷ (ADDR)</td>
</tr>
<tr>
<td>RDIV</td>
<td>ACC ← (ADDR) ÷ ACC</td>
</tr>
<tr>
<td>EQL</td>
<td>if ACC = (ADDR) then ACC ← 1 else ACC ← 0</td>
</tr>
<tr>
<td>NEQ</td>
<td>if ACC ≠ (ADDR) then ACC ← 1</td>
</tr>
</tbody>
</table>
else
  ACC ← 0

LSS
if ACC < (ADDR) then
  ACC ← 1
else
  ACC ← 0

LEQ
if ACC ≤ (ADDR) then
  ACC ← 1
else
  ACC ← 0

GTR
if ACC > (ADDR) then
  ACC ← 1
else
  ACC ← 0

GEQ
if ACC ≥ (ADDR) then
  ACC ← 1
else
  ACC ← 0

AND
ACC ← ACC ∧ (ADDR)

OR
ACC ← ACC ∨ (ADDR)

XOR
ACC ← ACC ∨ (ADDR)

LDX
XREG ← (ADDR)

STX
(ADDR) ← XREG

LDA
ACC ← (ADDR)

STA
(ADDR) ← ACC

PWAIT
(ADDR) ← (ADDR) - 1
if (ADDR) < 0 then
    the process waits
else
    it continues

VSIGNAL
    (ADDR) + (ADDR) + 1

if (ADDR) <= 0 then
    restart a waiting process
    and continue
else
    continue alone

It is crucial that instructions requiring two or more steps, especially PWAIT and VSIGNAL, are executed indivisibly. This requirement pertains to instructions in every group. TERM, DUMP, and CREATE are examples of other instructions that depend heavily on this requirement.

The format of Group 2 instructions is OPCODE and INSTRUCTION ADDRESS. The instruction address is simply the displacement from the start of the process code in program store. The function of each of the two instructions is as follows.

    BR        PC + ADDR
    BRF       if ACC = 0 then
                PC ← ADDR
Group 3 only contains one instruction. This instruction is a load immediate (LDI) with the format OPCODE and NUMERIC CONSTANT. The instruction loads the constant into the accumulator.

Group 4 contains the six instructions which require no operands. Thus, their format is simply OPCODE. The name and function of each is given below.

NOT ACC ← ACC
STAX XREG + ACC
TERM the process status is set to TERMINATED
the process is deleted from the READY list
the descendent count is decremented for each process linked to this one in the static link
the descriptor store and data store are liberated for any terminated process in the chain whose descendent count becomes 0
READ an integer is read into the ACC from M:SI
WRITE the contents of the ACC are printed on M:LO
DUMP the values for the four pointers are printed on the M:LO followed by each descriptor and its corresponding data store

Finally, the CREATE instruction is the sole occupant of Group 5. It has the format OPCODE, INITIAL PC, STATIC LINK, DATA STORE SIZE, PARAMETER COUNT, and PARAMETER PASS AREA START. The CREATE instruction creates a process descriptor, allocates the required data store, increments the descendent count of all processes related via the static link.
chain, and copies any parameters into the first part of the
data store segment.

**CODE GENERATION**

The relationship between CONCUR statements and the hypothetical machine instructions is relatively straightforward. The following examples illustrate this relationship.

The assignment statement has the general form:

```
variable := expression
```

The compiler must generate code first to evaluate the expression and then to store its result in the specified variable. For example:

```
COST := LIST-DISCOUNT becomes
STA TEMP
LDA LIST STA TEMP
LDA DISCOUNT } evaluating the expression
RSUB TEMP
STA COST } storing the result
```

Since the hypothetical machine is a single accumulator machine rather than a stack machine, the code generated, particularly for expression evaluation, makes frequent use of temporary variables. The compiler must keep track of the maximum number of temporary variables used by each process so that space for them may be allocated along with space for declared variables. Temporary variables occupy consecutive memory locations; that is, TEMP$_{i+1}$ is always allocated following
A process is invoked when its name is used in the statement part of a block.

In this case, if the process has parameters, the first code that is generated stores each parameter in a temporary storage location in the calling process. The actual invocation is accomplished with a CREATE instruction. For example:

\[
\text{POWER\ (VALUE-AVERAGE,2)} \text{ becomes}
\]

\[
\begin{align*}
\text{LDA} & \quad \text{VALUE} \\
\text{STA} & \quad \text{TEMP1} \\
\text{LDA} & \quad \text{AVERAGE} \\
\text{RSUB} & \quad \text{TEMP1} \\
\text{STA} & \quad \text{TEMP1} \\
\text{LDI} & \quad 2 \\
\text{STA} & \quad \text{TEMP2} \\
\text{CREATE} & \quad \text{POWER, <static link>, <data store size>, 2, TEMP1} \\
\end{align*}
\]

and \text{WAITLOOP} becomes \text{CREATE WAITLOOP, <static link>, <data store size>, 0, <null>}. 

Although only a simple END marks the end of a process, this particular END does prompt code generation. Specifically, a TERM instruction is generated to terminate the process. Thus, a process is invoked through a CREATE generated at its call and ended by a TERM which follows its last executable statement.
CONCUR'S IF statement is defined as:

The compiler generates code to evaluate the initial expression. A conditional branch must follow to select the appropriate statement sequence. When the ELSIF is employed, expression evaluation and conditional branching are required several times. For example:

```
IF CREDITHRS < 12 THEN
  TUITION := 82 * CREDITHRS
ELSIF CREDITHRS > 18 THEN
  TUITION := 966 + 82 * (CREDITHRS-18)
ELSE
  TUITION := 966
END
```

becomes

```
LDA CREDITHRS            \{ evaluating the first expression
STA TEMP1
LDI 12
GTR TEMP1
BRF ADDR1 \} conditional branch
LDI 82
STA TEMP1
```

```
  LDA CREDITHRS             \{ first statement sequence
  MULT TEMP1
  STA TUITION
  BR ADDR3
ADDRL LDA CREDITHRS        \{ evaluating the second expression
STA TEMP1
LDI 18
LSS TEMP1
```

46
Notice that the conditional branches require forward references to later parts of the IF statement. Since the CONCUR compiler is a one-pass compiler, a record must be kept of these references. A FIXUP routine is used to supply the appropriate addresses when they are available. This routine is also used for the LOOP statement and for process invocations encountered before the process is fully defined.

The LOOP statement's format is:

```
LOOP statement sequence
WHEN expression DO statement sequence EXIT statement sequence
END
```
This construct is a general purpose loop. Thus, it may require expression evaluation first or code for a statement sequence, depending on how it is used. As mentioned above, the LOOP generates forward references requiring FIXUP. Two forward references occur within the code for each WHEN clause.

For example:

```
LOOP
  WHEN CODE = 1
  | ADDR1 LDA CODE STA TEMP1 LDI 1 EQL TEMP1 BRF ADDR2 becomes conditional branch
  |
  | evaluating the first expression
  |
  | code for the first statement
  |
  | exit
  |
  | ADDR2 LDA CODE STA TEMP1 LDI 2 EQL TEMP1 BRF ADDR3 becomes conditional branch
  |
  | evaluating the second expression
  |
  | code for the second statement sequence
  |
  | exit
  |
  | ADDR3 LDA CODE STA TEMP1 LDI 3 EQL TEMP1 BRF ADDR4 becomes conditional branch
  |
  | evaluating the third expression
  |
  | code for the third statement sequence
  |
  | exit
  |
  | ADDR4 BR ADDR1 looping back
  |
  | ADDR5 end
```
Semaphore manipulation is accomplished through the SET, PWAIT, and VSignal statements. The PWAIT and VSignal instructions are defined as primitives in the hypothetical machine; thus, the CONCUR PWAIT and VSignal translate directly into their machine language counterparts. This insures that they are uninterrupted. The SET statement has no corresponding primitive, however. The format of the SET is

\[
\text{SET} \left( \text{ident.} , \text{expression} \right)
\]

The statement translates into the instructions required to evaluate and store the expression to which the semaphore is initialized. Finally, an additional load and store are required to initialize the waiting list for that semaphore. For example:

\[
\begin{align*}
\text{SET}(\text{MUTEX},1) & \quad \text{LDI} \ 1 \quad \text{\{initializing the semaphore value}\} \\
& \quad \text{STA} \ \text{MUTEX} \\
& \quad \text{LDI} \ 9999^* \quad \text{\{starting the waiting list empty}\} \\
& \quad \text{STA} \ \text{MUTEX+1}
\end{align*}
\]

*9999 is the value used to signal end-of-list.

The set statement is not indivisible. When correctly used, it will be executed before any processes depending on the semaphore are created. Thus, uninterrupted execution is not necessary.

Although machine instructions exist with names matching each of the CONCUR I/O statements (READ, WRITE, and DUMP), only the DUMP statement translates directly into its corresponding machine instruction. The READ and WRITE machine instructions
use the accumulator as the destination or source of the I/O operation. The READ and WRITE statement syntax shows that a more general destination or source may be specified in CONCUR itself.

Thus, for the READ statement the compiler must first generate a READ instruction and then include a STA to move the accumulator to the desired variable's address. For the WRITE statement the compiler generates code to evaluate the expression and then generates a WRITE instruction. For example:

\[
\text{READ(MATRIX(.K.)) becomes } \begin{array}{l}
\text{READ } \{\text{reading the value into ACC}} \\
\text{STA TEMP1 } \{\text{saving it}} \\
\text{LDA K} \\
\text{STAX } \{\text{evaluating the index expression}} \\
\text{LDA TEMP1} \\
\text{STA MATRIX(K) } \{\text{storing the input value}} \\
\end{array}
\]

and \text{WRITE(RATE*TIME) becomes } \begin{array}{l}
\text{LDA RATE} \\
\text{STA TEMP1} \\
\text{LDA TIME} \\
\text{MULT TEMP1} \\
\text{WRITE } \{\text{writing it out}} \\
\end{array}

The CONCUR compiler produces an object code list upon request. The JCL required to obtain this list is specified in Appendix D. Appendix F presents a complete CONCUR program source list followed by its corresponding object code. All of the examples in this section were verified through the compiler option.
CONCLUSION

This project has explored the syntax and run-time support features that can be supplied with a high-level language to permit concurrent programming. The minimal set of such facilities is the following:

1) the ability to define an asynchronous process,

2) a descriptor for an active process that maintains its status and data independent of other processes or other invocations of the same process,

3) the ability to create and destroy active processes, and

4) a mechanism for synchronizing processes when necessary.

The specific methods used in CONCUR to implement these facilities are not the only methods available. For example, Dijkstra extends ALGOL 60 with a construction he calls "a parallel compound" to permit definition of parallel processes (7,p.53). Bobrow and Wegbreit propose the use of a stack rather than dynamic storage allocation of linked blocks for maintenance of process descriptors (13). Tsichritzis and Bernstein discuss how processes can be created and destroyed and list a variety of synchronization mechanisms (1).
The synchronization feature appears to be the one most worthy of attention in CONCUR. The chosen implementation for the other features may lead to inefficiencies, but the use of semaphores for synchronization actually introduces significant potential for error. As noted earlier, the correct functioning of semaphores is particularly dependent on their proper initialization. Errors can also occur when the P and V operations on a semaphore are misplaced or omitted. Implementation of more tightly controlled synchronization, such as that included in Modula, would be a notable improvement.

The "interface module" of Modula is essentially a monitor, as developed by Brinch Hansen and Hoare (4). A monitor encompasses a critical resource and the procedures through which the resource can be accessed. Processes can only use the critical resource by invoking the appropriate procedure. The monitor guarantees mutual exclusion by allowing only one process to be active within it at one time. "Wait" and "signal" operations may be used by the monitor procedures to suspend and awaken processes executing within them. The monitor relieves concurrent processes of the duty of dealing directly with the synchronization primitives and removes the shared resource from direct, and thus possibly incorrect, manipulation by the processes. Each process needs only to call a procedure for the operation it requires. Thus, the risk of error is reduced.
A monitor feature could be added to CONCUR using its existing synchronization primitives -- semaphores and the SET, PWAIT, and V SIGNAL commands. First, to insure that the procedures within the monitor actually acted as procedures rather than processes, CONCUR would be required to generate a semaphore which the calling process would set to zero before the call and wait on immediately after the call. Each procedure within the module would need to end by signalling its caller via that semaphore. Second, to guarantee that only one process had access to the monitor at one time, CONCUR would need to associate a mutual exclusion semaphore with the monitor. The semaphore would be initialized to one. Then each procedure within the monitor would begin by performing a P operation on the mutual exclusion semaphore and end by performing a V operation on it.

Finally, CONCUR would be required to translate the monitor "wait" and "signal" operations into corresponding CONCUR operations. Monitor waits and signals do not translate directly into semaphore waits and signals. The former involve a "condition variable" which has no memory associated with it. That is, if a signal is given for a condition and no process is waiting for the signal, the signal will be lost; in contrast, a semaphore's value holds a record of every operation performed on the semaphore. Also, when a process waiting on
a semaphore is signalled, both it and the signalling process continue. Since only one process may use the monitor at one time, the process signalling the condition must pause until the process it awakened exits the monitor or is suspended again.

To insure that only the awakened process can continue and to give the signalling process higher priority than other processes waiting to enter the monitor, a second semaphore must be used to handle signalling processes. Hoare calls this semaphore "urgent" (14,p.551). It is initialized to zero. Associated with the semaphore is an "urgentcount" that shows how many signalling processes are currently suspended. To honor the higher priority of signalling processes, the exit from each procedure can no longer be a simple P operation on the monitor's mutual exclusion semaphore. Instead, the urgentcount must be tested. If a signalling process is waiting, the procedure exits with a V on the urgent semaphore to let the signalling process continue; otherwise it exits with the V on the mutual exclusion semaphore, and other processes may enter the monitor.

Condition variables and monitor waits and signals are further implemented by introducing a semaphore and a counter for each condition variable. Both are initialized to zero. The counter indicates the number of processes waiting on a condition. The semaphore actually suspends and awakens a
process for that condition. A wait operation can be stated in CONCUR as

```
COUNTER := COUNTER + 1;
IF URGENTCOUNT > 0 THEN
  VSIGNAL(URGENT)
ELSE
  VSIGNAL(MUTEX)
END;
PWAIT(CONDSEM);
COUNTER := COUNTER - 1;
```

The signal operation can be written as

```
URGENTCOUNT := URGENTCOUNT + 1;
IF COUNTER > 0 THEN
  VSIGNAL(CONDSEM);
  PWAIT(URGENT)
END;
URGENTCOUNT := URGENTCOUNT - 1;
```

Thus, it appears that more sophisticated and reliable synchronization facilities can be built upon primitives already implemented in CONCUR. After this improvement is made, however, CONCUR would still need additional modification to bring it to production level. Some of these modifications are discussed in the areas of interest listed below.

The main purpose of this thesis project was to increase understanding of language design, compiler writing, and selected concepts in operating systems and machine architecture as well as to explore the implications of concurrent programming methodology. Further areas of investigation are prompted by this thesis.
1) Additional language features. Many possible extensions become quickly evident when CONCUR is compared with another high-level language such as PASCAL. For example, there is an obvious need for more data types and data structures. Also, more control structures could be added. The CASE control structure would eliminate the obscure use of the LOOP construct for case testing. The ability to pass arrays, and perhaps even semaphores, as parameters and to pass specified parameters by reference rather than by value could prove helpful. The ability to declare a procedure instead of a process would free the programmer from responsibility for introducing and manipulating a semaphore for each procedure. Finally, more sophisticated I/O functions are clearly needed. (The inability to print headings or at least force a page break frustrated any attempt to print the unsorted list as well as the sorted one produced by the quicksort program in Appendix B.)

2) Improved compiler characteristics. For example, the inclusion of options to control listing, summaries, and execution would be
convenient. These options could encompass such things as suppressing the listing, forcing page breaks in the listing, permitting starts and stops for tracing execution, and allowing redirection of input data and output so that they were not tied to the source input or list output devices specified. The compiler's error handling could also be improved. Error checking could be introduced to decrease the potential for semaphore abuse. Better diagnostic messages could be provided, and more clever recovery techniques could be employed. Finally, some code optimization would be welcome. Unnecessary use of temporary variables should be one of the first inefficiencies addressed.

3) More sophisticated operating systems. For example, the round robin scheduling algorithm could be replaced by an algorithm that is more sensitive to the actual mix of ready processes. For manipulating wait queues, other techniques in addition to the FIFO approach could be explored. Other methods of storage allocation could be used.
4) More complex machine architecture. Although the hypothetical machine has primitives that support CONCUR's parallel processing features very well, it lacks comparable support for more common language features. A stack-oriented architecture could well be more effective than the single accumulator, single register organization selected.

5) More rigorous programming methodologies. Specifically, the quicksort program indicates a need to consider the implications of recursive programming within a concurrent environment. In addition, the work of Susan Owicki (15), David Gries (15,16), and others investigating correctness proofs for parallel programs deserves attention.

With the focus of hardware and software development turning to parallelism to increase the performance of computer systems in spite of physical limitations, these topics and others should prove to be of great interest.
APPENDIX A
CONCUR SYNTAX GRAPHS

Syntax graphs are used by Niklaus Wirth to represent the syntax of a programming language. In his book, Algorithms + Data Structures = Programs, he provides a thorough explanation of the rules for constructing a graph as well as the rules for translating a graph into a program (17).

PROGRAM

```
block
```

BLOCK

```
CONST ident = numeric constant
VAR ident : INTEGER
VAR ident : SEMA
VAR ident : ARRAY constant OF INTEGER
VAR PROCESS ident (ident ; INTEGER ) block ident
BEGIN statement sequence END
```

STATEMENT SEQUENCE

```
statement ;
```
APPENDIX B

SAMPLE PROGRAMS

PRODUCER/CONSUMER PROGRAM

A SEQUENTIAL PROGRAM

CONCURRENT QUICKSORT PROGRAM
PRODUCER/CONSUMER PROGRAM
When INDEX = NOBODY'S EXIT

BEGIN (MAIN PROGRAM)

END.
0 ERRORS FOUND.

END

CONSUMER

PRODUCER

END

WHEN INDEX = NOBUFFERS EXIT

BUFFERIF(INDEX) = 0

INDEX = INDEX + 1

LOOP

QUEUEEGIN = EMPTY

INDEX = 0

SET (INDEX,1)

SET (NULL,0)

} SET (AVAIL,NOBUFFERS)

BEGIN (* MAIN PROGRAM *)

*
A SEQUENTIAL PROGRAM
```
END PROCESS (INTEREST, STOP: INTEREST)

PROCEDURE (INTEREST, INTEREST, STOP: INTEREST)

FOR I = 1 TO SIZE
    WHEN START < STOP EXIT
    LOOP
        START = START + 1
        WHEN START < STOP NEXT
        IF
    END IF
END LOOP
```

CONCURRENT QUICKSORT PROGRAM
```

```
VAR TEMP: INTEGER;

(* EXCHANGES THE HI AND NHI ELEMENTS OF THE ARRAY. *)

* THIRD LEVEL PROCESS

PROCEDURE EXCHANGE (HI: INTEGER, N: INTEGER);

VAR: X: INTEGER;

BEGIN (EXCHANGE)

SIGNAL (EXCHANGE);

VAR N: INTEGER;

BEGIN (PARTITION)

PROCEDURE EXCHANGE (HI: INTEGER, N: INTEGER);

VAR X: INTEGER;

BEGIN (EXCHANGE)

FIND PARTITION (M: INTEGER, N: INTEGER);

VAR X: INTEGER;

BEGIN (FIRST LEVEL PROCESS)

PROCEDURE QUICKSORT (M: INTEGER, N: INTEGER);

VAR X: INTEGER;

BEGIN (QUICKSORT)

(* SCHEM FOR MAIN. EXECUTE FOR QUICKSORTS *)

(* OF QUICKSORTS EXPECTED TO SIGNAL (*)

(* SCHEM FOR DELAY UNTIL QUICKSORT ENDS *)

(* INDEX FOR QUICKSORT LISTS *)

(* ELEMENTS IN QUICKSORT LISTS *)

(* THE LISTS TO BE SORTED *)

VAR A: ARRAY MAXSIZE OF INTEGER;

CONST MAXSIZE = 100; MAXSIZE = 999;

WRITE: THE SQUEEZE LIST,

A MODIFICATION OF THE QUICKSORT ALGORITHM PROPOSED BY PHILK AND THEN

A CONCISE QUICKSORT ALGORITHM READS IN A LIST OF NUMBERS, SORTS THEM USING

COURIER COMPILER VERSION 4 JANUARY 1978

09:30 APR 20, 78
FUNCTION (PROGRAM,)
BEGIN (PARTITION)
VAR TEMP: INTEGER;
VAR TEMPS: ARRAY [1..M] OF INTEGER;
BEGIN (EXCHANGE)
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARRAY [1..M] OF INTEGER
EXCHANGE: BEGIN
VAR X: INTEGER
AND FOR J > I X(J,J) = X(J'I)
FOR J < I X(J,J') = X(J'I)
PARTITION: BEGIN
IF TEMP = V'M
THEN
BEGIN (EXCHANGE)
VAR TEMPS: ARR
0 ERRORS FOUND.
END

114 i END
113 i

WHEN INDEX < ELEMENT EXIT
112 i INDEX = INDEX + 1
111 i WRITE (A, INDEX)
110 i LOOP
109 i INDEX = 1
108 i

08:41 04/29/78 06:00PM...444 A C 6 44
APPENDIX C

SOURCE LISTING OF CONCUR COMPILER
CASE KINOCO (OBJECT OF (IDENTIFIER) = ALPHAB); NAME = ALLPH; ENTRY = [LINK]; IDENTIFIER = [ARRAY]; LENGTH = INTERVAL; ARRAY = [PROCESS]; START = ALPH; END = [PROCESS]; \nCASE KINOCO: OBJECT OF (IDENTIFIER) = ALPHAB; NAME = ALLPH; ENTRY = [LINK]; IDENTIFIER = [ARRAY]; LENGTH = INTERVAL; ARRAY = [PROCESS]; START = ALPH; END = [PROCESS]; \nCASE KINOCO: OBJECT OF (IDENTIFIER) = ALPHAB; NAME = ALLPH; ENTRY = [LINK]; IDENTIFIER = [ARRAY]; LENGTH = INTERVAL; ARRAY = [PROCESS]; START = ALPH; END = [PROCESS]; \nCASE KINOCO: OBJECT OF (IDENTIFIER) = ALPHAB; NAME = ALLPH; ENTRY = [LINK]; IDENTIFIER = [ARRAY]; LENGTH = INTERVAL; ARRAY = [PROCESS]; START = ALPH; END = [PROCESS]; \nCASE KINOCO: OBJECT OF (IDENTIFIER) = ALPHAB; NAME = ALLPH; ENTRY = [LINK]; IDENTIFIER = [ARRAY]; LENGTH = INTERVAL; ARRAY = [PROCESS]; START = ALPH; END = [PROCESS];
procedure getnextchar;

(* ****************
* PROGGRAM INCOMPLETE!
* *****************)

procedure getnextchar;

begin

end.

end.

end.

end.

end.
```pascal
procedure To Pass A Constant Declaration;

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLARATION*)

(*PROCEDURE CONSTANT to DECLAR
END (PROCEDURE);}
END
RELEASE TEMP;
BEGIN
GETLASTA, +:num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
GETLASTA, +num didn't have.'+0.01!;
```
BEGIN
IF SIMULA = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
ELSE
IF SYMBOL = RAPAREN THEN GERMASWALD ELSE EPRAO(22)
END
BEGIN
IF SIMULA = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
ELSE
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END

BEGIN
IF SYMBOL = COMMA THEN EPRAO(14)
END
ELSE
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
BEGIN
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
ELSE
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
BEGIN
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
ELSE
IF SYMBOL = COMMA THEN GERMASWALD ELSE EPRAO(22)
END
```

PROCEDURE INSERT(PROPERTY: INTEGER; VAR LISTHEAD: INTEGER); PROCEDURE INSERT A DESCRIPTOR INTO A LIST.

PROCEDURE DELETE(PROPERTY: INTEGER; VAR LISTHEAD: INTEGER); PROCEDURE DELETE A DESCRIPTOR FROM A LIST.
BEGIN
  IF j > 0 THEN
    END = ARRISTANT[I]." I = 1
  ELSE
    BEGIN
      WHILE LISTA < 0 DO
        B equivalence = FREE
      END
      END = ARRISTANT[I]." I = 1
  END
END
| 2e | 0x5 | -  |
| 2f | 0x5 | -  |
| 2g | 0x5 | ALLOCATE |
APPENDIX D

COMPILER OPERATING INSTRUCTIONS

Input to the compiler is read from M:SI. The source listing and execution output are produced on M:LØ. The compiler trace is produced on F:00. The object code is listed on F:01. The execution trace is produced on F:02. To run the compiler, these files must be assigned. Unwanted output can be avoided by assigning the file to NØ.

EXECUTION VIA TIME-SHARING:

1. Assign the files using the SET command. For example:
   
   !SET M:SI/EXPR0G1   (source code is in a data file called EXPROG1)
   !SET M:LØ ME       (listing and execution output will appear on the terminal)
   !SET F:00 NØ       (compilation trace suppressed)
   !SET F:01 ME       (object code will be listed on the terminal)
   !SET F:02 NØ       (execution trace suppressed)

2. Run the compiler.

   !CONCUR.60764DYT

3. The compilation will begin with the following message.

   #PASCAL: START COMPILER
EXECUTION VIA BATCH:

1. Assign the files using the ASSIGN command. For example:

   !ASSIGN M:SI(FILE,EXPR0G1) (as above..except that output will appear on the line printer)
   !ASSIGN M:LØ(DEVICE,LP),(VFC)
   !ASSIGN F:00(DEVICE,NØ)
   !ASSIGN F:01(DEVICE,LP),(VFC)
   !ASSIGN F:02(DEVICE,NØ)

2. Run the compiler.

   !RUN(LMN,CONCUR,60764DYT)
When it detects an error in the source program, the compiler signals the error by skipping to a new line and printing a question mark beneath the offending symbol. The question mark is followed by a number indicating the type of error that has occurred. The compiler counts the errors detected and prints this count at the end of the source listing. Any compiler-detected error is fatal; that is, if the error count is non-zero at the end of compilation, execution is suppressed. The errors which the compiler detects are listed below with their corresponding numbers. A sample program with errors follows.

1. Use = instead of :=?
2. Constant expected.
3. Symbol = expected.
4. Identifier expected.
5. Semicolon expected.
6. Process name expected.
7. Integer expected...parameters must be integer.
8. END expected.
10. Variable expected.
11. Undeclared identifier.
12. Assignment to constant, process, or semaphore not allowed.
14. Arrays must be integer.
15. OF expected.
16. THEN expected.
17. The preceding factor cannot be followed by this symbol.
18. DO or EXIT expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain process identifier.
22. Right parenthesis expected.
23. Left parenthesis expected.
24. An expression cannot begin with this symbol.
25. Semaphore expected.
26. Subscript expected...neither process nor semaphore allowed.
27. Number expected.
28. Left bracket expected.
29. Right bracket expected.
30. Number is too large.
31. Variable must be INTEGER, SEMA, or ARRAY.
32. Colon expected.
33. Block expected...beginning with CONST, VAR, PROCESS, or BEGIN.
34. Comma expected.
*Pascal: Normal END-RUN
EXECUTION SUPPRESSED
* 4 ERRORS LUNO.
END

CONSUMER

WHEN INDEX = NOBUFFERS EXIT
 иф BUFFERTR(INDEX) = 0
    INDEX = INDEX + 1;
THE LOOP
  IF BUFFERTR(INDEX) = EMPT;
    SET (WAIT, NOBUFFERS);
  FI
  FI
END CONSUMER

END PROGRAM

WRITE (OUTPUTOUTA)

WHEN OUTPUTOUTA = END SIGNAL EXIT
SIGNAL(AVAL);
APPENDIX F

SOURCE PROGRAM
WITH OBJECT CODE LISTING
BEGIN
START = START + 1
WHILE (ARRA[N][ARRAY][START])
END LOOP
WHEN START < STOP EXIT
ENDDONE(START, STOP)
END LOOP
READ (ARRAY, START) = : INDATA
STOP = START + 1
WHEN INDATA = ENDING EXIT
READ (ARRAY, START) = : INDATA
END LOOP
STOP = 0!
START = 1!
IF (DONE, 0)!
BEGIN
END SIGNAL
BEGIN
STOP = STOP - 1!
START = START + 1!
READ (ARRAY, START) = : TEMP!
ARRAY (ARRAY, START) = : ARRAY!
TEMP = ARRAY (START)!
WHEN STOP => START EXIT
BEGIN
ENDDONE
VAR TEMP: INTEGER
PROCEDURE INDENT (START: INTEGER, STOP: INTEGER)
BEGIN
LOCAL: INTEGER
ARRAY (ARRAY, ARRAY A: SIZE OF INTEGER)
VAR INTEGER: INTEGER
CONST A SIZE = 104 ENDIND = 9999!
(SAMPLE CONCUK PROGRAM USING SEMAPHORE TO MAKE PROCEDURE.
COPY EXPLORE, 4060 PETUL)
0 ERRORS FOUND.

41 | END
40 | END
39 | START = START + 1
38 | WHILE (ARRAY[START] > START)
37 | WHEN START = STOP EXIT LOOP
36 | FINISH
35 | IF ARRAY[START] = FINISH then
34 | ARRAY[START] = FINISH
33 | ENDIF
32 | ARRAY[START] = START + 1
31 | WHEN FINISH = ENDSIZE EXIT LOOP
30 | BEGIN
29 | IF DONE
28 | IF START = FINISH then
27 | ARRAY[START] = ARRAY[START] + 1
26 | ELSE
25 | IF START = STOP then
24 | ARRAY[START] = ARRAY[START] - 1
23 | ELSE
22 | NO INVERT
21 | VISION(DONE)
20 | END
19 | IF STOP = START - 1 then
18 | ARRAY[STOP] = ARRAY[STOP] - 1
17 | ELSE
16 | ARRAY[STOP] = ARRAY[STOP] + 1
15 | WHEN STOP = START EXIT LOOP
14 | BEGIN
13 | VAR TEXT: INTEG
12 | PROCESS INVERT(START, STOP, INTEGER)?
11 | DO NOT SELL
10 | START: INTEGER
9 | STOP: INTEGER
8 | ARRAY[START]: ANY SIZE OF INTEGER
7 | VAR INTEGER: INTEGER
6 | COAST SIZE = 107 ENDING = 9999
5 | BEGIN
4 | WHEN ENDSIZE = 107 EXIT LOOP
3 | (Sample C64 Compiler Program Using Semaphore To Make Proccess Like Procedure.)
2 | ...
1 | OPTIONS: LOJ

C64 Compiler Version 1 JANUARY 2, 79
<table>
<thead>
<tr>
<th>Index</th>
<th>Operation</th>
<th>OPM</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>RED</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>0</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>6666</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>6666</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**GENERATED OPCODE**
...
REFERENCES


