A PROGRAMMING LANGUAGE FOR
CONCURRENT PROCESSING

APPROVED:

Dan W. Scott
Major Professor

Minor Professor

Dan W. Scott
Chairman, Department of Computer Sciences

Robert S. Touloukian
Dean of the Graduate School
This thesis is a proposed solution to the problem of including an effective interrupt mechanism in the set of concurrent-processing primitives of a block-structured programming language or system. The proposed solution is presented in the form of a programming language definition and model. The language is called TRIPLE.

The major difference between TRIPLE and other similar languages is the ability of the user to create and describe pseudo-processors to which he can assign one or more processes. A processor is a conceptual machine to which some user-specified number of processes can be assigned. A processor can have a user-defined interrupt structure. The key feature of the interrupt mechanism is that the user specifies interruption of a processor rather than a process, as in many other language designs. The interruption of a processor suspends all processes at a lower level than the interrupt until the interrupt has been processed or is momentarily blocked. The process interrupt of other systems and languages is a subset of the processor interrupt. This more
highly structured interrupt mechanism avoids several of the problems frequently found in other language designs.

This thesis is organized into four chapters. The first chapter provides the background material necessary to place the problem and its proposed solution in the proper perspective. The second chapter contains a formal definition of the syntax of TRIPLE along with an introduction to the semantics. The third chapter contains the design of a semantic model and the definition of the semantics of TRIPLE, using the model. The last chapter is a summary of the areas in which the design of TRIPLE contributes to the knowledge of concurrent-processing programming language design.
A PROGRAMMING LANGUAGE FOR
CONCURRENT PROCESSING

THESIS

Presented to the Graduate Council of the
North Texas State University in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Portia M. Jackson, B. S.
Denton, Texas
August, 1972
This thesis is the definition of a set of concurrent-processing primitives which incorporates an effective interrupt mechanism. The concurrent-processing primitives are developed in the context of a block-structured higher-level programming language called TRIPLE.

It is assumed that the reader has a knowledge of block-structured programming languages and concurrent processing. Some introductory material is included in Chapter I to insure a proper vocabulary and to provide the proper perspective for viewing TRIPLE.

Chapter II contains the formal definition of the syntax along with an informal discussion of the meaning of each type of syntactic construct. Chapter III contains the design of a semantic model for TRIPLE and the definition of the semantics of TRIPLE, using the model. A summary of the areas in which the design of TRIPLE contributes to the knowledge of concurrent-processing programming language design is contained in Chapter IV.
TABLE OF CONTENTS

LIST OF ILLUSTRATIONS ........................................... v

Chapter

I. INTRODUCTION ................................................... 1
   Concurrent Processing
   Block-Structured Environments
   Interrupt Mechanisms
   TRIPLE, A Mini-Language with an
   Effective Interrupt Mechanism

II. A DESCRIPTION OF TRIPLE ..................................... 7
   Basic Language Elements,
      Variables and Expressions
   Declaration Statements
   Basic Executable Statements
   Procedure, End, and Call Statements
   Process, Interrupt, and Resume Statements
   Request and Release Statements

III. A SEMANTIC MODEL OF TRIPLE ............................... 32
   The Basic Elements of the
      TRIPLE Contour Model
   Models of Numbers, Semaphores, Processors, and Arrays
   Models of Statement and Procedure Lables
   Models of Procedure and Call Statements
   The Process-Statement Model
   The Interrupt-Statement Model
   The Resume-Statement Model
   The End-Statement Model
   Models of Request and Release Statements

IV. CONCLUSION .................................................... 86

BIBLIOGRAPHY ..................................................... 90
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A TRIPLE Program</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>Partial Program Containing Example Goto Statements</td>
<td>20</td>
</tr>
<tr>
<td>3.</td>
<td>A Program Showing the Method of Procedure Definition</td>
<td>23</td>
</tr>
<tr>
<td>4.</td>
<td>The Algorithm Component</td>
<td>33</td>
</tr>
<tr>
<td>5.</td>
<td>A Record of Execution with n Snapshots</td>
<td>34</td>
</tr>
<tr>
<td>6.</td>
<td>A Ready Process</td>
<td>35</td>
</tr>
<tr>
<td>7.</td>
<td>A Blocked Process</td>
<td>35</td>
</tr>
<tr>
<td>8.</td>
<td>A Contour</td>
<td>36</td>
</tr>
<tr>
<td>9.</td>
<td>Two Contours Associated with a Procedure</td>
<td>37</td>
</tr>
<tr>
<td>10.</td>
<td>Example of Nested Contours</td>
<td>37</td>
</tr>
<tr>
<td>11.</td>
<td>Data Representation in a Contour</td>
<td>38</td>
</tr>
<tr>
<td>12.</td>
<td>The Model of a Contour Containing Two Numbers</td>
<td>39</td>
</tr>
<tr>
<td>13.</td>
<td>A Semaphore with an Empty Pointer</td>
<td>40</td>
</tr>
<tr>
<td>14.</td>
<td>A Semaphore Queue Containing Two Processes</td>
<td>40</td>
</tr>
<tr>
<td>15.</td>
<td>The Model of a Processor P</td>
<td>41</td>
</tr>
<tr>
<td>16.</td>
<td>Model of a Processor having No Started Processes</td>
<td>42</td>
</tr>
<tr>
<td>17.</td>
<td>Model of a Processor having Three Started Processes</td>
<td>42</td>
</tr>
<tr>
<td>18.</td>
<td>The Model of a Process on Processor R Queued Waiting for a Process Header in Processor P</td>
<td>43</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>19. A Processor with No Interrupt Structure</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>20. A Processor with Two Interrupt Levels</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>21. A Processor with Two Active Interrupts</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>22. A Processor with an Active Interrupt and Pending Interrupts on the Same Level</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>23. Algorithm for the Control Component of a Processor</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>24. A Partial Snapshot Showing a Process That Advanced Since the Last Snapshot</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>25. A Partial Snapshot Showing Two Processes That Advanced Since the Last Snapshot</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>26. An Array of Semaphores</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>27. A Statement Label and a Processor Label</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>28. A Contour A Containing Two Contours Created by Two Distinct Entrances to Procedure B</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>29. Dynamic Links</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>30. Partial Computation Showing the Effect of Executing a Call Statement</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>31. Skeleton Contour Showing the Location of Dummy Parameters</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>32. Models of Parameters Passed by Location and by Value</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>33. Partial Computation Showing the Effect of Executing a Process Statement Referencing a Processor with no Available Process Headers</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>34. Partial Computation Showing the Effect of Executing a Process Statement Causing a Process to be Started</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>35. Creation of Coroutines with the Process Statement</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>36</td>
<td>Algorithm Showing Interrupt Statement Execution</td>
<td>62</td>
</tr>
<tr>
<td>37</td>
<td>Snapshot 1 of a Computation using the Algorithm in Figure 35</td>
<td>63</td>
</tr>
<tr>
<td>38</td>
<td>Snapshot 2 of a Computation using the Algorithm in Figure 35</td>
<td>64</td>
</tr>
<tr>
<td>39</td>
<td>Snapshot 3 of a Computation using the Algorithm in Figure 35</td>
<td>65</td>
</tr>
<tr>
<td>40</td>
<td>Partial Computation Showing the Result of Executing a Resume Statement</td>
<td>67</td>
</tr>
<tr>
<td>41</td>
<td>Partial Computation Showing the Effect of End Statement Execution at a Time when the Contour Cannot be Deallocated</td>
<td>69</td>
</tr>
<tr>
<td>42</td>
<td>Partial Computation Showing the Effect of a Computation End</td>
<td>71</td>
</tr>
<tr>
<td>43</td>
<td>Partial Computation Showing the Effect of a Call End</td>
<td>72</td>
</tr>
<tr>
<td>44</td>
<td>An Algorithm to be used in Demonstrating the Execution of a Process End</td>
<td>74</td>
</tr>
<tr>
<td>45</td>
<td>Snapshot 1 from a Computation using the Algorithm in Figure 43</td>
<td>74</td>
</tr>
<tr>
<td>46</td>
<td>Snapshot 2 from a Computation using the Algorithm in Figure 43</td>
<td>75</td>
</tr>
<tr>
<td>47</td>
<td>Snapshot 3 from a Computation using the Algorithm in Figure 43</td>
<td>76</td>
</tr>
<tr>
<td>48</td>
<td>Snapshot 4 from a Computation using the Algorithm in Figure 43</td>
<td>77</td>
</tr>
<tr>
<td>49</td>
<td>An Algorithm to be used in Illustrating the Coroutine End</td>
<td>78</td>
</tr>
<tr>
<td>50</td>
<td>Snapshots Before and After the Execution of a Coroutine End using the Algorithm in Figure 48</td>
<td>79</td>
</tr>
<tr>
<td>51</td>
<td>Partial Computation Showing the Effect of an Interrupt End</td>
<td>80</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>52.</td>
<td>The Execution of a Request Statement Resulting in a Positive or Zero Semaphore</td>
<td>81</td>
</tr>
<tr>
<td>53.</td>
<td>The Execution of a Request Statement Resulting in a Negative Semaphore</td>
<td>82</td>
</tr>
<tr>
<td>54.</td>
<td>The Execution of a Release Statement Resulting in a Positive Semaphore</td>
<td>83</td>
</tr>
<tr>
<td>55.</td>
<td>The Execution of a Release Statement Resulting in a Zero or Negative Semaphore</td>
<td>84</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

This thesis is a proposed solution to the problem of including an effective interrupt mechanism in the set of concurrent-processing primitives of a block-structured programming language or system. The proposed solution is presented in the form of a programming language definition.

Concurrent Processing

A computer system is said to be capable of concurrent processing if it can simultaneously manage several activities or processes, causing them to advance towards completion. Denning (3) has stated that concurrency is a feature common to third-generation systems.

In a system that encourages concurrency, a computation can consist of several processes, a different number from one moment to the next. It is desirable to permit the creation and deletion of processes. The processes comprising a computation must be able to communicate with one another. This communication usually takes the form of common data storage and a group of system intrinsics. These communication intrinsics, together with a process creation and deletion facility, are commonly called the set of concurrent-processing primitives of a language or system.
Block-Structured Environments

This thesis is restricted to the study of concurrent processing in block-structured environments. Even with this restriction, however, the results are broadly applicable, since many current programming languages and computer systems are block-structured. Examples of block-structured languages are FL/1 (5) and ALGCL (7). An example of a block-structured computer system is the Burroughs B6700 (8).

Interrupt Mechanisms

It is generally agreed that it is desirable to include an interrupt mechanism among the concurrent-processing primitives of block-structured languages. It is not, however, agreed how this should be done. A full interrupt capability should make it possible for one process to interrupt another with consistent results. Consistent results can be obtained only if the interrupted process can continue its execution just as if the interrupt had not occurred. This capability has been provided to varying extents in several languages and systems.

One of the most common problems in current interrupt mechanisms is interrupting a momentarily blocked process and placing it in a consistent state after the interrupt has been processed. Organick (8) points to this as one of the major failures cited by the Burroughs designers of the B6700 computer system. Berry (1) was not able to solve this problem.
in the design of Oregano, a block-structured programming language. Other problem areas in the design of interrupt mechanisms in block-structured environments are

1. the passing of parameters to an interrupt processing routine,
2. the retention of the interrupt processing routine's full environment, and
3. the lack of a facility for allowing the interrupt processing routine to delay its processing until the process it interrupted has passed a critical section.

TRIPLE, A Mini-Language with an Effective Interrupt Mechanism

Ledgard (6) coined the word "mini-language" to identify a programming language that is designed specifically to study some particular aspect of programming languages in general. TRIPLE is a mini-language designed to permit the careful examination of concurrent-processing primitives, including an effective interrupt mechanism.

Those features of TRIPLE that are not relevant to concurrent processing have been kept simple. The following list of features gives examples of this simplicity:

1. The only standard data type is the integer, which can be used for arithmetic and logic,
2. The goto statement must specify a local label.
3. The conditional statement has no else phrase and may not have another conditional statement in the then phrase.

4. Input and output facilities barely exist.

5. A procedure may not return a value associated with the procedure name, as is done by a function.

In the area of concurrent processing TRIPLE has drawn freely on other language designs, modifying them as necessary to fit them in the context of this study. The following features are similar to those in other languages:

1. The semaphore with P and V operations are due to Dijkstra (4); however, the names RELEASE and REQUEST are due to Betourne (2).

2. The departure from the stack model of memory management and the use of a retention strategy are due to Berry (1).

The major difference between TRIPLE and other languages is that a processor is a declared data item. A processor is a conceptual machine to which some user-specified number of processes can be assigned. A processor can have a user-specified number of interrupt levels. This structure permits the design of an interrupt mechanism without the problems of those mentioned earlier.

Figure 1 is a sample TRIPLE program. Chapters II and III will provide the description of TRIPLE necessary to interpret the program.
OS: PROCEDURE:
  DECLARE P PROCESSOR WITH INTERRUPTS(READER,WRITER);
  PROCESS MAIN USING P;
MAIN: PROCEDURE:
  DECLARE BUFFER1(100) NUMBER;
  DECLARE BUFFER2(100) NUMBER;
  DECLARE S1 SEMAPHORE;
  DECLARE S2 SEMAPHORE;
  DECLARE GRIND PROCESSOR;
  DECLARE CRUNCH PROCESSOR;
NEXT:
  INTERRUPT FOR READER(BUFFER1, S1);
  INTERRUPT FOR READER(BUFFER2, S2);
  PROCESS CRUNCHER(BUFFER1, S1) USING GRIND;
  PROCESS CRUNCHER(BUFFER2, S2) USING CRUNCH;
  GOTO NEXT;
END;
READER: PROCEDURE(INBUF,I);
  DECLARE INBUF(100) NUMBER;
  DECLARE I SEMAPHORE;
  REQUEST I;
  INPUT(INPUT);
END;
WRITER: PROCEDURE(OUTBUF, O);
  DECLARE OUTBUF(100) NUMBER;
  DECLARE O SEMAPHORE;
  OUTPUT(OUTBUF);
  RELEASE O;
END;
CRUNCHER: PROCEDURE(NUMS, S);
  DECLARE NUMS(100) NUMBER;
  DECLARE S SEMAPHORE;
  DECLARE I NUMBER;
  DECLARE J NUMBER;
  I = 1;
  NEXT: J = I + 1;
  IF J = 100 THEN GOTO XIT;
  NUMS(I) = NUMS(J) * NUMS(100) / NUMS(I);
  I = I + 1;
  GOTO NEXT;
XIT: INTERRUPT P FOR WRITER(NUMS, S);
END;

Fig. 1--A TRIPLE program
CHAPTER BIBLIOGRAPHY


CHAPTER II

A DESCRIPTION OF TRIPLE

This chapter defines the syntax of TRIPLE in a manner similar to the definition of ALGOL by Naur (1). The semantics of TRIPLE are only briefly described in this chapter. Chapter III contains a rigorous definition of those features of TRIPLE which are of interest in concurrent processing.

The metalanguage used in describing TRIPLE is Backus Naur Form or Backus Normal Form (1). Terminal symbols represent themselves. Nonterminal symbols are enclosed in meta brackets, "<" and ">." The symbol ":=" is read "is composed of" and is used to separate the defined symbol on the left of a production from the definition on the right. The symbol "v" is read "or" and is used to separate alternate definitions in a production. The symbol NULL is used to represent the null or empty production.

Basic Language Elements, Variables, and Expressions

Basic Symbols

Syntax.---

<letter> ::= "A v B v C v D v E v F v G v H v I v J v K v L

v M v N v C v P v Q v R v S v T v U v V v W v X v Y v Z
<digit> ::= 0 v 1 v 2 v 3 v 4 v 5 v 6 v 7 v 8 v 9
<separator> ::= , v ; v =
<paren> ::= ( v )
<unary operator> ::= + v - v ~
<arithmetic operator> ::= + v - v * v /
<logical operator> ::= & v | v
<relational operator> ::= < v <= v = v >= v > v =
<operator> ::= <arithmetic operator> v <logical operator> v <relational operator> v <unary operator>
<delimiter> ::= <operator> v <separator> v <paren>

Semantics.--Letters are used in forming identifiers. Digits are used in forming identifiers and numbers. The various delimiters usually represent themselves in a production. The particular meaning of each delimiter will be shown by the context in which it is used.

Identifiers

Syntax.--
<identifier> ::= <letter> v <identifier> <letter> v <identifier> <digit>
<reserved identifier> ::= CALL v DECLARE v END v FOR v GOTO v IF v INPUT v INTERRUPT v INTERRUPTS v NUMBER v OUTPUT v PROCEDURE v PROCESS v PROCESSOR v RELEASE v REQUEST v RESUME v SEMAPHORE v THEN v USING v VALUE v WITH
Examples.—The following are examples of identifiers:

1. A
2. Z123
3. Q2R
4. ANTIDISESTABLISHMENTARISM

Semantics.—Identifiers are used as names for simple variables, arrays, semaphores, processors, labels, and procedures. Identifiers may be freely chosen except that they may not be chosen from the list of reserved identifiers.

Reserved identifiers are special terminal symbols that have specific meanings in the language. The meaning of each reserved identifier is determined by the context in which it is used. Reserved identifiers represent themselves in a production.

Numbers and Logical Values

Syntax.—

\[<\text{unsigned number}> ::= <\text{digit}> v <\text{unsigned number}> <\text{digit}>\]
\[<\text{number}> ::= <\text{unsigned number}> v - <\text{unsigned number}>\]
\[v + <\text{unsigned number}>\]
\[<\text{logical value}> ::= <\text{number}>\]

Examples.—The following items are examples of numbers:

1. 1
2. -287
3. 32767
Semantics.--Numbers are interpreted as whole decimal numbers. When numbers are interpreted as logical values, the number 0 is equivalent to the value false; any other number is equivalent to the logical value true.

Semaphores

Syntax.--
<semaphore count> ::= <number>
<semaphore> ::= <semaphore count> QHEAD

Semantics.--The semaphore is a complex data structure. It consists of a semaphore count which is accessible as a number value and the single entity, QHEAD. QHEAD is empty if the semaphore count is zero or positive. QHEAD is the first pointer in a linked list if the semaphore count is negative.

Processors

Syntax.--
<process header count> ::= <number>
<processor> ::= <process header count> MACHINE

Semantics.--A processor is a conceptual machine to which some specified number of processes can be assigned. A positive process header count is the number of process headers available for starting processes on the machine. If the process header count is negative, it is the number of
processes waiting to be started on the machine. The process header count is accessible as a number value.

The MACHINE portion of the processor includes the data structures and control mechanisms necessary to schedule multiple processes assigned to the machine and to manage a priority-interrupt structure.

Variables

Syntax.—

<variable identifier> ::= <identifier>
<simple variable> ::= <variable identifier>
<array identifier> ::= <identifier>
<subscripted variable> ::= <array identifier>
    ( <expression> )
<variable> ::= <simple variable> y <subscripted variable>
<number variable> ::= <variable>
<semaphore variable> ::= <variable>
<processor variable> ::= <variable>

Examples.—The following are examples of variables:

1. AQ
2. EL(AQ+B*7)
3. Y(I(J))

Semantics.—A variable is a name given to a single value. The value may be a number value, a semaphore value, or a processor value. The name of a value allows
it to be referenced and changed. Subscripted variables are components of vector arrays.

Expressions

**Syntax.**

- `<adding operator> ::= + v -`
- `<multiplying operator> ::= * v /`
- `<unsigned operand> ::= <number variable>
  - `v <semaphore variable> v <processor variable>
  - v ( <expression> )`
- `<operand> ::= <unsigned operand>
  - `v <unary operator> <unsigned operand>`
- `<term> ::= <operand>
  - `v <term> <multiplying operator> <operand>`
- `<arithmetic expression> ::= <term>
  - `v <arithmetic expression> <adding operator> <term>`
- `<logical expression> ::= <arithmetic expression>
  - `v <logical expression> <relational operator>
    <arithmetic expression>`
- `<expression> ::= <logical expression>
  - `v <expression> <logical operator>
    <logical expression>`

**Examples.**—The following items are examples of unsigned operands:

1. AB
2. (A + 2)
The following items are examples of operands:
1. AB
2. -AB
3. (A + 2)
4. +7

The following items are examples of terms:
1. -AB
2. AB * 3
3. C * -8
4. -A * (C+7) * Q(I)

The following items are examples of arithmetic expressions:
1. AB
2. C * -8 + AQ
3. I - 1
4. I + J - C

The following items are examples of logical expressions:
1. A < B < C
2. AB
3. (I - 1) >= F
4. -A = C = B

The following items are examples of expressions:
1. A
2. (I + J - C) * 2 & -A = C | F
3. AB * 3
4. A + B * C - Q(I) / 17
Semantics.—The result of an expression is a number value. The number value is obtained by applying each operator to the proper operand or operands in the order determined by the precedence rules for the operators. The following precedence rules are determined by the syntax:

1. Expressions are evaluated from the most deeply nested expression outward.
2. Unary operators are evaluated first.
3. Multiplication operators are evaluated second.
4. Adding operators are evaluated third.
5. Relational operators are evaluated fourth.
6. Logical operators are evaluated last.
7. Operators of the same precedence are evaluated from left to right.

The unary operators "+" and "−" have the usual arithmetic meaning. The unary operator "¬" read "not" yields a logical value. Since a logical value is a number value, the result of the not operator is 0 if the operand value was non-zero and 1 if the operand value was zero.

The multiplying and adding operators have the usual arithmetic meaning. The relational operators yield a logical result which is the number 0 or 1 corresponding to false and true, respectively. The logical operators "&" and "|" are read "and" and "or," respectively. These operators also yield a logical result.
When a number variable is used as an operand in an expression, the value of the number is supplied in the usual manner. When a semaphore variable is used as an operand, the semaphore count is supplied. When a processor variable is used as an operand, the number of available process headers is supplied as the value of the operand.

Declaration Statements

Declaration statements are used to specify the type of values a variable may assume.

Syntax

<size designator> ::= ( <number> ) v NULL

<value designator> ::= VALUE v NULL

<number name> ::= <identifier>

<number declaration> ::= <number name> <size designator>

            NUMBER <value designator>

<semaphore name> ::= <identifier>

<semaphore declaration> ::= <semaphore name>

            <size designator> SEMAPHORE

<processor name> ::= <identifier>

<interrupt name> ::= <procedure name>

<interrupt list> ::= <interrupt name>

                    v <interrupt list> , <interrupt name>

<interrupt designator> ::= WITH INTERRUPTS

                    ( <interrupt list> ) v NULL
Semantics and Examples

The declaration statement is not an executable statement. Its purpose is to reserve a place for a value of a particular type. The syntax of the particular declaration statement depends on the type of value for which space is being reserved.

Number declarations.—The following statements are examples of number declarations:

1. DECLARE ACK NUMBER
2. DECLARE CLE(3) NUMBER
3. DECLARE BBB NUMBER VALUE

A vector array of numbers can be declared by specifying the size of the array in parenthesis following the number identifier. If the number is a parameter of the procedure in which it is declared, the value designator can be used to specify that the number is to be passed by value rather than by location.
Semaphore declarations.—The following statements are examples of semaphore declarations:

1. DECLARE SV SEMAPHORE
2. DECLARE SYNC(10) SEMAPHORE
3. DECLARE A SEMAPHORE
4. DECLARE EXCLUSION(2) SEMAPHORE

A vector array of semaphores can be declared by specifying the size of the array in parenthesis following the semaphore identifier.

Processor declarations.—The following statements are examples of processor declarations:

1. DECLARE P PROCESSOR
2. DECLARE Q(10) PROCESSOR(2)
3. DECLARE Z PROCESSOR(4) WITH INTERRUPTS(A,B,C)
4. DECLARE R PROCESSOR WITH INTERRUPT(ERROR)

An array of identical processors is declared by placing the size of the array in parenthesis following the processor identifier. If a processor is to have more than one process header, the number of process headers must be specified in parenthesis following the reserved identifier PROCESSOR. If a processor is to have an interrupt structure, a list of procedure names must be specified in parenthesis following one of the reserved identifiers, INTERRUPT or INTERRUPTS. The first procedure name in the list will be assigned to the highest interrupt level. The remaining procedure names will be assigned in the order in which they appear.
Basic Executable Statements

TRIPLE's basic executable statements provide a set of facilities common to most programming languages.

Assignment Statements

Syntax.--

<receiving identifier> := <number identifier>

v <semaphore identifier>

<assignment statement> := <receiving identifier>

= <expression>

Examples.--The following items are examples of assignment statements:

1. A = B
2. C(Q+2) = B | F + 2
3. S = 0
4. R(I) = F2 / F3 * 7

Semantics.--The assignment statement is provided as a convenient method of changing the value of a number identifier or a non-negative valued semaphore identifier. The assignment statement is executed by evaluating the expression and setting the value of the receiving identifier to the resulting value.

A positive or zero valued semaphore may be assigned a positive or zero value. A negative valued semaphore may not be changed with the assignment statement.
Conditional Statements

Syntax.--

<condition> ::= <expression>

<unconditional statement> ::= <assignment statement>
   v <goto statement> v <input statement>
   v <output statement> v <call statement>
   v <process statement> v <interrupt statement>
   v <resume statement> v <request statement>
   v <release statement>

<conditional statement> ::= IF <condition> THEN
   <unconditional statement>

Examples.--The following items are examples of conditional statements:
1. IF A = B THEN GOTO L
2. IF Z THEN A = 47
3. IF A * 2 + 93 - Z THEN CALL SUB
4. IF Q(I) < 0 THEN RESUME

Semantics.--The conditional statement is provided so that decision-making mechanisms can be constructed. The first step in the execution of the conditional statement is the evaluation of the condition. The condition is evaluated as an expression and the result, a number value, is given a true or false interpretation. If the number value is not zero, the condition is true and the unconditional statement
is executed; otherwise, the condition is false and the unconditional statement is not executed.

**Goto Statements**

**Syntax.**

\[
\text{<goto target statement> ::= <conditional statement>}
\]

\[
\text{v <unconditional statement> v <end statement>}
\]

\[
\text{<statement label> ::= <identifier>}
\]

\[
\text{<goto target> ::= <statement label> ; <goto target statement>}
\]

\[
\text{<goto statement> ::= GOTO <statement label>}
\]

**Examples.**—In Figure 2 statements 3 and 4 are examples of goto statements.

```plaintext
1 S: PROCEDURE;
2   DECLARE X NUMBER;
   .
   .
3   GOTO L2;
4   IF X THEN GOTO XIT;
5   L2: X = 7;
   .
   .
6   XIT: END;
```

Fig. 2--Partial program containing example goto statements.
Semantics.—The only use for statement labels is their reference in goto statements. The goto statement alters the usual flow of control when executed. The next statement to be executed after a goto statement is the goto target statement rather than the statement immediately following the goto.

Input and Output Statements

Syntax.—

\[
\text{<input/output list> ::= <variable>}
\]

\[
v <\text{input/output list}>, <\text{variable}>
\]

\[
\text{<input statement> ::= INPUT ( <input/output list> )}
\]

\[
\text{<output statement> ::= OUTPUT ( <input/output list> )}
\]

Examples.—The following items are examples of input or output statements:

1. \(\text{INPUT (A)}\)
2. \(\text{INPUT(B(I+2),C)}\)
3. \(\text{OUTPUT(X,Y,Z)}\)
4. \(\text{OUTPUT(T(Q*4))}\)

Semantics.—The input and output statements provide a minimal facility for supplying external values to a running program and for displaying values produced by a running program. The values displayed or accepted are number values. The variables named in the input/output list must conform to the same rules as the receiving variable for an
assignment statement. The named variables will be converted to number values on output.

Procedure, End, and Call Statements

A procedure is a sequence of statements initiated by a procedure statement and terminated by the first end statement not part of a nested procedure definition. The procedure is not executed inline, but can be invoked in several manners. The call statement is the simplest mechanism for invoking a procedure.

The Procedure Definition

Syntax.--

<dummy argument> ::= <identifier>
<dummy argument list> ::= <dummy argument>
  v <dummy argument list> , <dummy argument>
<procedure statement> ::= PROCEDURE
  v PROCEDURE ( <dummy argument list> )
<non-procedure statement> ::= <declare statement>
  v <statement label> : <executable statement>
  v <executable statement>
<executable statement> ::= <assignment statement>
  v <input statement> v <output statement>
  v <call statement> v <process statement>
  v <interrupt statement> v <resume statement>
  v <request statement> v <release statement>
  v <conditional statement> v <goto statement>
Examples.--In Figure 3 is an example program consisting of a procedure containing one null procedure and one non-null procedure.

```plaintext
MAIN: PROCEDURE;
    DECLARE A NUMBER;
    DECLARE C NUMBER;
    RAP: PROCEDURE(A);
        DECLARE A NUMBER;
        A = 9;
    END;
EMPTt: PROCEDURE;
END;
CALL RAP(C);
END;
```

Fig. 3--A program showing the method of procedure definition
Semantics.—A TRIPLE program is a procedure. The simplest possible program is a procedure statement followed by an end statement.

A procedure statement may specify dummy parameters in the argument list. The actual parameters must be supplied when the procedure is invoked. The procedure must contain declaration statements for all dummy parameters.

A procedure is invoked by a transfer of control to its procedure statement. Special statements are provided for this purpose. The only way to exit a procedure is to execute its end statement. Control can be transferred to the end statement with a goto statement.

The Call Statement

Syntax.--

<actual argument> ::= <expression>

<actual argument list> ::= <actual argument>

 v <actual argument list> , <actual argument>

<procedure reference> ::= <procedure name>

 v <procedure name> ( <actual argument list> )

<call statement> ::= CALL <procedure reference>

Examples.—The following items are examples of call statements:

1. CALL SUB

2. CALL R(ALPHA,Z+2)
3. CALL FACTOR(39)
4. CALL WAIT(SEM(9))

Semantics.—The call statement is used to invoke a procedure in a way that will cause control to be returned to the statement immediately after the call when the procedure has ended. The invoked procedure is supplied with actual arguments by placing the actual argument in the position in the actual argument list corresponding to the proper dummy argument's position in the dummy argument list.

Process, Interrupt, and Resume Statements

The process and interrupt statements are each methods of invoking a procedure and simultaneously creating a new process. The process executing the process or interrupt statement continues its execution with the next statement without waiting for the invoked procedure to complete. A new process is created by both the process and interrupt statements. The resume statement is a special means of suspending progress in a procedure to permit coroutine execution.

The Process Statement

Syntax.—

<procedure reference list> ::= <procedure reference>

v <procedure reference list>,

<procedure reference>
**<processor designator> ::= USING <processor identifier>**

v NULL

**<process statement> ::= PROCESS <procedure reference list>**

**<processor designator>**

**Examples.**—The following items are examples of process statements:

1. PROCESS A
2. PROCESS SUB(2) USING G
3. PROCESS S,Q(F,I) USING P

**Semantics.**—A process statement will be executed when it is encountered if there are a sufficient number of process headers on the processor to be used in starting the new process or processes. If there is an insufficient number of process headers, the execution of the process statement will be delayed until the process headers are available.

The parameter-passing mechanism for a process-statement-invoked procedure is the same as for a call-statement-invoked procedure. The values of the passed parameters will be their values at the time the process or processes are actually started.

A process is started on the specified processor for each procedure reference in the process statement. If multiple processes are started, they are tagged as a set of coroutines only one of which can be active at a time.
A process is a transient entity. It will cease to exist when the procedure specified in the procedure reference of the process statement ends. During its life the process is, for the most part, independent of other processes. It has access to the environment in which the process statement creating it was executed. The process is advanced only by the processor specified in the process statement. The process may be required to share the processor with other processes and interrupt processes.

The Interrupt Statement

Syntax.--

<interrupt reference> ::= <interrupt name>

v ( <actual argument list> )

<interrupt statement> ::= INTERRUPT <processor name>

FOR <interrupt reference>

Examples.--The following items are examples of interrupt statements:

1. INTERRUPT P FOR ERROR
2. INTERRUPT P FOR ERROR(27, I)
3. INTERRUPT WRITER FOR FULLBUFFER(Z)

Semantics.--The interrupt statement specifies a processor to be interrupted. It also specifies on which of a processor's interrupt levels the interrupt is to take place. If the interrupt level is inactive at the time
the interrupt statement is executed, the level is made active and the interrupt process is started immediately. If the interrupt level is already active, the interrupt must be queued for later processing after the interrupt being processed and all previously queued interrupts are completed. In any case, the process executing the interrupt statement is never blocked. It continues to the next statement.

Parameter-passing mechanisms for the interrupt statement are the same as for the call statement. It should be noted, however, that the value of parameters passed by value to the interrupt procedure is the value at the time the interrupt statement is executed rather than at the time the interrupt is actually processed.

**The Resume Statement**

**Syntax.**--

<resume statement> ::= RESUME

**Semantics.**--The resume statement switches execution from one coroutine to another coroutine of the same set. When the coroutine executing the resume statement again executes, it will continue at the statement just after the resume statement. Coroutines execute in round-robin fashion in the order in which they were named in the process statement creating them. A resume statement executed by the last named coroutine will pass execution to the first.
Request and Release Statements

The request and release statements are provided to operate on semaphores. They are useful in process synchronization.

Syntax

<request statement> ::= REQUEST <semaphore name>
<release statement> ::= RELEASE <semaphore name>

Examples

The following items are examples of request and release statements:

1. REQUEST S
2. RELEASE S
3. REQUEST QT(J)
4. RELEASE QT(J-2)

Semantics

The execution of a request statement causes the semaphore count to be decremented. If the resulting count is zero or positive, the process executing the request statement continues immediately to execute the next statement. If the resulting semaphore count is negative, the executing process is queued on the semaphore and not permitted to proceed to the next statement until some other process has increased the semaphore count by executing a release statement a sufficient number of times to remove the queued process.
The execution of a release statement causes the semaphore count to be incremented. If the resulting count is positive, the executing process continues to the next statement. If the count is zero or negative, a queued process must be removed before the execution of the release statement is complete.
CHAPTER III

A SEMANTIC MODEL OF TRIPLE

In this chapter a semantic model is developed for TRIPLE. The model is sufficiently rigorous to be useful in proving assertions about programs written in TRIPLE. The model is clear and simple enough to be a great aid in the writing and understanding of TRIPLE programs. Most importantly, however, the model gives a way of describing a TRIPLE program in execution without reference to any particular implementation of the language. Since the model is based on the contour model as described by Johnston (1), it is called the TRIPLE contour model.

The Basic Elements of the TRIPLE Contour Model

The contour model is an effective descriptive mechanism for block-structured computations. In the model the two components of a computation are the algorithm component and the record-of-execution component.

The algorithm component is the static program which is referenced but never altered during the execution of the algorithm. An algorithm is shown in Figure 4. In the algorithm component a number is associated with each statement so that the statement can be referenced by the record-of-execution component. Statements may be omitted
Fig. 4--The algorithm component

from the representation of the algorithm component if they are not of consequence to the computation being considered. The omission of one or more statements is indicated by a series of three dots.

The record-of-execution component is a sequence of snapshots which depicts the changes during the execution of the algorithm. A record-of-execution component is shown in Figure 5. Each snapshot shows to what point or points in the program execution has progressed and the state of any data items that were created during the execution.

Each snapshot contains a representation of the process initiated by the system when the computation was started. If other processes exist, they will also be represented in
Fig. 5--A record of execution with n snapshots

the snapshot. Each snapshot also contains a model of the environment of each process shown in the snapshot. The basic unit of the environment model is a contour.
The Model of a Process

A process is represented in the TRIPLE contour model by two pointers. The environment pointer is to the environment of the process. The instruction pointer is to the next statement of the algorithm component to be executed for the process.

A process can be in the ready state or the blocked state. A process in the ready state can have the statement pointed to by its instruction pointer executed for it by the processor to which it is assigned. A ready process is shown in Figure 6. It is represented by a circle containing

![Fig. 6--A ready process](image)

the environment pointer and the instruction pointer. A process in the blocked state is not eligible for activity from a processor. A blocked process is shown in Figure 7.

![Fig. 7--A blocked process](image)
It is represented by a square containing the environment pointer and the instruction pointer.

The instruction pointer of a process is always shown below its environment pointer so that there will be no confusion of the two pointers. The instruction pointer is often shown as a statement number. The environment pointer is sometimes shown as a contour name.

The Environment Model

Each time a procedure is entered during the execution of a program, any data items declared in the procedure are allocated. This allocation is modeled in the next snapshot by the creation of a rectangular area called a contour. A contour is shown in Figure 8.

![Fig. 8--A contour](image)

A contour has a name displayed in the break in the rectangle. The name consists of the associated procedure name sometimes followed by a number. This number may be required to identify the particular contour when there is more than one associated with a procedure. Figure 9 shows two such contours.
A contour may be enclosed by another contour. The nesting of contours corresponds to the static nesting of the procedure definitions. Figure 10 shows a possible nesting of contours obtainable from a particular algorithm.

![Diagram of nested contours]

**Fig. 9**—Two contours associated with a procedure

**Fig. 10**—Example of nested contours
Space is reserved within each contour for the names and values of the data items declared in the associated procedure as well as for any additional information that may be needed in the execution of the procedure such as a return address. The data items are displayed in the upper left corner of the contour, as shown in Figure 11.

Fig. 11--Data representation in a contour

Each data item model consists of the name of the data item, necessary information about the structure of the data item, and the value of the data item. The name appears as the leftmost entry for each item of data. The representation of the value depends on the type of the data item.
Numbers

A number is the simplest of data items. The model of a number shows the name of the number and the value of the number within the contour where it was allocated. The value is a decimal integer. The value of a number can be undefined if the number has not been assigned a value. Figure 12 shows two numbers within a contour. The number named A has a value of 7. The value of the number named Z is undefined.

<table>
<thead>
<tr>
<th>A</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 12--The model of a contour containing two numbers

Semaphores

A semaphore consists of a number and a pointer. The number is sometimes referred to as the semaphore counter. If the value of the counter is zero or positive, the pointer is empty as shown in Figure 13. When a semaphore is allocated its counter is zero.
If the semaphore counter is negative, the pointer is the head of a linked list. The negative of the counter is the number of nodes in the list. Nodes are shown in the model as small circles. Each node is associated with a blocked process. The linked list of processes is sometimes called a queue since it is accessed as a queue. Figure 14 shows a semaphore queue of two blocked processes.

Fig. 13--A semaphore with an empty pointer

Fig. 14--A semaphore queue containing two processes
A processor is a complex entity. It is composed of the following items:

1. a process header table definition and status,
2. an interrupt structure definition and status,
3. a control component, and
4. a number value.

As shown in Figure 15, the model of a processor displays the process header table and the interrupt structure as data items within the contour associated with the procedure declaring the processor. As the control component model is identical for all processors, it is not shown associated with a particular processor. The number value of a processor is derived from the process header table. Processors are independent of one another. It is correct to think of the processors as implemented on distinct hardware units sharing a common environment.

![Fig. 15--The model of a processor P](image-url)
The process header table.—A processor's process header table contains the information associated with its started processes and any queued processes which have requested its unavailable process headers. If a processor has no started processes, the process header table is shown to be empty as in Figure 16.

Fig. 16--Model of a processor having no started processes

The maximum number of processes that can be assigned to a processor at one time is determined by the number of process headers in the process header table. The number of process headers is shown in the model below the processor name. Figures 16 and 17 show a processor with a maximum of three processes. Figure 17 shows all the process headers in use.

Fig. 17--Model of a processor having three started processes
If there are no processes waiting to be started on a processor, the process header queue is empty and not shown. For example, Figure 1? is such a processor. If there are processes waiting to be started on a processor, the process header queue head is shown adjoining the process header table on the right as in Figure 18. The process header queue can be thought of as part of a semaphore whose value is the number of available process headers on the processor.

Fig. 18—The model of a process on processor R queued waiting for a process header in processor P.

be thought of as part of a semaphore whose value is the number of available process headers on the processor.
The interrupt structure.—A processor's interrupt structure consists of some number of interrupt levels each associated with a procedure label. The interrupt level priority corresponds to the location of the interrupt level in the structure. The highest priority level is highest in the structure.

If a processor has no interrupt structure, the dashed line separating the nonexistent interrupt structure from the process header table is not shown. A processor having no interrupt structure is shown in Figure 19. If a processor has an interrupt structure, each interrupt level

![Fig. 19--A processor with no interrupt structure](image1)

is separated from the next interrupt level or the process header table by a dashed line. A processor with two interrupt levels is shown in Figure 20.

![Fig. 20--A processor with two interrupt levels](image2)
An active interrupt level has a started interrupt process on the level. The interrupt level associated with procedures B and C are active in Figure 21. These interrupt levels can be referred to as levels B and C, referencing them by the associated procedure names. The dashed line below the highest active interrupt level is replaced by a double solid line. This line indicates that no process or interrupt process below the double line is eligible for processor activity unless the process above the line is blocked.

Fig. 21--A processor with two active interrupts
If there are interrupts pending on an active interrupt level, an interrupt level queue head is shown adjoining the interrupt level on the right. Figure 22 shows a processor with an interrupt level queue on level $F$. The entries in the queue are contours each defining a distinct interrupt. When one of the queued interrupts is to be processed, the environment pointer of the interrupt process is initialized to the appropriate contour.
The control component.—The control component is identical for all processors. It is modeled by an algorithm which determines which of a processor's processes or interrupt processes it is to advance at any moment. To advance a process or interrupt process, the processor executes a statement for it. A snapshot may be taken between statement executions. A statement may not be partially executed between snapshots.

The processor control algorithm is shown in Figure 23. The algorithm causes a processor to poll the interrupt levels between statement executions. A statement is executed for the highest level ready interrupt process or for the current ready process if no interrupt level is active. Each process gets a turn to be the current process in a round-robin fashion. If the process is not ready when its turn comes, it must wait until its next turn before it again becomes a candidate for activity by the processor.

Processors are independent of one another although they share the control component algorithm. One processor's place of execution in the algorithm is independent of all other executions of the control algorithm. Between snapshots one or more processors may be arbitrarily chosen to show activity. The order in which multiple processors execute statements in a sequence of snapshots is indeterminate.

The control component of a processor is analogous to that of a sophisticated computer control mechanism or the basic operating system for a much simpler computer.
start

set current process to first process

interrupt structure

set level to highest interrupt level

execute one statement

level active

process blocked

lowest level

decrease level

current process active

execute one statement

Fig. 23--Algorithm for the control component of a processor
If a process advanced since the last snapshot, it will be denoted by a pointer as shown in Figure 24. Normally only one statement will be executed between snapshots.

![Diagram](image)

*Fig. 24—A partial snapshot showing a process that advanced since the last snapshot.*

However, this is not a requirement of the model. If statements are executed for several processes by the same or different processors, all the advanced processes will be indicated by pointers. Figure 25 shows a partial snapshot with two processes advanced since the last snapshot.

![Diagram](image)

*Fig. 25—A partial snapshot showing two processes that advanced since the last snapshot.*
The number value of a processor.—Each processor has a number value that can be referenced in expressions. The value is the number of available process headers if any are available. If no process headers are available, the number value is zero or the negative of the number of processes on the process header queue.

Arrays

Arrays are of only one dimension. There are arrays of numbers, arrays of processors, and arrays of semaphores. The name of a particular element of an array is formed from the name of the array followed by the number of the element enclosed in parenthesis. An array named S containing ten elements is shown in Figure 26. Elements can be omitted from the array in a snapshot if the omitted elements are not pertinent to the snapshot. The omitted elements must be replaced by a series of three vertical dots.

<table>
<thead>
<tr>
<th>S(1)</th>
<th>3</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(2)</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>S(3)</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(10)</td>
<td>23</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 26—An array of semaphores
Models of Statement and Procedure Labels

Statement labels must be local to the procedure in which they are referenced. The statement label is an instruction pointer to the appropriate statement in the algorithm. A statement label is shown in Figure 27.

![Statement label and a procedure label](image)

Fig. 27--A statement label and a procedure label

Procedure labels need not be local to the procedure in which they are referenced; but they must be in a surrounding procedure if they are not local. A procedure label is an instruction pointer to the appropriate procedure statement in the algorithm. A procedure label is shown in Figure 27.

A reference to a procedure label is shown in each interrupt level of a processor. The reference is simply the name of the procedure. The instruction pointer for the procedure is found by searching contours outward until the procedure name is found.
Models of Procedure and Call Statements

The result of a procedure statement execution is the creation of a contour. The nesting level of the contour is that of the procedure statement in the static program. In Figure 28 all entrances to procedure B will cause a B contour to be nested in contour A, since procedure B is nested in procedure A.

![Diagram](image1)

Fig. 28—A contour A containing two contours created by two distinct entrances to procedure B.

Each contour has a generated data item, called the dynamic link, containing a pointer to the environment in which control was transferred to the procedure statement. In some cases the dynamic link contains a pointer to the statement that transferred control to the procedure statement. Figure 29 shows two dynamic links. The dynamic link in contour D has a null instruction pointer. The primary
Figure 29—Dynamic links

Purpose of the dynamic link is in determining the type of action to be taken when the end statement associated with the contour is executed.

The call statement is the simplest manner of transferring control to a procedure statement. The sequence of snapshots in Figure 30 shows the model of call statement execution. The first snapshot is taken just before the execution of the call statement, so the instruction pointer is to the call. The last snapshot is taken just after the call statement is executed. Since the call statement causes procedure entry, the last snapshot is after execution of the procedure statement. The instruction pointer of the active process is to the next executable statement after the procedure statement. The environment pointer is to the new contour. The instruction pointer of the dynamic link is to the statement just after the call. The environment pointer of the dynamic link is to the contour defining the environment at the time the call statement was executed.
Fig. 30--Partial computation showing the effect of executing a call statement.
The call statement execution is complicated by the addition of parameter-passing mechanisms. A parameter is passed by placing its name in the proper position in the argument list of the call statement. In the corresponding position of the procedure definition argument list is a dummy name for the parameter. In the model dummy parameters appear before any other data items in the contour associated with the procedure. Figure 31 shows a skeleton contour for a procedure with parameters.

Fig. 31--Skeleton contour showing the location of dummy parameters.
Parameters can be passed by value or by location. If a parameter is passed by value, the value of the actual parameter is copied to the value of the dummy parameter at the time of the call. The value of the dummy parameter can be used and changed, but the value of the actual parameter cannot be modified by the procedure. If a parameter is passed by location, its address is calculated at the time of the call and made known in the new contour. The value of the dummy parameter is a pointer to the actual parameter. The pointer to the actual parameter has two parts. The first part is a pointer to the contour where the actual parameter was allocated. The second part is the location relative to the beginning of the contour of the data item in the contour.

Figure 32 shows the execution of a call to a procedure having parameters passed by location and by value. Prior to the first snapshot, a call to procedure Y was made. The call to procedure Z takes place in the environment defined by contour Y. The next snapshot is taken immediately after procedure Z was entered. Dummy parameter R points to actual parameter A in contour X. Dummy parameter S has been given the value of the actual parameter B at the time of the call. Dummy parameter T points to actual parameter C in contour Y. The environment pointer of contour Z's dynamic link points to contour Y since the environment pointer of the active process was to Y at the time of the call.
1  X: PROCEDURE;
2    DECLARE A NUMBER;
3    CALL Y;
4
5  Y: PROCEDURE;
6    DECLARE B NUMBER;
7    DECLARE C NUMBER;
8    A = 100;
9    B = 200;
10   C = 300;
11   CALL Z(A,B,C);
12
13  Z: PROCEDURE(R,S,T);
14    DECLARE R NUMBER;
15    DECLARE S
16       NUMBER VALUE;
17    DECLARE T NUMBER;
18
19  END;
20

Fig. 32--Models of parameters passed by location and by value
The Process-Statement Model

The process statement is used to create processes and assign them to a processor. The process statement references a procedure that will determine the initial instruction pointer and environment pointer for the newly created process. The process statement also specifies the name of the processor that is to service the process. If no processor name is specified, the process is to be assigned to the processor executing the process statement.

When a process statement is executed, the referenced processor may not have available a process header to assign to the process. If this is the case, the process for which the process statement is being executed is blocked and placed in the queue of processes waiting for process headers to become available. Figure 33 shows a snapshot taken just after a process statement was executed when no process header was available on the referenced processor. The active process is blocked on the process header queue of processor Q.

A process header will become available when an end statement causing a process end is executed. At that time the blocked process is made ready and the process statement is executed.

A process is started by making an entry for it in an available process header of the processor referenced in the process statement. The environment of the process is created by executing the procedure statement of the procedure
Fig. 33--Partial computation showing the effect of executing a process statement referencing a processor with no available process headers.

referenced in the process statement. Parameter-passing mechanisms are identical to those of the call statement. The instruction pointer of the process is initialized to the first executable statement following the procedure statement. The instruction pointer of the newly created contour is null, since no return will be made to the invoking process. The environment pointer of the contour exists because the environment in which the process statement was executed is part of the environment of the new process.
Figure 34 shows a partial computation in which a process statement is executed, causing a process to be started on processor Q.

```
1 A: PROCEDURE;
2 DECLARE Q PROCESSOR;
3 PROCESS B USING Q;
4 B: PROCEDURE;
   .
5  END;
   .
6  END;
```

**Snapshot 1**

**Snapshot 2**

---

Fig. 34—Partial computation showing the effect of executing a process statement causing a process to be started.
A process statement may specify that more than one process for a processor be started. This is done by specifying a procedure reference in the process statement for each process to be started. The process statement will be executed when a sufficient number of process headers is available to start all the processes simultaneously. The processes started in this manner will be chained together so that they may function as coroutines. The first process specified in the statement will be created in the active state. Any other processes will be created in the blocked state. Figure 35 shows the creation of coroutines using the process statement.

Fig. 35—Creation of coroutines with the process statement
The Interrupt-Statement Model

The interrupt statement is used to interrupt a processor. The name of the processor that is to be interrupted is specified in the interrupt statement. The level of the interrupt is determined by matching the procedure name specified in the interrupt statement with one of the procedure names in the interrupt structure of the processor.

When an interrupt statement is executed, an interrupt process is started or the interrupt is queued depending on whether or not the interrupt level is already active. Figure 36 shows the algorithm that will be used in examining the various effects of the interrupt statement.

```
1  BETA: PROCEDURE;
2   DECLARE X PROCESSOR;
3   DECLARE Y PROCESSOR WITH INTERRUPT(YI);
4   PROCESS XCODE USING X;
5   PROCESS YCODE USING Y;
6   XCODE: PROCEDURE;
7     DECLARE A NUMBER;
8     A = 47;
9     INTERRUPT Y FOR YI(A);
10    END;
11  YCODE: PROCEDURE;
12    END;
13  YI: PROCEDURE(B);
14    DECLARE B NUMBER VALUE;
15    END;
16    INTERRUPT Y FOR YI(16);
17    END;
```

Fig. 36--Algorithm showing interrupt statement execution
Figure 37 is a snapshot of a computation using the algorithm in Figure 36 taken just prior to the execution of the interrupt statement in procedure XCODE.

![Diagram showing computation snapshot]

**Fig. 37**—Snapshot 1 of a computation using the algorithm in Figure 36.

Figure 38 is a snapshot of a computation using the algorithm in Figure 36 taken just after the execution of the interrupt statement in procedure XCODE by processor X.
The active interrupt level has the effect of preventing any lower level activity on processor Y unless the interrupt process becomes blocked.

A process executing an interrupt statement is never blocked as a result of the interrupt statement execution. If the interrupt must be queued due to an active interrupt level, a contour is created by the execution of the procedure.
statement referenced in the interrupt statement. The contour is placed at the end of the queue of contours waiting on the interrupt level. Figure 39 is a snapshot of a computation using the algorithm in Figure 36 taken just after the execution of the interrupt statement in procedure BETA by the master processor. The interrupt statement causes contour Y1/2 to be placed on the interrupt level queue.

Fig. 39--Snapshot 3 of a computation using the algorithm in Figure 36.
The Resume-Statement Model

The resume statement is provided to permit coroutine execution. Coroutines are a set of processes only one of which is eligible to be active at a time. An active coroutine can pass its eligibility to the next coroutine of the set making it active, by executing a resume statement. The coroutine executing the resume statement loses its eligibility and remains blocked until a resume statement is executed by the coroutine to its left. Coroutines become eligible for activity in a round-robin fashion. The partial computation in Figure 40 shows snapshots before and after the execution of a resume statement by one of a set of coroutines. When the process statement that created the coroutines was executed, the coroutine associated with the first procedure name mentioned in the process statement was created in the active state. The other coroutines were created in the blocked state waiting for the execution of a resume statement by the eligible coroutine.

Even though a coroutine is eligible for activity, it may be blocked and on a semaphore queue. Since all the other coroutines of the set would be blocked waiting for the execution of a resume statement, no coroutine will advance until the eligible coroutine is released from the semaphore queue.

Other processes may exist along with a set of coroutines on a single processor. In this case the coroutines compete as if they were a single process for processor activity.
Fig. 40—Partial computation showing the result of executing a resume statement.
The End-Statement Model

The execution of an end statement causes deallocation of the contour pointed to by the environment pointer of the process or interrupt process for which the end statement is executed. The deallocation is immediate if there are no dynamic links in any contour pointing to the contour to be deallocated. If there are dynamic links pointing to the contour, the process or interrupt process is blocked to wait for the proper condition for deallocating the contour. Figure 41 shows a partial computation with snapshots taken before and after an end statement is executed at a time when the corresponding contour cannot be deallocated. Contour F must be retained since contour ZP's dynamic link points to it. The pointer from contour F to the master process symbolizes the fact that the master process cannot continue until some end statement has made contour F eligible for deallocation by removing the last pointer to it.

An end statement may cause deallocation of several contours. When a contour is deallocated, the contour pointed to by its dynamic link may be ready for deallocation. If the contour has a process blocked on an end statement and there are no dynamic links other than the one just followed pointing to the contour, it can be deallocated. Of course, its dynamic link must be followed in the same manner. Any process for which a contour is deallocated is left in the same state as it would have attained had it not been blocked on the first attempt to execute the end statement.
Fig. 41--Partial computation showing the effect of end statement execution at a time when the contour cannot be deallocated.
The action following each contour deallocation is determined by the manner in which the contour was created. If the contour is not the initial contour, it was created when one of the following statements was executed:

1. a call statement,
2. a process statement referencing only one procedure,
3. a process statement referencing multiple procedures, or
4. an interrupt statement.

The different ways in which contours are created necessitate different types of action when the contour is deallocated. The five ending actions are identified by the corresponding manner of contour creation. The following types of ending actions are identified:

1. computation end,
2. call end,
3. process end,
4. coroutine end, and
5. interrupt end.

**The Computation End**

If the contour was created when the computation was started, it is the initial contour. The dynamic link of an initial contour consists of a null instruction pointer and a null environment pointer. Deallocation of the contour terminates the computation. The partial computation in
Figure 42 shows snapshots before and after the execution of an end statement that terminates a computation.

1 Q: PROCEDURE;
   .
   .
   2 END;

Fig. 42--Partial computation showing the effect of a computation end.

**The Call End**

If the contour was created by a call statement, the dynamic link will contain a pointer to the environment at the time of the call. The environment pointer of the current process is restored to that value. The dynamic link will also contain an instruction pointer to the statement just past the call. The instruction pointer of the current process is set to that value. Figure 43 shows a partial computation with snapshots taken before and after the execution of an end statement that causes a call end.
Fig. 43—Partial computation showing the effect of a call end.

1 A: PROCEDURE;
   ...
   ...
2 CALL Z;
   ...
   ...
3 Z: PROCEDURE;
   ...
   ...
4 END;
5 END;

Snapshot 1

Snapshot 2
The Process End

If the contour was created by a process statement, the dynamic link will contain a null instruction pointer. The environment pointer is used only in determining whether or not the contour to which it points is also eligible for deallocation.

The process end causes the process header occupied by the process to be emptied and the processor count incremented by one. If there are processes queued waiting for process headers, an attempt is made to remove the first process on the queue. The queued process can be removed if a sufficient number of process headers are available to start all the processes specified by the process statement it is trying to execute. If processes are started, they are left in the same state as they would have been had they been started on the first attempt to execute the process statement. The process that was blocked because of an insufficient number of process headers is left in an active state with its instruction pointer moved beyond the process statement. In Figure 44 is a program that will be used in Figures 45, 46, 47, and 48 to illustrate the process end. When the snapshot in Figure 45 is taken two processes have been previously started on processor Q. Since there were no process headers available when the master process executed statement 5, the process is blocked on the process header queue with the process statement not completed. One of the processes on
Fig. 44—An algorithm to be used in demonstrating the execution of a process end.

```
1 ALPHA: PROCEDURE;
2   DECLARE Q PROCESSOR(2);
3   PROCESS ASUB USING Q;
4   PROCESS BSUB USING Q;
5   PROCESS CSUB USING Q;
6   ASUB: PROCEDURE;
7     PROCESS CSUB;
8     END;
9   BSUB: PROCEDURE;
10     END;
11   CSUB: PROCEDURE;
12     END;
13   END;
```

Fig. 45—Snapshot 1 from a computation using the algorithm in Figure 44.
processor Q is also blocked awaiting a process header. Between the snapshots in Figures 45 and 46 an end statement was executed. The end statement caused deallocation of contour BSUB which resulted in a process end. The available process header permitted unblocking of the master process and the execution of process statement 5. Between the snapshots in Figures 46 and 47, end statement 12 was executed, resulting in the deallocation of contour CSUB and the end of the process for which the end statement was executed.

Fig. 46--Snapshot 2 from a computation using the algorithm in Figure 44.
The available process header was used to start a new process. The CSUB contour is different from the one just deallocated. Its dynamic link points to the ASUB contour rather than the ALPHA contour. End statement 13 was also executed between the snapshots in Figures 46 and 47. Contour ALPHA could not be deallocated since the dynamic link in contour ASUB points to it. The master process is left blocked waiting for the deallocation of contour ALPHA. The next snapshot

Fig. 47--Snapshot 3 from a computation using the algorithm in Figure 44.
in Figure 48 is taken after the execution of end statement 8. Contour ASUB could not be deallocated because of contour CSUB. The execution of end statement 12 causes contours CSUB, ASUB, and ALPHA to be deallocated and the computation to be terminated.

Fig. 48--Snapshot 4 from a computation using the algorithm in Figure 44.
The Coroutine End

The coroutine end is similar to the process end. All the actions of the process end apply to the coroutine end. Coroutines, however, require a bit of additional attention. When one of a set of coroutines is terminated, the coroutine to its right is resumed and the coroutine set is reduced by one member. In Figure 49 is an algorithm that will be used in illustrating the coroutine end. In Figure 50 are snapshots taken before and after an end statement is executed by one of a set of three coroutines causing a coroutine end.

1 H: PROCEDURE;
2 DECLARE T PROCESSOR(2);
3 PROCESS D,E,F USING T;
4 D: PROCEDURE;
5   .
5   .
5   END;
6 E: PROCEDURE;
7   .
7   .
7   END;
8 F: PROCEDURE;
9   .
9   .
9   END;
10 END;

Fig. 49--An algorithm to be used in illustrating the coroutine end.
Fig. 50—Snapshots before and after the execution of a coroutine end using the algorithm in Figure 49.
The Interrupt End

If the deallocated contour was created for the initial environment of an interrupt process, its deallocation causes termination of the interrupt process. The termination of the interrupt process will cause deactivation of the interrupt level if there are no interrupts queued for the level. If there are queued interrupts, the level is left active and a new interrupt process is created with a queued contour as the initial environment pointer and the instruction pointer set just past the procedure statement associated with the interrupt level. In Figure 51 is a partial computation showing the effect of an interrupt end causing deactivation of the interrupt level.

Fig. 51—Partial computation showing the effect of an interrupt end.
Models of Request and Release Statements

Request and release statements are used to operate on semaphores. They are provided along with semaphores to facilitate mutual exclusion of processes as well as assist in other process synchronization problems. The semaphore on which the request or release operation is to be performed is the only operand specified in either statement.

The Request Statement

When a request statement is executed for a process or interrupt process, the semaphore count is decremented by one. If the resulting count is zero or positive, there is no further action and the process is left in the active state with its instruction pointer moved past the request statement. Figure 52 shows the result of executing a request statement resulting in a positive or zero semaphore.

Fig. 52—The execution of a request statement resulting in a positive or zero semaphore.
If the result of decrementing the semaphore count is negative, the process executing the request statement is left blocked on the semaphore with the instruction pointer still pointing to the request statement. Figure 53 shows a partial computation in which a request statement is executed resulting in a negative semaphore.

![Snapshot 1][1]

![Snapshot 2][2]

Fig. 53—The execution of a request statement resulting in a negative semaphore.

**The Release Statement**

When a release statement is executed for a process or interrupt process, the semaphore count is incremented by one. If the resulting count is positive, there is no further action and the executing process is left in the active state with its instruction pointer moved past the request statement. Figure 54 shows the result of executing a release statement resulting in a positive semaphore.
Fig. 54--The execution of a release statement resulting in a positive semaphore.

If the result of incrementing the semaphore count is zero or negative, there are processes queued on the semaphore and the oldest one is removed. It is placed in the active state with its instruction pointer moved past the request statement that placed it on the semaphore. Figure 55 shows the result of executing a release statement resulting in a zero or negative semaphore count.
Fig. 55--The execution of a release statement resulting in a zero or negative semaphore.
CHAPTER BIBLIOGRAPHY

CHAPTER IV

CONCLUSION

The major contribution of this thesis to the knowledge of concurrent processing is TRIPLE's novel interrupt mechanism. A secondary contribution of this thesis to the more general area of programming language design is the description of the semantics of TRIPLE using a model developed for that purpose.

Interrupt mechanisms originated as a method of efficient synchronization of devices external to the central processor and the activity of the central processor. One can abstract this physical system by regarding each external device as a process assigned to its own processor. The activity of the central processor can also be viewed as a process. A natural extension of the abstraction is to consider each processor capable of supporting more than one process by sharing its resources among competing processes. TRIPLE is a programming language realization of the abstraction with the following features of particular interest:

1. The environment organization of the system of processes is block structured.

2. Processors are allocated and deleted as part of the environment.
3. A popular set of concurrent-processing primitives is included to facilitate the creation of processes and to allow processes on the same and different processors to communicate.

4. A process can interrupt a processor in much the same way as an external device interrupts the central processor in traditional computer systems.

TRIPLE's design involves a modification of the usual conceptual boundary between hardware and software. The design of TRIPLE supplies evidence that this modification may be necessary in order to provide a complete interrupt capability in a programming system. TRIPLE does not have several of the problems common in other systems. The problems are resolved in the following ways:

1. The interruption of a processor rather than a process allows the execution of any processes on the interrupted processor to produce results consistent with those that would have been obtained had the processor not been interrupted. Consistent results are attainable since the status of each process on a processor is recorded and changed independently from all others. An interrupt creates a new process that may slow the progress of other processes on the same processor; but, it will not alter the results of those processes unless it is intended by the user that it do so.
2. The synchronization of an interrupt process with other processes on the same or different processors is made simple by the fact that an interrupt process can communicate with other processes using any of the concurrent-processing primitives in TRIPLE and can be blocked in the same manner as any other process. By using the tools provided, an interrupt process can block itself until some other process has passed a critical section. It can cause processing to occur at any level by generating interrupts or by creating processes.

3. The full environment of an interrupt process is retained until the interrupt process has completed its processing, since the retention strategy as proposed by Berry (1) and used in TRIPLE specifies a mechanism which retains the environment in which the interrupt was generated until the interrupt has been processed.

4. Since the methods of interrupt and process creation are uniform extensions of the call statement, problems associated with the passing of parameters are avoided. The parameter-passing mechanisms are the same for all types of procedure entry. The value attribute is specified so that the value of a parameter passed to an interrupt process will be the value at the time the interrupt is generated rather than at the time the interrupt is processed.
CHAPTER BIBLIOGRAPHY

BIBLIOGRAPHY

Books


Articles


Publications of Learned Organizations


Unpublished Materials
