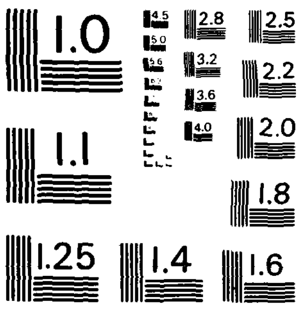
AD-A126 047 DEVELOPMENT OF A CONCURRENT TREE SEARCH PROGRAM(U) 1/2
NAVAL POSTGRADUATE SCHOOL MONTEREY CA C N POWLEY
OCT 82

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

DEVELOPMENT OF A CONCURRENT TREE SEARCH PROGRAM

by

Curt Nelson Powley

October 1982

Thesis Advisor: D. R. Smith

Approved for public release, distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A125647	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a Concurrent Tree Search Program		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis October 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Curt Nelson Powley		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE October 1982
		13. NUMBER OF PAGES 167
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Concurrency, Concurrent Programming, Parallel Programming, Tree Search, Search, Artificial Intelligence, Abstract Data Types, Data Structures, Messages		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Search, especially tree search, is fundamental to the field of artificial intelligence. Even with good heuristic functions, the time it takes on a single processor to solve progressively more difficult tree search problems grows exponentially and quickly becomes constraining. It seems reasonable that the use of concurrency should significantly improve the speed of a tree search. After discussing concurrent programming issues as background, this thesis outlines some high-level approaches to concurrent tree search.

[illegible]

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Development of each high-level approach includes development of required operating system interfaces. With the warning that choosing the best approach requires empirical evaluation, a concurrent tree search algorithm for the eight-puzzle is presented.

Development of a Concurrent Tree Search Program

by

Curt Nelson Powley
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
October 1982

Author: Curt N Powley

Approved by: [illegible]
Thesis Advisor

[illegible]
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Science and Engineering

ABSTRACT

Search, especially tree search, is fundamental to the field of artificial intelligence. Even with good heuristic functions, the time it takes on a single processor to solve progressively more difficult tree search problems grows exponentially and quickly becomes constraining. It seems reasonable that the use of concurrency should significantly improve the speed of a tree search. After discussing concurrent programming issues as background, this thesis outlines some high-level approaches to concurrent tree search. Development of each high-level approach includes development of required operating system interfaces. With the warning that choosing the best approach requires empirical evaluation, a concurrent tree search algorithm for the eight-puzzle is presented.

TABLE OF CONTENTS

I. INTRODUCTION 12

A. PROBLEM 12

B. APPROACH 12

C. ORGANIZATION OF THE THESIS 13

II. FUNDAMENTALS OF TREE SEARCH 15

A. THE EIGHT-PUZZLE 15

B. STATES 16

C. OPERATORS 17

D. GRAPHS AND STATE-SPACES 18

E. TREES 19

F. HOW TO TREE SEARCH 20

1. When a Solution is Found 20

2. Breadth - First Search 22

3. Heuristics 22

G. SEARCH MODIFICATIONS FOR A GRAPH 23

1. Cost of a Solution 23

2. Finding the Optimal Solution 25

3. Avoiding Duplication of States 26

H. CHAPTER SUMMARY 27

III. CONCURRENT PROGRAMMING ISSUES 29

A. MUTUAL EXCLUSION 30

1. The Problem 30

a. Train Example 30

b. A Computer-Oriented Example 31

2. Approaches to Mutual Exclusion Problem 32

a. Automatic Mutual Exclusion Will Not Work 32

b. Point of View is Code, Not Shared Resource; Critical Region 33

c. Enforcement of Critical Regions 34

(1) Variable Associated with the Processes 34

(2) Storage Interlock 35

(3) Variable Associated with the Resource 36

(a) Test-and-Set Instruction 36

(b) P and V, Semaphores 37

(c) Mutual Exclusion on Semaphores Themselves 41

(d) P and V Versus Test-and-Set ... 42

3. Meaning, Mutual Exclusion, and Abstract Data Types 42

a. Meaning: Sequential Versus Concurrent Programs 42

b. Separating "What" from "How" 43

c. Monitors 44

d. Abstract Data Types 45

e. Abstract Data Types and Mutual Exclusion 46

f. An Example -- Mutual Exclusion on a Priority Queue 48

g. Incorporating Mutual Exclusion in Abstract Data Types -- Recap 50

4. Mutual Exclusion Summary 50

B. PRECEDES RELATION 51

1. Example 52

2. Activity Graphs 52

3. Approaches to Precedes Type Concurrency ... 53

a. Internally Coordinated Processes 53

b. Externally Coordinated Processes 54

(1) CoBegin/CoEnd 54

(2) Precedes Relation Specifications .. 56

C. SYNCHRONIZATION 57

1. The Problem 57

a. Racetrack Example 58

2. Approaches to Synchronization 59

a. Path Expressions 59

b. Shared Variable 60

c. Eventcounts and Sequencers 61

D. COMMUNICATION BY MESSAGES 64

1. Restricted Messages 64

a. Eventcounts, Await, and Advance 64

b. Semaphores, P and V 65

c. Semaphores Versus Eventcounts 66

2. Unrestricted Messages 68

a. Categorization of Messages 69

(1) Broadcast Versus Consumable Messages 69

(2) Broadcast Messages Locally Consumable 69

(3) Broadcast Messages Queuing Versus Superceding 70

(4) Specifying Receivers of a Message . 70

(5) Blocking 70

b. A Suggested Message System 71

(1) Message Types 71

(2) Operators 72

(3) Specification of Receiver 73

(4) Syntax 73

(a) Operators 73

(b) Declarations 74

(5) Example: Multiple Producers, Multiple Consumers with Buffer 76

3. Advantages of Using Message Passing 78

E. CONCURRENT VERSUS DISJOINT PROCESSES: NONDETERMINISM 80

1. Precedes Relation Example 81

2. Mutual Exclusion Example 82

3. Optimal Eight-Puzzle Example 83

4. Non-Optimal Eight-Puzzle Example 84

5. Nondeterminism Recap 86

F. CHAPTER SUMMARY 87

IV. DEVELOPING A CONCURRENT TREE SEARCH PROGRAM 89

A. THE NEED TO CONSIDER THE UNDERLYING ARCHITECTURE 89

B. DIVISION OF TREE SEARCH AMONG PROCESSES 94

1. Shared Frontier Approach 94

2. Division of Tree Approach 94

C. MEMORY MANAGEMENT PROBLEMS 96

D. STARTING THE SEARCH 98

E. WHILE THE SEARCH IS IN PROGRESS 101

1. Promulgating the Value of Best Solutions .. 101

2. Ensuring Uniform Heuristic Distriibutiion . 105

a. Using Priorities 106

b. Distributing Frontier Nodes by Process Creation 109

c. Distributing Nodes by Passing 109

d. Detecting Hot Processes 111

e. Uniform Heuristic Distribution Summary 114

3. Perhaps a Limited Global Frontier 116

4. Memory Overflow 117

F. FINISHING THE SEARCH 119

1. Recognizing that the Search is Completed .. 119

2. Identifying the Best Solution 120

3. Outputing the Best Solution 121

G. CHAPTER SUMMARY 122

V. A CONCURRENT TREE SEARCH ALGORITHM 124

A. APPROACH 124

B. HIGH-LEVEL VIEW 125

C. ON THE STRUCTURED USE OF MESSAGES IN PROGRAMS . 127

D. ALGORITHM SYNTAX 130

E. EXPLANATION OF ALGORITHM 139

1. Outline 13

2. During Search 14

3. Start Up 14

4. Completion of Search 143

5. Solution Output 144

6. Message Declarations 147

F. CHAPTER SUMMARY 147

VI. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS 148

A. THESIS SUMMARY 148

B. BACKGROUND SUMMARY 149

C. CONCLUSION 152

D. RECOMMENDATIONS 152

APPENDIX A: IMPLEMENTATION OF A CONCURRENT PRIORITY QUEUE USING A HEAP 154

APPENDIX B: DERIVATION OF POURER/TAKER ALGORITHM WITH EVENTCOUNTS AND SEQUENCERS 161

LIST OF REFERENCES 163

INITIAL DISTRIBUTION LIST 166

LIST OF FIGURES

Figure 1. The Eight-Puzzle 16

Figure 2. An Example Eight-Puzzle Problem 16

Figure 3. P and V Implemented With Test and Set 40

Figure 4. P and V Algorithm With P and V as Critical Regions 41

Figure 5. Activity Graph 52

Figure 6. Sender/Receiver Problem Solved By Eventcounts and by Semaphores 67

Figure 7. Typical Multiprocessor System 93

Figure 8. Concurrent Tree Search Program 132

I. INTRODUCTION

A. PROBLEM

Tree search, as typified by the eight-puzzle problem, is fundamental to the field of artificial intelligence. Even with good heuristic functions, the time it takes on a single processor to solve progressively more difficult tree search problems grows exponentially and quickly becomes constraining. It seems reasonable that the use of concurrency should significantly improve the speed of a tree search.

The term "concurrency" denotes a broad range of topics. Concurrent studies are ongoing in concurrent architectures, automatic implementation of concurrency in sequential programs, and concurrent programming. This thesis deals with improving a tree search by writiing a high-level concurrent tree search program. It is implicitly assumed that the program is to be written in an imperative language such as Pascal, but most of the discussion is also applicable to applicative languages such as LISP.

B. APPROACH

A first impulse might be to write and implement a concurrent tree search program without first evaluating the adequacy of the available concurrent programming tools. A better approach is to consider if existing concurrency tools provide an adequate high-level approach for writing an effective concurrent tree search program. If they do not, then a high-level approach should be developed as part of writing the tree search program.

This thesis evaluates available concurrency tools and aproaches and finds that they are inadequate for writing a high-level tree search program which makes effective use of conventional architectures. Accordingly, different issues involved in writing a concurrent tree search are explored while at the same time considering different high-level approaches as candidate frameworks for writing the program. An algorithm is presented based on some of the tools and approaches developed, but the reader is cautioned that empirical testing is necessary to determine the best approach and the best program.

C. ORGANIZATION OF THE THESIS

In this introduction the problem has been stated and it has been placed in the context of the need to ensure an adequate high-level approach to the problem. Chapter Two explains as background the fundamental tools of tree search. As a prelude to developing a tree search program, Chapter Three discusses concurrent programming issues including available tools and several representative high-level approaches. Chapter Four states the key problems which must be solved for a concurrent tree search program to be effective on conventional architectures. In conjunction with solving the problems, high-level approaches including necessary operating system interfaces are developed. With a sound basis for understanding concurrent programming and concurrent tree search problems, Chapter Five presents a high-level algorithm for concurrent tree search. Finally, Chapter Six summarizes the thesis and puts forth conclusions and recommendations.

II. FUNDAMENTALS OF TREE SEARCH

Explained in this chapter are the basic techniques of tree search. The eight-puzzle is used as an example of tree search for this chapter as well as for the entire thesis. Nilsson's textbook, Problem Solving Methods in Artificial Intelligence [Nils71] is the reference on which this chapter is based.

Solving puzzles and games are often the subject of artificial intelligence research. As Minksy says, "It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity, so that one can engage some really formidable situations after a relatively minimal diversion into programming." [Mins68: p. 12]

A. THE EIGHT-PUZZLE

A puzzle which is frequently used as the basis for computer tree search programs is the eight-puzzle. The eight-puzzle consists of eight numbered, moveable tiles in a 3x3 frame of nine cells. See Figure 1. Since there are nine cells in the frame and eight tiles in the cells, one of the cells is always empty. The empty cell is called the blank.

Any adjacent tile can be moved into the blank, in effect "moving" the blank also.

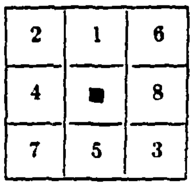
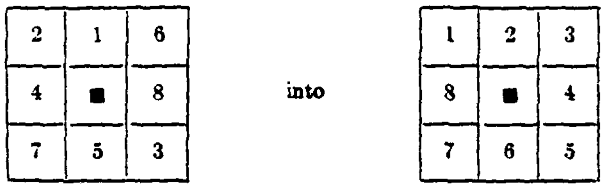


Figure 1. The Eight-Puzzle

The eight-puzzle problem is how to change an initial configuration of tiles into a goal configuration. Consider the initial and goal configuration of Figure 2. A solution to that problem would be an appropriate sequence of moves such as: "move tile 5 up, move tile 7 right, ..., etc."



Initial Goal

Figure 2. An Example Eight-Puzzle Problem

B. STATES

In developing approaches for solving arbitrary eight-puzzle problems, it helps to formalize the elements of the problem. A particular configuration of the tiles is called a state. There is a finite, although large, number of possible eight-puzzle states. An eight-puzzle problem consists of an ordered pair of states (one the initial state, one the goal state) drawn from the set of all possible states. The set of all possible eight-puzzle problems, then, is the set of all ordered pairs of states constructable from the set of all states. This set of possible problems is also finite but large.

Note that a problem in which both the initial and goal state are the same is a trivial problem, as no move is required. It should be pointed out, however, that such a trivial problem has an infinite number of solutions: any sequence of moves which changes the initial state into other states and back to the initial state is a solution. Similarly, any problem which has a solution has an infinite number of solutions. It is also worth mentioning that some problems have no solutions, i.e., it is impossible to move from certain initial states to certain goal states.

C. OPERATORS

A move which changes one state to another is called an operator. There are four operators: they have the effect of "moving" the blank up, down, left, or right (it is sometimes easier to think in terms of moving the blank rather than moving the tile; the notion is similiar to thinking of hole movement rather than electron movement in transistor theory). For a given state, not all operators may be useable. For example, if the blank is in the "upper-right corner" of the puzzle, it can only be moved left or down. In fact, the only states for which all operators are useable are those with the blank in the center tile.

An eight-puzzle problem, then, is specified as an initial state and a goal state. A solution to a problem, if one exists, is a sequence of operators which when applied to the initial state produce the goal state.

D. GRAPHS AND STATE-SPACES

From a given initial state, certain operators may be applied to produce other states. From each of these other states, operators may be similarly applied to produce more states. This process can be continued _ad infinitum_ (although only a finite number of states are reachable). It is useful to think of this potential expansion of an initial node as a graph. The graph contains nodes corresponding to states and directed arcs between the nodes corresponding to operators. Such a graph is implied for each node and it corresponds to the _space of states_ reachable from that node. Any two states which can be reached from one another (i.e., a solution exists to the problem consisting of those two states) have an identical state-space graph. It turns out that there are only two connected state-space graphs for the eight-puzzle (which implies that the set of states can be partitioned into two disjoint subsets such that any state in one subset is unreachable from a state in the other subset).

To solve an eight-puzzle problem, one applies operators to the initial state, and to states produced from the initial state until a goal state is found. This corresponds to making explicit certain parts of the implicit state-space graph associated with the initial state. Solving an eight-puzzle problem, then, corresponds to searching (making explicit) an implicit state-space graph.

E. TREES

It is helpful to think of the state-space graph as a tree. A tree is a special type of graph with the following characteristics. In a tree, there is one node, called the root node, which has no parent nodes (i.e., no directed arcs pointing to it). In addtion, all nodes, other than the root, have exactly one parent node. There is an implicit state-space tree associated with each initial state, with the root node of the tree corresponding to the initial state. Moreover, there is a unique path from the root node to each node in the tree. This path to a node corresponds to the sequence of operators (arcs) and resultant states (nodes) which transform the initial node into that node. Hence, a path from the root node to a goal node specifies a solution to the problem consisting of the root node and the goal node. Because there may be many different paths from an nitial state to any other state, the state-space associated with an initial state is not actually a tree. However, by thinking of it as a tree, the following discussion on how to search is simplified. Considerations which are necessary to account for the fact that the state space is a graph but not a tree will be discussed separately.

F. HOW TO TREE SEARCH

Searching a tree consists of applying operators to nodes reachable from the root node. The act of applying an operator to a node is called expanding a node. The node expanded is the parent node and the resultant nodes are its children. At any instant during the search of an implicit tree, there is a portion of the tree which has been made explicit. The explicit part of the tree consists of nodes which have been expanded and those which have not yet been expanded. Those nodes which have not yet been expanded are the leaf nodes, or frontier, of the explicit tree. It is from the frontier that the next node to be expanded must be chosen. Thus, the order in which nodes are chosen from the frontier determines the order of search of the implicit tree.

1. When a Solution is Found

When a goal node is found during expansion of a node, a solution has been found. The solution is the path (sequence of operators) from the root node to the goal node. In a computer program, nodes are typically represented by unique records. With this representation, which is used for this thesis, it is necessary to include within each node record some means of determining its parent, e.g., a pointer. The pointers can be followed from the goal to the root node to construct the solution. However, since the solution begins at the root node and finishes at the goal node, following pointers in the opposite direction (from the goal to the root node) traces a solution in "reverse" order. One way to put the nodes in the right order is to stack them as they are followed from goal to root node. When the nodes are unstacked they are in the right order. Note that the solution actually consists of a sequence of operators, so outputing the puzzle states does not explicitly specify a solution. For re-constructing the solution operators, it is useful to place in each node record (except the root) a representation of the operator which produced that node. In the solution output, then, the state representation of the node is preceded by the operator which produced the node. Such a solution output would produce the sequence: initial state, operator, next state, operator, next state, ..., next state, operator, goal state. Although it is only necessary to specify the operators in the solution, including the intermediate states clarifies the output.

Because the implicit state space graph associated with an initial node is extremely large, methodologies are needed to produce an effective search for a goal node.

2. Breadth-First Search

One orderly search method is a breadth-first search. The frontier of the tree is maintained as an ordered list. Initially, the frontier contains only the root node. The next node to expand is always chosen from the front of the frontier list and the children of the expanded node are always placed at the end of the frontier list. The effect of this approach is to expand all the nodes of a particular depth in the tree before expanding any deeper nodes.

3. Heuristics

In a breadth-first search, all the nodes at depths less than the goal node are explored. As a result, the number of nodes to be explored grows exponentially with increasing depth of the goal node. This implies that time and space requirements for a single processor become constraining for difficult problems. The problem with breadth-first search is that the tree is explored blindly with no intelligence involved in deciding which nodes of the frontier to expand next. A way to improve the search is to associate with each node a value, called the heuristic value, which reflects the liklihood that the node is part of the solution path. By ordering the frontier by heuristic value of nodes, the tree can be explored in a more intelligent order. The heuristic value is estimated by applying a heuristic function to the state of a node. The lower the heuristic value of a node is, the greater is the liklihood that a node is on the solution path. A simple example of a heuristic function is one which counts the number of tiles that are out of place with respect to the tiles of the goal node. It is stressed that heuristic functions are estimates, and thus may sometimes be misleading. If a candidate heuristic function reduces the search effort from that required for a blind search, then it is worth using.

G. SEARCH MODIFICATIONS FOR A GRAPH

1. Cost of a Solution

In a tree, there is only one path from the root node to the goal node. But in a graph there are many paths and thus many solutions. When there is more than one solution, it is helpful to compare them. A cost is associated with each solution and a solution with the lowest cost is an optimal solution. In the eight-puzzle, cost may be simply defined as the number of operators in the solution. In a graph, this is equivalent to the number of arcs in the solution path. Thus a solution of eight moves has a lower cost than a solution of ten moves.

When there is more than one possible solution, it is desirable to order the frontier by the estimated cost of each frontier node. By "cost" of a frontier node is meant the cost of the best solution constrained to go through that node. It can be seen that this cost is the sum of two components: the length of the shortest path from the root node to the frontier node under consideration, and the length of the shortest path from the frontier node to the goal node. The first component of the cost can be estimated as the length of the path followed in producing the frontier node. It is only an estimate in a graph because there may be a shorter path. Since the best path can be no worse than a path already found, this estimate is an upper bound on the length of the shortest path from the root node to the frontier node. This estimate can be inserted into a node record when it is created: the estimate for a child is simply the estimate of the parent plus one.

The second component of the cost of a frontier node is the length of the best path from the node to the goal node. This cost can be estimated with a heuristic function similiar to that discussed earlier.

Using Nilsson's notation [Nils71: pp. 57-59], let f, g, and h be functions such that f(n) = the cost of the best solution constrained to go through node n, g(n) = cost of the best path from the root node to node n, and h(n) = the cost of the best path from node n to the goal node. Then

$$f(n) = g(n) + h(n)$$

Let the "hat" character "^" placed over a letter representing a cost denote an estimate of that cost. Then an estimate of the cost of the best solution constrained to go through node n is

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

The function $\hat{f}$, then, is composed of the length of the best path already found from the root node to n plus the heuristic estimate of the best path from n to the goal node. $\hat{F}$ is called the <u>evaluation function</u> and can be used to order the frontier nodes so that the next node is the one most likely to be on an optimal solution path.

2. Finding the Optimal Solution

When a solution is found in a tree, it must be the optimal solution because there is only one solution in a tree. When a solution is found in a graph, it may not be optimal since there are many solutions (optimal and non-optimal) in a graph. Sometimes, it is sufficient to find any solution, but more often it is desirable to find an optimal solution. Since the existence of one solution in a graph implies that there are an infinite number of solutions, some way is needed to control the search for an optimal solution. In a search which uses the evaluation function $\hat{f}$ to order the frontier, the search can be controlled by updating the frontier each time a solution is found. The update consists of throwing away all nodes in the frontier with a higher estimated cost than the actual cost of the best solution already found. The search ends when the frontier is empty.

The problem with updating the frontier by discarding nodes with a higher estimated cost than a solution found is that an optimal solution is not guaranteed. A frontier node on an optimal path will be discarded if the evaluation function estimates its cost to be higher than a solution already found. It seems intuitively correct that if an evaluation function never over-estimates the cost of a path from a frontier node to the goal node, this won't happen. That is, if h is a *lower bound* on h, an optimal solution is guaranteed to be found. This intuitive conclusion is correct and a proof can be found in Nils71 (pp. 59-61).

3. Avoiding Duplication of States

Another consideration for searching a graph is to avoid creating different records which represent the same state. In a graph, because there is more than one path to a node, the same node may be found on different paths. If a node is found which has been found before, only the node record representing the better solution path should be kept.

To determine if the state of a node has been previously found, it is necessary to check all node records which have been created. The frontier list contains those node records which have been created but not expanded. It is also necessary to maintain a list of all node records which have already been expanded. Nilsson calls this list the closed list [Nils71: p. 48]. When creating a new node record, then, both the frontier list and closed list are checked to see if a node record with the same state exists. If one does, the node record with the best associated cost is kept.

H. CHAPTER SUMMARY

The term "tree search" is used loosely in the rest of this thesis to refer to graph search. The type of tree search being considered is typified by the eight-puzzle problem. Thus, this thesis does not apply directly to AND/OR graph searches (which are also described in Nils71).

A good example of a puzzle requiring tree search is the eight-puzzle. Configurations of the puzzle are represented by states. Operators are used to map one state to another state. Searching for a solution to an eight-puzzle problem is associated with making explicit parts of an implicit state-space tree. The use of heuristics significantly reduces the time and space required to solve such problems. If any solution to an eight-puzzle problem exists, then numerous solutions exist and it is desirable to find an optimal solution. Using an evaluation function which contains a heuristic function that is a lower bound on the h component of cost guarantees that an optimal solution will be found.

III. CONCURRENT PROGRAMMING ISSUES

The notion of concurrent processes is an outgrowth of operating system design. Solutions to concurrent programming problems have only recently begun to incorporate high level tools and approaches suitable for applications programming rather than systems programming. An overview of some of the key issues and techniques in concurrent programming is presented in this chapter. The problem of mutual exclusion is discussed first since it is a fundamental issue and illustrates some key differences between concurrent and sequential programming. Although solving the mutual exclusion problem paved the way for development of concurrent programming tools, mutual exclusion should be viewed as a low level tool upon which more sophisticated approaches can be based. With this in mind, the next two sections present high level approaches to problems which both involve ordering of events: precedes relation and synchronization. Some tools, such as the use of eventcounts and sequencers, can be used to implement these orderings and are discussed. Section four discusses communication by messages as a fundamental concurrency tool. A framework for categorizing message types is developed and the syntax for a message system is introduced. Lastly, the differences between sequential and concurrent programs are further illustrated by considering the degrees of non-determinism which can be found in concurrent programs.

It is assumed that the reader is familiar with the basic notions of process and resource. Definitions for these can be found in Cali82. In this thesis, the term "concurrent" means "overlapping in time", whereas the term "simultaneous" means actually occurring at the same time".

A. MUTUAL EXCLUSION

1. The Problem

a. Train Example

The following example introduces the notion of mutual exclusion. Two trains have separate routes except for one small section of shared track. Obviously, a train must have exclusive access to the section of track it is on at any given time; i.e., all trains must mutually exclude each other in their use of track. (The track is a resource which allows only one user at a time; there are other resources such as movie screen, which allow more than one simultaneous user.)

The trains are running asynchronously--the speed of their engines may vary (even stop) at random. To prevent a collision, the trains must somehow be synchronized with respect to the shared section of track. That is, at least some synchronization must be introduced into an asynchronous situation.

The essence of the mutual exclusion problem is to somehow prevent simultaneous use of a shared resource. That two trains should not collide is obvious. The following computer example illustrates a non-obvious need to enforce mutual exclusion.

b. A Computer-Oriented Example

Two processes are running at asynchronous speeds. Each process includes the statement V <-- V + 1, where V is a shared variable (resource). It is clear that the intent of V <-- V + 1 is that after the statement's execution the value of V should be one greater than had the statement not been executed. But, this intent may not be realized. For a conventional computer, the high level statement V <-- V + 1 will be translated into several machine instructions. The value of V is read, incremented, and then the new value is stored into V. When two processes execute the machine instructions of V <-- V + 1 concurrently, they may both read and increment before storing. (The undesired result is to lose one of the increments). Such a possibility indicates the need to ensure that the several actions of the statement V <-- V + 1 are inseparable,. That is, the processes must mutually exclude one another during the statement execution. But this conclusion is not obvious. Since the meaning of V <-- V + 1 is obvious, why should the meaning not be realized when the statement is executed concurrently? This question will be discussed at a later point.

A situation as described above in which the outcome may vary improperly due to changes in the relative speeds of processes is called <u>race</u>. (It should be mentioned that the term "race" is used inconsistently in the literature. Sometimes it is used to mean a situation which can vary (properly or improperly) due to changes in relative process speeds, rather than a situation which varies improperly. Improper variance is a vague notion because the properness of a situation is relative to the intent of the programmer. However, the term race will imply improper variance in this thesis because this usage better fits the spirit of the discussion.)

2. Approaches to the Mutual Exclusion Problem

a. Automatic Mutual Exclusion Will Not Work

A first solution to the mutual exclusion problem might be to automatically detect accesses to shared resources and ensure that such accesses are serialized in time. For example, a compiler could easily detect that a statement V <-- V + 1 involves a shared variable and then ensure that the execution of this statement mutually excludes any other statements involving the shared variable (how the mutual exclusion can be implemented has not yet been discussed). But, just as the statement V <-- V + 1 involves several inseparable actions, what if this statement were embedded in a group of inseparable statements? Suppose that V is an array of size N, the statements

```
DO FOR i <-- 1 to N
  V(i) <-- V(i) + 1
END DO
```

might have been written with the intent of updating the entire array at once. If another process were allowed to access the array in the midst of an update, it would read an unintended statement.

One might be tempted to let the compiler check for such "DO FOR" loops and ensure mutual exclusion on the loop. But, what if the loop were embedded in another loop? At what level does the compiler stop? Arguments for automatic enforcement of mutual exclusion miss the point. Although the need for mutual exclusion can be inferred, which section of the code needs to be mutually exclusive depends on the programmer's intent, i.e., the purpose of the code. This intent is not detectable.

b. Point of View is Code, Not Shared Resource; Critical Region

It is worthwhile to explicitly discuss something which until now has only been implied. Mutual exclusion is always associated with some shared resource. Therefore, it is tempting to view the mutual exclusion problem solely from the point of view of the shared resource and assert that as long as no two or more processes simultaneously use the resource, the mutual exclusion need has been satisfied. But, as illustrated by the V <-- V + 1 example, the mutual exclusion involves certain "entire uses" of a resource. That is, the mutual exclusion involves sections of code within the processes that use the resource. These sections of code for which mutual exclusion is required are called critical regions. Again, note that critical regions are associated with shared resources; a critical region does not refer to a section of the shared resource. Thus, a definition of a critical region can be expressed as follows:

Critical Region - A group of actions involving one or more shared resources such that the group of actions must be indivisible with respect to some other actions involving the same resources.

c. Enforcement of Critical Regions

(1) Variable Associated with the Proccesses. Consider two processes which have critical regions associated with a shared variable. A first attempt at enforcing mutual exclusion for these critical regions might be for each process to have a boolean variable which it changes to true if that process is in its critical section, or to false otherwise. These boolean variables are shared between two processes. A process enters its critical section only after

ensuring that the other processes' boolean variable is false (implying that the other process is outside its critical section). The problem with this attempted solution is the same one that occurred in the V <-- V + 1 example: the actions of getting information and taking action based on that information are separable. Both processes can read that the other is outside its critical section and then each enter its critical region. Using variables associated with the processes, Th. J. Dekker [Dijk68A] found a solution to the critical region problem. Although inelegant, this solution paved the way for better approaches by demonstrating that the mutual exclusion problem was solvable.

(2) Storage Interlock. The astute reader will have raised another question about the solution discussed above. Will not simultaneous read and write accesses to the shared boolean variables cause race as it did in the V <-- V + 1 example? In that example, race was caused because the V <-- V + 1 statement was translated into several machine instructions. The machine instructions could be interleaved with other machine instructions; however single machine instructions such as read and write cannot be divided. At the lowest level, hardware provides this type of mutual exclusion called storage interlock [Cali82]. Storage interlock ensures only one machine instruction at a time can access a shared variable, hence, simultaneous access is

impossible. Computer-oriented solutions to the mutual exclusion problem rely eventually on storage interlock.

(3) Variable Associated with the Resource. Dekker's solution [Dijk68A] to the mutual exclusion problem was based on variables associated with each processes' critical region. By changing the point of view from that of the processes' critical region to that of the shared resource, the solution becomes more elegant.

(a) Test and Set Instruction. Since critical regions are associated with a shared resource (or group of resources), it makes sense to associate a shared variable with the resource. This variable can be viewed as a door to a critical region involving the resources associated with the door. Either the door is open or shut. A process checks the position of the door until it is open, then it shuts the door and enters its critical section. The problem, again, is the separation of checking a value and changing it.

The Test-and-Set (TS), instruction introduced by IBM, provides a hardware solution to this problem. In one indivisible instruction, it reads the value of a variable and writes another value to it. In effect, with this instruction, a process can read the position of the "door" and, as part of the same action, shut it (whether it was initially shut or open). If the door was open, the process enters its critical section; otherwise it attempts to "test and set" the value of the door again. Let the door variable be a boolean variable called D, where D equal to true corresponds to open and D equal to false corresponds to shut. Assume TS(D) reads D and changes it to false. Mutual exclusion of a critical section is enforced by the algorithm:

```
REPEAT                      /Loop until door is open/
  door-open <-- TS(D)
UNTIL door-open
/Critical Region/
D <-- TRUE                  /Re-open door for another process/
```

D is initialized to true (door open) to allow one of the processes to initially enter its critical section.

This algorithm is a solution to the mutual exclusion problem. It is based on the hardware mutual exclusion of both the test-and-set and the assignment (D <-- true) instructions. A disadvantage of this solution is that processes waiting to enter their critical section must repeatedly check the value of the door variable. This unproductive repetition is known as busy waiting.

(b) P and V, Semaphores. The nature of the Test-and-Set solution to mutual exclusion of critical sections can be summarized as follows: at the entry point to the critical section, a process is blocked from entering until the "door" is "open"; when a process leaves its critical section, it re-opens the door. Note the connection between exiting and entering. When one process exits, precisely one other is allowed to enter. The exiting process communicates with the entering process by changing the position of the door. The entering process receives this information by continually checking the position of the door (busy-waiting).

Thus, when a process initially tries to enter its critical section, either it can enter immediately or it must wait until an exiting process communicates that the door is open. To eliminate busy waiting, it would be desirable to have the entering process "go to sleep" if the door was not open, and have an exiting process directly signal one of the sleeping processes to "wake-up". Dijkstra's P and V operations [Dijk68A] allow exactly that.

P and V operate on a semaphore, which can be thought of as a shared integer initialized to some value. A binary semaphore, as opposed to a general semaphore, is one which can take on values of only zero or one. For mutual exclusion, only a binary semaphore is needed. Accordingly, the more general types of semaphore will not be discussed.

Several operating system actions must be understood before proceeding with the discussion on P and V. Blocking a process means that the operating system has taken the name of the process off the list of processes which are eligible to be run. That is, when a process is blocked, it will not be allocated any computer time. Waking up a process means that the operating system has placed the name of the process back on the list of processes which are eligible to be run.

The semaphore represents the door to a critical section. It is initialized to one, indicating an open door. Similiarly, a semaphore equal to zero represents a shut door. A process entering its critical section executes P(semaphore). The effect of the P operation is to block the process if the door was shut (semaphore = 0) or, if the door was open, to shut the door and allow the process to continue. A process exiting its critical region signals one blocked process to wake-up by executing V(semaphore). The effect of the V operation is to wake-up any process waiting to enter its critical region and allow it to proceed. If no processes are blocked, the V operation re-opens the door (semaphore changed to 1).

Algorithmically, the effect of P and V can be shown as follows:

```
P(sem)

 IF sem = 1 THEN
    sem <-- 0
    /enter critical region/
 ELSE
    /wait for a V operation
    to proceed (Block Process)/
 END IF
```

```
V(sem)

 IF /there are no waiting processes/ THEN
    sem <-- 1
 ELSE
    /pick a process and tell it
    to proceed (Wake-up Process)
 END IF
```

Because blocking of a process and waking up a blocked process are operating system functions, both the P and V operations are closely coupled with the operating system.

Enforcement of a critical section with P and V can be simply stated as

```
P(door)
   /critical region/
V(door)
```

where door is a semaphore initialized to one.

Note that P and V have the same _external_ appearance as Test-and-Set. One could implement P and V as shown in the following figure.

```
P(sem)                                     V(sem)

  REPEAT                                   door-position <-- false
    door-position <-- TS(sem)
  UNTIL door-position
```

Figure 3. P and V Implemented With Test and Set

But this version of P and V does not actually block processes waiting to enter their critical sections. A more efficient implementation of P and V requires operating system process management to block and wake-up processes.

(c) Mutual Exclusion on Semaphores Themselves. The astute reader will note that the P and V procedures themselves need to be treated as critical regions. For example, if two processes execute P(sem) and find sem = 1 before either changes the value of sem to zero, both will enter their critical sections. One solution is to have a "door" variable called PV-door associated with both the P and V procedures. Figure 4 shows a correct implementation of P and V using Test-and-Set.

```
P(sem)
    REPEAT
        PV-door-open <-- TS(PV-door)
    UNTIL PV-door-open.

    IF sem = 1 THEN
        sem <-- 0
        / enter critical region/
    ELSE
        /put myself on list of waiting processes/
    END IF
    PV-door <-- true / allow other execution of P and V/

V(sem)
    REPEAT
        PV-door-open <-- TS(PV-door)
    UNTIL PV-door open

    IF /there are no waiting processes/ THEN
        sem <-- 1
    ELSE
        /pick a waiting  process and allow it to proceed
        (wake-up process)/
    END IF
    PV-door-open <-- true.
```

Figure 4. P and V Algorithm With P and V as Critical Regions

(d) P and V Versus Test-and-Set. One might ask why the more complex algorithm of Figure 4 would be used rather than the simpler algorithm of Figure 3; both involve the test-and-set instruction and, thence, busy-wait. Consider just the P operations shown in Figures 3 and 4. The difference between them is that the busy-wait of Figure 3 involves busy-waiting the entire time another process is in its critical region, while the busy-wait of Figure 4 involves just busy-waiting to execute the P procedure--once executed the process is blocked waiting for another process to finish its critical region. A busy-wait for the use of P could be considerably shorter than a busy-wait for execution of a critical region. The V operation of Figure 3 requires no busy-wait because it consists of only one indivisible storage instruction, whereas the V operation of Figure 4 does require a busy-wait. However, the potential length of the busy-wait just due to the P operation of Figure 3 is much greater than the sum of the potential busy-waits of both the P and V operations of Figure 4.

3. Meaning, Mutual Exclusion, and Abstract Data Types

a. Meaning--Sequential vs. Concurr^nt Programs

In the introduction to the mutual exclusion problem, the question was asked: Since the meaning of a statement such as V <-- V + 1 is obvious, why should the meaning not be realized in concurrent execution? In other words, what differences between concurrent and sequential programs make concurrent programming so difficult? This question will now be addressed.

In a sequential program, the meaning of statements is implicitly and closely coupled with the statements. The meaning of V <-- V + 1 is obvious. The compiler translates V <-- V + 1 into several machine instructions. When these instructions are executed inseparably, i.e., without being interleaved with other instructions involving the variable V, the meaning of the statement V <-- V + 1 is preserved. If the meaning of these three machine instructions is considered separately rather than as a group, the meaning of the higher level statement is lost--it is not preserved. Thus, the meaning that was implied in the environment of a sequential program may not be preserved when the program is executed concurrently with other programs.

b. Separating "What" from "How"

How can high-level access to shared resources be structured so the program is clearer? It is useful to have a method for specifying what is done and then separating what is done from how it is done. That is, separate "what" from "how". A good approach is to use single operations to access shared resources. An operation would typicaly be a procedure call; the procedure performs the desired action and enforces mutual exclusion at a lower level. To make the meaning of the operations clear, a set of associated specifications should exist. Specifying exactly what an operator/procedure does can be difficult. The specification must be precise and the programmer using these lower level procedures must ensure that his/her intended meaning matches the specified meaning.

For example, V <-- V + 1 could be performed by

```
A <-- 1
ADD (V,A)
```

where the specification for ADD indicates that after its execution V will be A greater than it would have been had the ADD not been called. The procedure ADD can then ensure, at a lower level, that its meaning is enforced within the context of the compiler, hardware, and operating system.

c. Monitors

Closely related to the above approach is the concept of monitors [Hoar74]. In a monitor, all concurrent operations are grouped together with the data structures affected. An advantage of "textual grouping of critical regions with the data they update" [Cali82] is the ease of comprehending the correctness of the concurrent portions of the program.

In a monitor, however, only one of the procedures may be executed at a time (each procedure in a monitor is a critical region). There may be situations where it is desirable to allow concurrency between the procedures in a monitor. One may introduce complexity by allowing this, but the complexity would occur at a lower heirarchical level and might be offset by an increase in efficiency. An example which allows concurrent operations on a data structure is discussed later.

d. Abstract Data Types

The notion of separating "what" from "how" is not a new one. In the design of data structures, for example, it is an excellent and well-known technique to separate the specifiction of an abstract data type from its implementation. A data structure such as a priority queue is specified in terms of a set of axioms. The high-level view of the data structure is completely formed by these axioms.

Higher level programs use operators defined by the axioms to create and access the data structure, but the implementation of the data structure is hidden to them. This allows flexibility: the implementation of the data structure can be changed without changing the higher level program; and the higher level program need not worry about the lower level details. Splitting the implementation from the realization of the data structure also makes correctness proofs about the data structure and program easier.

e. Abstract Data Types and Mutual Exclusion

As explained previously, mutually exclusive access to shared data structures should be done via high-level operators. In turn, access to data structures should be done via high-level operators which present an abstract data structure view. How should mutual exclusion operations be related to abstract data structure operations? This section considers that question.

One approach is for the user of the abstract data type to implement mutual exclusion in terms of the abstract data type. That is, mutual exclusion is enforced with respect to the operations provided by the axioms of the abstract data type. Operations requiring mutual exclusion are implemented as a call to a subprogram; the subprogram establishes a critical region around operations which access the abstract data type. Thus, for this approach, the user of the abstract data type is responsible for providing mutual exclusion and mutual exclusion is provided in terms of the abstract data type.

Another approach is to incorporate mutual exclusion within the abstract data type. That way, the user of the abstract data type would not implement mutual exclusion; rather, operators on the abstract data type would incorporate mutual exclusion. Within this approach, the implementor of the abstract data type has two approaches to incorporating mutual exclusion. The simple approach is to treat each operation as a critical region. This approach has the same effect as implementing mutual exclusion in terms of the abstract data type. The difference is in who is implementing the mutual exclusion--the user of the abstract data type or the implementor of the abstract data type.

Rather than treat an entire operation as a critical region, the abstract data type implementor may choose a more refined approach and treat only portions of operations as critical regions. Consider a data structure operation which may take many steps, such as an insertion into a priority queue. If insertions could be done concurrently, perhaps by locking just portions of the data structure, the potential for a bottleneck of numerous processes doing insertions could be reduced. However, implementing mutual exclusion at this low a level may require additional specifications in the axioms for the abstract data structure. For example, it may be that concurrent insertions to a priority queue can be done, but deletions cannot be done concurrently with insertions. This introduces a new notion into abstract data types--that of including mutual exclusion specifications in the axioms for the abstract data structure.

A disadvantage of implementing mutual exclusion "internally" to the data structure is that the mutual exclusion becomes coupled with the data structure; one cannot change the data structure implementation without affecting the mutual exclusion. Another disadvantage is that the problem becomes more complex.

f. An Example--Mutual Exclusion on a Priority Queue

The need for including mutual exclusion specifications in the axioms of an abstract data type can be made clearer by an example. In a tree search problem such as the eight-puzzle, the unexpanded nodes of the tree (the leaves, or _frontier_) need to be kept in order of next to expand. An abstract data structure which does this is the priority queue, a queue in which the "best" value is always available for removal from the queue. Important operations on the priority queue are removal of the best item for expansion and insertion of new items which have just been generated. One approach to a tree search problem such as the eight puzzle is to have the frontier of the tree be a shared priority queue which is accessed by many concurrent processes. Each process removes a node from the priority-queue, expands it, checks for a goal node, and inserts non-goal nodes into the priority queue. It is easy to see that there is potential for an enormous bottleneck of processes at the priority queue. If operations on the priority queue take a significant number of steps and each operation is a critical region, many processes may spend considerable time waiting to insert or remove nodes; and if each process is assigned its own processor the use of the processors can become inefficient. What if there were a way to "lock" only small sections of the priority queue during each step of an insertion or deletion so that concurrent insertions could be correctly done? This might significantly improve the tree-search program.

Appendix A discusses a means for doing this using the heap data structure implementation of a priority-queue. This example illustrates the need for detailed specifications of the mutual exclusion characteristics. In this case, the specifications would include: any number of concurrent insertions are allowed, but only one removal at a time may take place. The important point is that a programmer using this priority-queue as a concurrent data structure need not worry about the data structure implementation and need not worry about mutual exclusion except to ensure that the specifications of the data structure satisfy the program's needs.

If one tries to "optimize" mutual exclusion by using the data structure implementation, another interesting consideration arises. One should now consider which data structures are best for concurrent access. For example, a heap structure worked fairly well for concurrent access, whereas another structure might not. Conveniently, a heap is also optimal for speed of access (it is much better than, e.g., a linked list). However, it might be that some data structures are good for speed of operation but poor for concurrent access, or vice-versa. Picking the best data structure may become a more demanding exercise.

g. Incorporating Mutual Exclusion in Abstract Data Types--Recap

Incorporating mutual exclusion in abstract data types provides useful hierarchial structuring. The user of the abstract data type is freed from implementing mutual exclusion, and the mutual exclusion implementation is placed at a lower hierarchial level. In addition, it allows the implementor of the abstract data type the flexibility to increase the efficiency of mutual exclusion by "refining" critical regions so they encompass less than entire operations. Allowing this flexibility introduces a new notion into abstract data types -- including mutual exclusion specifications in the axioms.

4. Mutual Exclusion Summary

Understanding the mutual exclusion problem is necessary for concurrent programming. However, the use of mutual exclusion to control access to a shared resource causes a bottleneck: only one process at a time may be in a critical region associated with a shared resource;

other processes must wait unproductively to enter their critical regions. Sometimes, the bottleneck can be reduced by "refining" the mutual exclusion so that abstract data type operations can be done concurrently. Better yet, problems can often be structured so that mutual exclusion is not required. A synchronization approach orders processes with respect to a shared resource so mutual exclusion is unnecessary. In place of using shared resources, message passing can sometimes be used for communication between processes. (Synchronization and message passing are discussed later.)

It is the author's belief that, as a general rule, the use of mutual exclusion should be avoided if another equally effective approach exists. If mutual exclusion is used, it should preferably be implemented at a lower hierarchial level and preferably be incorporated into abstract data types.

B. PRECEDES RELATION

This section and the next each present a high level structuring approach for a concurrent program which is appropriate for certain types of problems. One of the simplest types of concurrency programs is one which can be structured in terms of a precedes relation. The basic notion of a precedes relation situation can be illustrated by an example.

1. Example

Several groups of people, such as plumbers and electricians, are building a house. Each group is doing different jobs, but they have a common interface of the house they are building. The jobs have certain time-ordering restrictions. For example, interior plumbing and electrical must be done before walls are installed. That is, certain activities must precede others.

2. Activity Graphs

The operations research tool of an activity graph uses a network diagram which usually depicts exactly this type of situation. A typical activity graph is shown in Figure 5.

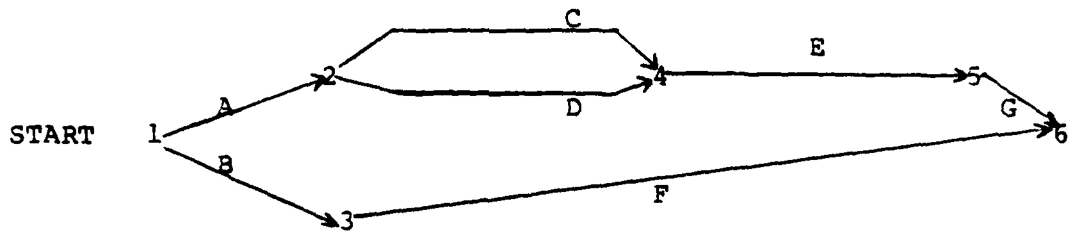


Figure 5. Activity Graph

This chart represents the activities (tasks) involved in some project (such as painting a house), and shows the required ordering between the activities. The activities are represented by the branches between nodes and are labelled by letters. Each node is called an event and represents the accomplishment of activities preceding that event. Arrowheads on the branches indicate the sequences in which the events must occur. Thus, event 4 represents the accomplishment of activities of A, C, and D. Moreover, event 4 must precede activity E. Activities C and D are not ordered with respect to one another and may proceed concurrently. In a computer program, the activities shown on the chart could be processes. The entire chart would represent a computer program consisting of a group of processes with certain ordering restrictions. In the simplest case, the processes allowed to run concurrently would be disjoint, i.e., they would share no resources.

3. Approaches to Precedes Type Concurrency

To coordinate groups of people such as those building a house, there are two basic approaches. One is to have the groups coordinate among themselves, e.g., a group knowing what groups must precede it could wait for signals from those groups indicating they were done. The second approach is to put someone external to the groups in charge of orchestrating them. That person would tell groups when to proceed. These two approaches also apply to a computer program. The former approach will be discussed first.

a. Internally Coordinated Processes

There are different methods for processes to communicate among themselves. Message-passing and communication by a shared variable (which requires mutual exclusion) are two types. Either of these methods could be used to order a precedes situation. Details of the methods will be discussed in the next several sections.

The primary disadvantages of the communication approach to a precedes relation situation are twofold. First, it requires that the code for the synchronization of the processes be spread out textually. This scattering of code makes it difficult to comprehend what is going on; thus, troubleshooting as well as writing the code is difficult. Second, the communication requirement integrates the burden of concurrency control into the high level code. A recurring theme in concurrent programming is to remove, as much as possible, the concurrency requirements from high level code. Placing the code which coordinates the processes at a lower level and/or grouping it in one place makes the programming job much easier.

b. Externally Coordinated Processes

The preferred approach, then, to coordinating precedes type processes is by viewing them externally.

(1) CoBegin/CoEnd. A single construct for doing this is Dijkstra's CoBegin/CoEnd construct introduced as ParBegin/ParEnd [Dijk68A].

For example:

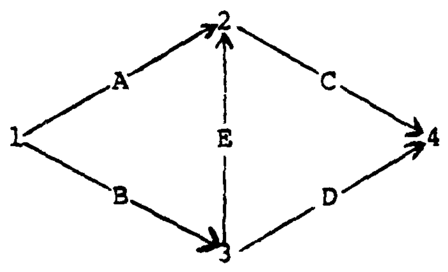
```
S1
CoBegin
   A; B:
CoEnd
S2
```

means A and B can proceed concurrently; S1 must precede both A and B, and A and B must both precede S2.

To represent more complicated situations, nesting of CoBegin/CoEnd groups is used. The relations required by Figure 5 are expressed as:

```
CoBegin
    Begin
    A; CoBegin C; D; CoEnd; E; G;
    End
    Begin
    B; F;
    End
CoEnd
```

A disadvantage of the CoBegin/CoEnd approach is that complex situations become confusing to write or read. Moreover, the CoBegin/CoEnd construct is not powerful enough to specify some situations. For example, the following activity graph cannot be specified by CoBegin/CoEnd:



The "problem" activity in this graph is E. E can proceed concurrently with A and D, but this cannot be speciified in addition to specifyiing the constraints between the other activities.

A construct which is general enough to express the preceeding activity graph is the FORK/JOIN command pair described in Conw63. FORK/JOIN will not be described.

(2) Precedes Relation Specifications. The term "precedes relation" has been used because a simple relation called precedes can be used to fully specify situations of the type being discussed. A precedes relation consists of a set of tuples. Each tuple has two elements where each element is a group of names of processes. The meaning of each tuple is simple: the group of processes specified in the first element must precede the group of processes in the second element.

Consider Figure 5 again. The required ordering of activities (processes) can be completely specified by a precedes relation. The only restriction in the processes are that some must precede each other. Figure 5 is completely specified by the following precede relation:

{((A), (C, D)), ((C, D), (E)), ((E), (G)), ((B), (F))}

Note that additional tuples such as ((A), (E)) could be added but are not necessary. The precedes relation is transitive;

thus ((A), (E)) is implied by ((A), (C, D)) and ((C, D), (E)). Additionally, it is implied that A and B are the starting processes because nothing precedes A or B. It is similarly implied that G and F are the ending processes. To make the beginning and ending processes more obvious, it is useful to incorporate dummy processes called start and finish and to include ((Start), (A, B)) and ((G, F), (Finish)) in the relation.

The proposed precedes relation specification, like the CoBegin/CoEnd construct, has the disadvantage that it may be confusing to read.

C. SYNCHRONIZATION

1. The Problem

The term synchronization is used in a broad sense to describe any type of concurrency issue. In this section, the term is used to mean a way of structuring a problem. A synchronization approach to a problem involves implementing an ordering of processes with respect to shared resources, rather than defending the resources directly by mutual exclusion. The precedes relation situation discussed in the previous section can be viewed as a simple type of synchronization involving no shared resources.

a. Racetrack Example

Although contrived, the following example illustrates the notion of synchronization (it is a "producer-consumer" type problem). On a circular racetrack, one gallon buckets are placed at regular intervals. Initially, the buckets are full (one gallon) of water. Two workers, a pourer and a taker, are assigned. The pourer's job is to continuously walk around the track, putting one gallon of water in each bucket. The taker's job is to continuously walk around the track, taking a gallon of water from each bucket.

Because the two workers are using the same buckets (shared resources), their actions affect each other. Initially, when the buckets are all full, the taker can go around the track just once before needing the pourer to put water in the buckets. Similiarly, the pourer cannot start until the taker has started.

What are some approaches to coordinating these people? One approach is mutual exclusion. The pourer and taker could mutually exclude each other in their use of the racetrack. For example, starting with the taker, the two could alternately make one entire trip around the racetrack. A less restrictive mutual exclusion approach would be to allow just one person at a time to access a bucket. In either case, some method is required to prevent taking from an empty bucket or pouring into a full one.

The nature of mutual exclusion approaches such as the ones above is to cause waiting by one person for another person because of a shared resource. From the point of view of not wasting the people's time, a more efficient approach is to synchronize the people so they can work concurrently on different buckets. The taker could simply follow the giver forever. Their speeds don't have to be identical; the restriction is that they don't pass each other. Keeping the workers the required number of buckets apart without relying on mutual exclusion is the essence of the synchronization approach to this problem.

2. Approaches to Synchronization

a. Path Expressions

Path expressions are regular expressions which describe the allowable ordering of operations with respect to a shared object. The motivation for path expressions is to provide high level tools for concurrency control. The programmer does not have to worry directly about synchronization primitives, but instead just specifies the allowable sequences of operations on a shared resource. This approach is similiar to the precedes-relation approach, except for the viewpoint. Path expressions are relative to shared objects, whereas a precedes-relation is relative to a group of processes. A simple solution to the racetrack/water-bucket problem using a path expression would specify that for each bucket a pourer must follow a taker, and that this order may be repeated forever. (This is expressed as (P;T)* where P represents pourer, T represents taker, * means "zero or more times", and the expression is applicable to each bucket).

A significant advantage of the path expression approach is that it requires no synchronization control in the code of processes. The disadvantage of path expressions is that more complex ordering restrictions are hard to specify. For example, if there were multiple pourers and takers in the racetrack problem, this approach would not work because each worker would not know what bucket to access next.

b. Shared Variable

The restriction in the racetrack problem is that the difference between the number of pours and takes stays within alowed bounds. This restriction is easily enforced in a computer solution by associating with each process a shared variable recording the number of operations done by that process. Before pouring water, the pourer process checks the difference between the total number of operations it's done and the total number of operations the taker process has done. If the number is not within allowed bounds, the pourer process waits until it is. The taker process does a similiar thing. Although this solution does not require mutual exclusion of the water buckets, it does require mutual exclusion of the shared variables. Furthermore, it requires synchronization within the code of the processes.

c. Eventcounts and Sequencers

Eventcounts and sequencers were developed to allow a better solution to the type of synchronization problem being disucssed. The primitive data types and associated operations are:

	NAME	DESCRIPTION
DATA TYPE	Eventcount	Non-decreasing, non-negative integer
	Advance(E)	Signals occurence of an event by incrementing eventcount E
OPERATIONS	Await(E,V)	Blocks process until eventcount E reaches value of V
	Read(E)	Reads value of eventcount E
DATA TYPE	Sequencer	Non-decreasing, non-negative integer
OPERATION	Ticket(S)	Returns a unique "Ticket" number; used to order processes

The solution to the racetrack problem using eventcounts is similiar to the solution using shared variables. It is a better solution because the synchronization is done with higher level primitives which do not rely on any mutual exclusion. The pourer has associated with it an eventcount, P, indicating how many pours it has completed. It has a local variable, i, which also represents how many pours it has completed. The taker has similiar variables. If N is the number of buckets on the racetrack (or slots in an array for the typical producer-consumer scenario), then the solution is given by the following algorithms:

```
POURER
    i <-- 0 / i = number of completed pours/
    DO FOREVER
        Await (T, i + 1)
        Advance (P); i <-- i + 1
    END DO
TAKER
    i <-- 0 / i = number of completed takes/
    DO FOREVER
        Await (P, (i + 1) - N)
        Advance (T); i <-- i + 1
    END DO
```

A derivation of this algorithm is given in Appendix B.

Note the symmetry of this solution: A process writes to an eventcount which the other process reads; a process reads an eventcount to which the other process writes. Thus there is no write competition.

An important feature of the eventcount solution is that all operations by the pourer and taker can be done concurrently. Their operations have been synchronized so that mutual exclusion is unnecessary.

Sequencers and tickets are used to force an ordering of events. If there were multiple takers following the pourer around the racetrack, tickets could be used to order them. Each taker would get a ticket indicating where to pour, and then wait until pourers with lower tickets were done. Sequencers and tickets were motivated by the ticketing operations used in bakeries or other busy stores.

In general, then, a synchronization approach to a problem, when feasible, is more elegant and efficient than a mutual exclusion approach which relies on protection of a resource rather than coordination of the processes. Event-counts and sequencers are superior to using an ad hoc shared variable approach since they provide the necessary primitives such as Await. However, eventcounts and sequencers still suffer from the necessity to include synchronization control in the code of the processes.

In the simple example given, the path solution was the most elegant because it removed the synchronization from the processes and placed it textually in one place. However, it is sometimes necessary to consider the efficiency of implementation of the high level solution. If path expressions required an underlying mutual exclusion, the eventcount solution might be desirable. Although the primitives are required at a higher level, the eventcount solution to the producer-consumer example guarantees that mutual exclusion will not be necessary.

D. COMMUNICATION BY MESSAGES

Many solutions to concurrent problems require communication between processes. For example, a communication path exists when two processes may read or write to a shared variable. This section considers communications done in a more structured, higher level manner--by messages.

Messages in the most restricted form have no content. As representatives of this type of message, eventcounts and semaphores will be compared. Unrestricted messages will then be considered, and finally, a suggested message system will be proposed.

1. Restricted Messages

a. Eventcounts, Await and Advance

Eventcounts, along with the advance and await operations, can be used to communicate information between processes (eventcounts were discussed in the last section). What information is exchanged? The advance operation merely increases the value of an eventcount. Await prevents proceeding until an eventcount has increased to (or beyond) a specified value. The information received is simply a lower bound on the number of advances which have been done. An advance can be thought of as a signal which contains no information other than the fact that it has been sent: <u>the existence of such a signal is the message</u>. (Sometimes messages without information content are called "timing signals"). The await primitive combines detection of message existence with a blocking action. By specifying a relationship between the number of advances done and some internal variables (such as the number of messages previously received), the Await operation receives the message that some relation exists.

b. Semaphores, P & V

P and V operations associated with semaphores were introduced by Dijkstra as a means of solving the mutual exclusion problem (discussed in the first section of this chapter). P and V can be used as a receiver and sender operation for a message. P (the receiver operation), like the await operation, combines detection of a message with a blocking action. Likewise with P and V, the information exchanged is only the existence of a message.

A semaphore can be thought of as a shared integer which is initialized to some value. The effect of the signal operation, V, is to increase the semaphore by one. A P operation causes waiting until the semaphore is greater than zero, then decrements the semaphore.

c. Semaphores Versus Eventcounts

Further clarification of how semaphores and eventcounts work can be achieved by comparing how the two are used to solve a simple problem. Two processes, a sender and a receiver, each execute some code repeatedly. At one point in the sender's code, a message is sent to the receiver; the message contains no information other than its existence--it is just a "signal". Similarly, at some point in its code, the receiver checks to see if a message has been sent; if no message has been sent, it waits until one is received. Each time it passes through its code the receiver checks for a new message; i.e., it doesn't consider previously detected messages as allowing it to proceed. Note that this problem is a synchronization type of problem; the sender must pass through a certain point in its code before the receiver can pass through a corresponding point in its code. The sender can get arbitrarily far "ahead" of the receiver, and must always stay "ahead".

Figure 6 shows solutions for this problem using first semaphores and then eventcounts.

Although the two solutions are similar, there are some subtle, but important differences. In the eventcount solution, only the sender changes the value of the shared eventcount; the receiver only reads it. But in the semaphore solution, both the sender and receiver may change

```
SENDER                                  RECEIVER

Solution with SEM <-- 0;     /initial value    DO FOREVER
 Semaphore,                  of semaphore/      /code/
 P and V          DO FOREVER                    P(sem);          /block self till mes-
                   /code/                                        sage received/
                   V(sem);   /send message/     /code/
                   /code/                      END DO
                  END DO

                  SENDER                       RECEIVER

Solutionwith SEQ<-- 0;      /initial value    i <-- 0;          /i represents number
 Eventcount                  of Sequencer/                       of messages received/
 and Await        DO FOREVER                   DO FOREVER
                   /code/                       /code/
                   Advance(SEQ)/send message/   Await(seq, i+1)/Block self till
                   /code/                                       message received/
                  END DO                        i <-- i+1       /one more message
                                                                 received/
                                                 /code/
                                                END DO
```

Figure 6. Sender/Receiver Problem Solved By Eventcounts and By Semaphores

the value of the shared semaphore. Thus there is write competition between P and V, a situation which complicates the concurrency considerations.

In addition to the write competition problem in the semaphore solution, there is a security problem. By changing the value of the semaphore, the receiver is "broadcasting" its actions. Contrast this with the eventcount receiver. It keeps track locally of the number of signals received and waits till the number of signals sent has the desired relation with the number received; in this manner, the receiver does not broadcast its actions in any way. The await operation is a pure "observer" of events, whereas the P operation is not. In contrast, both the advance operation and V operation are pure "signalers" of events. The need for a both pure observers and pure signalers is a consideration in designing secure operating systems [Reed79].

2. Unrestricted Messages

Communication restricted by the use of signals whose content is merely their existence is useful, but sometimes it is desirable to transmit more information. In a track race, it is sufficient to signal the runners to start with a gunshot--they know what to do. On the other hand, it would be a mistake to send someone to the store without telling them what to purchase.

a. Categorization of Messages

There are many types of messages. It is useful to categorize some logical divisions.

(1) Broadcast Versus Consumable Messages. Messages can be divided into two broad categories: broadcast and consumable. A broadcast message is available for everyone with access to it to see. An analog is a message bulletin board. In the eight puzzle problem, a broadcast message could be used to promulgate a new best solution. A consumable message, on the other hand, can be received by only one process--the message is "consumed" in being received. If a process in the eight puzzle problem wanted another process to expand a node, it could use a consumable message containing the value of the node.

(2) Broadcast Messages Locally Consumable. A broadcast message is sent to a set of processes, and is available for all of them to receive. Once a given process has received the broadcast message, should it be able to receive it again, or should it perceive that there are no more messages for it? The answer depends on the purpose of the message. Accordingly, broadcast messages can be divided into those that are "locally consumable" and those that are not. Locally consumable broadcast messages seem the most useful, but it may be easier to implement broadcast messages to not be consumable.

Note that a locally consumable broadcast message addressed to only one receiver is equivalent in effect to a consumable message addressed to one receiver.

(3) Broadcast Messages Queueing vs. Superceding. Another consideration of broadcast messages is whether newer ones should supercede older ones. Again, both categories are useful. If older messages are made obsolete by newer ones, then a process should only receive the latest message; i.e., messages should not queue. On the other hand, it may sometimes be desirable to queue broadcast messages.

(4) Specifying Receivers of a Message. When a message is sent, it needs to get to where it is going. How does one specify the receivers of a message? The intended receiver may be a single process or a group of processes. There are two ways to specify the receiver of a message. One way is to name the receiver. A letter addressed to a person does this. Another way is to use a common area, or "bin", for holding the messages--by restricting access to this bin of messages, the flow of messages is controlled. These two approaches can be combined--one could specify the name of a process as well as the name of a bin.

(5) Blocking. The method of receiving a message is another issue. Should a process trying to receive a message be blocked until a message is available, or should there be a means of checking for the existence of a message

as a prelude to receiving it? Most suggested primitives for message reception involve blocking a process untiil a message exists. An exception is the Read operation associated with eventcounts. Read can be used in lieu of the Awaiit operation when it is desired to obtain the value of an eventcount without being blocked. A more general primitive for checking the existence of a message is proposed in this paper.

Similar to the notion of blocking a process receiving a message is the notion of blocking a sending process. A program designed with a fixed size queue for accumulating messages sent but not received would need to block a process sending a message to a full queue. This blocking necessity may be inherent in some message types, but it may be desirable to keep the queue structure and potential blocking at a lower hierarchial programming level so that the higher level processes need not be aware of them.

b. A Suggested Message System

This section suggests the syntax and rules for a message system based on some of the features previously discussed.

(1) Message Types. Three message types are available: consumable, broadcast-queue, and broadcast-supercede. A consumable message can be received by only one

process; once received, the message is no longer available for other processes (including the one that received it). The two types of broadcast messages, broadcast-queue and broadcast-supercede, are "locally consumable". All members of the audience to which a message is broadcast can receive the message, but each process can receive the message only once. In broadcast-queue messages, new messages do not supercede old ones. In broadcast-supercede messages, new messages do supercede old ones.

(2) Operators. There are two receive operators: await-receive (await for short) and exist-receive (exist, or E, for short). Await-Receive (<message specification>) where <message-specificatioon> refers to the syntax required to describe the message (contents and address), blocks the process using it until a message of <message-specification> is received.

The blocking characteristic exhibited by await-receive is fundamental to most message receipt primitives which have been previously suggested in the literature (as mentioned, the Read operation associated with eventcounts is an exception). For example, Dijkstra's P operator [Dijk68A] (used with semaphores) and Reed and Kanodia's await operator [Reed79] (used with eventcounts) each block the calling process until a condition is satisfied. This blocking characteristic is useful if the programmer doesn't want the program to proceed unless some condition or event occurs. However, often the converse situation is desirable: when the program should continue unless some condition or event occurs. For this type of situation, the exist-receive operator is introduced. Exist-Receive (<message-specification>) returns a boolean value--if true, message has been received; if false, no message has been received. Using this operator, a program can check for external messages, and take action when they exist.

In addition to the reception operators, there is one sending operator. It is send (<message-specification>).

(3) Specification of Receiver. The receiver is specified by the "bin" method previously discussed, and can additionally be specified by name. The notion of a message bin is that messages can be directed by specifying a common bin-area. The bin-area is specified implicitly by the declaration of an instance of a message previously declared. (This is clarified in the following section on syntax.)

(4) Syntax. The message system will be clarified by discussing its syntax.

(a) Operators. The syntax of the three operators is listed below:

```
EXIST-RECEIVE (<MSG-NAME>, <SENDER-NAME>*)
AWAIT-RECEIVE (<MSG-NAME>, <SENDER-NAME>*)
SEND (<MSG-NAME>, <RECEIVER-NAME>*)
```

NOTES

1. The "*" indicates zero or more occurrences of this item may be used.
2. <Sender-name> and <receiver-name> could include such operating system primitives as children and parent (of dynamically created processes).

(b) Declarations. Declarations are required both within the program and within each process. (By "program" is meant a single program in which any number of concurrent processes may be declared and initiated). Declarations required within a program are Pascal-like and are listed below:

```
TYPE
    <message-type-name> = message  <message type>
                                   <message structure>
                                   {queue-length: <positive-
                                     integer>}
```

NOTES

1. <MESSAGE-TYPE-NAME> is the declared name of a certain message (type and structure); it is usually declared for a specific purpose (such as broadcasting new best solution values in the eight-puzzle program).

2. <MESSAGE-TYPE> is one of: consumable, broadcast-queue, or broadcast-supercede.
3. <MESSAGE-STRUCTURE> is the format of the message, e.g., integer or record. If the message has no content, this is indicated by using the words "NO-CONTENT" for <message-structure>.
4. Optionally, the programmer may specify the maximum number of messages which may be queued before a process is blocked by trying to send a message. The number of messages is specified in the <positive-integer> portion of {queue-length: <positive-integer>}.

Declarations required within a process are listed below.

```
VAR
    <BIN-NAME>:  <MESSAGE-TYPE-NAME> MESSAGE
```

NOTES

1. <MESSAGE-TYPE-NAME> is a message name declared in the program body.
2. <BIN-NAME> is an instance of the specified <message-type-name>. The <BIN-NAME> is a variable allocated in the processes' space and is of the format specified in the program as <message-type-name>. It serves as the receiving and sending area for messages. When a message is received, its value has been placed in the variable <BIN-NAME>. To send a

message, <BIN-NAME> is first assigned a value (if it has any content) and then the value sent. The <BIN-NAME> variable can be treated as any other variable, except if it has a no-content structure. It is an error to try to read or write to a no-content variable.

(5) Example: Multiple Producers, Multiple Consumers with Buffer. To clarify this message system, the multiple producer, multiple consumer problem will be solved. Let the buffer be an array of size N numbered from 1 to N. Producers desire to produce and place their product in the i th slot of the array by executing produce(i). Consumers similarly consume. The problem, of course, is to somehow coordinate the producers and consumers so they don't interfere with one another, and so there is no race condition. The solution is shown below.

```
PROGRAM

        TYPE
            SLOT-TO-USE = MESSAGE  CONSUMABLE
                          INTEGER: 1 .. N
                          QUEUE-LENGTH:N
        /PROCESS PRODUCE/
            PRODUCE = PROCESS
            VAR
                FULL, EMPTY:  SLOT-TO-USE    /MESSAGE/
            BEGIN
                DO FOREVER
                    AWAIT-RECEIVE (EMPTY)
                    FULL <-- EMPTY
                    PRODUCE (FULL)
                    SEND (FULL)
            END DO
        END /PRODUCE PROCESS/
```

```
/PROCESS CONSUME/
    TYPE
        CONSUME = PROCESS
        VAR
             FULL, EMPTY:  SLOT-TO-USE
        BEGIN
             DO FOREVER
                 AWAIT-RECEIVE (FULL)
                 EMPTY <-- FULL
                 CONSUME (EMPTY)
                 SEND (EMPTY)
             END DO
        END /CONSUME PROCESS/

BEGIN /PROGRAM/
    CREATE - PROCESS (PRODUCE)
    CREATE - PROCESS (CONSUMER)
END /PROGRAM/
```

As written, this program won't work. Producers and consumers send messages to each other which consist of the slot number they just filled or emptied. Since these messages are consumable, producers and consumers will always alternate with respect to any given slot. The problem is that someone has to get things started. Say the buffer is initially empty. Then the producers need N messages telling them to produce into slots 1 through N. Adding the following "dummy consumer" procedure to the program code gets the processes started and correctly completes this program.

```
PROCEDURE DUMMY-CONSUMER

  BEGIN
    FOR i <-- 1 to N DO
        EMPTY <-- i
        SEND (EMPTY)
    END DO
  END /DUMMY - CONSUMER/
```

If the buffer were initially full, replacing the word EMPTY by FULL in the procedure above would correctly start the program.

It should be mentioned that the above solution does not force the produced products to be consumed in the order that they were produced. This illustrates that messages do not naturally order events. This characteristic of messages is sometimes helpful and sometimes harmful.

3. Advantages of Using Message Passing

In the multiple consumer-producer example no mutual exclusion enforcement was necessary at the highest level (that of the program). Mutual exclusion requirements, if any, would be implemented at lower levels, probably within the operating system. This hierarchical "push-down" of mutual exclusion requirements is in keeping with the philosophy discussed in the section on mutual exclusion. There are other potential advantages for using message passing, namely more flexibility for compatibility with the underlying architecture, and more hierarchial structuring. These are illustrated in the following example.

In the eight-puzzle problem, there may be a need to occasionally promulgate to processes the value of a new best solution. An approach which doesn't use message passing is to use a shared variable which contains the value of the best solution found. Periodically, a process could read the variable's value and check if it agrees with what it thinks is the best solution. This approach could be hierarchically structured by using a read procedure (within a monitor-type section of code) to access the shared variable. The read procedure would ensure, at a lower level, that concurrency requirements were met.

A message-passing approach to this problem would consider the intended use of the information being sent. At the highest level (processes doing expansion), the message would be declared as a broadcast type in which newer messages supercede old ones.

The lower level procedures for message sending and receiving could interface with the operating system to take advantage of this type of message. With a broadcast type message, it is not necessary to have a shared variable. An update of the best solution (sending a message) could send new copies to remote sites. If a process gets an old message value because an update is still in progress, no undesired results occur. The ability to implement the message passing at a lower level as sending copies is important for two reasons. First, it allows elimination of the bottleneck of needing one shared variable. Second, it allows the flexibility to adapt to a different architecture such as a distributed system. To implement the shared variable approach at a lower level in a distributed system is difficult because there is no shared memory. Furthermore, the lower level procedures in a shared variable approach have no knowledge of the nature of the use of the variable. By structuring the problem with a message-sending solution, the use of the message information is categorized (by declaring a message type) so that lower level procedures can take advantage of it.

The message passing approach also allows greater hierarchical structuring. Once lower level procedures have been written which implement message passing, the programmer need not worry about designing data structures (except for the message structures) as is necessary for shared variables. For example, a shared data structure approach might require designing a queue whereas a message approach already has an implicit queue.

E. CONCURRENT VERSUS DISJOINT PROCESSES: NONDETERMINISM

Race has already been discussed as one potential difference between disjoint processes and interacting concurrent processes. It was shown in the V <-- V + 1 example how the effect on a shared variable can vary from run to run. Assume that all race problems have been corrected. Does that mean that for a given input, the output of a group of concurrent processes will always be the same? This section explores that question.

In a disjoint sequential program with a given input, the set of statements executed, the time ordering of those statements, the set of outputs and the time ordering of output will always be the same. That is, the program is deterministic. How does this differ from a program consisting of concurrent processes? The differences vary depending on the type of concurrent program. These differences will be considered in order of increasing disparity.

What is meant by input and output of a process? Define input simply as anything the process has access to which can be varied outside the process from run to run. Define output simply as what the program affects outside its boundaries. Note that the specification of the boundaries of a process are somewhat arbitrary. Now, consider a single process which is part of a group of concurrent processes. For a given input, its behavior will be deterministic--the statements executed and their time ordering as well as the output and its time ordering will always be the same (again, this is assuming no race problems exist).

1. Precedes Relation Example

Now consider a program consisting of several concurrent processes. Let the program be a precedes-relation type program as defined earlier, with output only at the end of the program. For a given input, the time ordering of statements varies nondeterministically between "check-points". The statements are interleaved in an arbitrary, unknown fashion. However, the output is always the same. Moreover, communications or "meeting points" of the processes always occur deterministically with respect to the set statements already executed.

2. Mutual Exclusion Example

For the next example, two processes asynchronously access a shared resource. For whatever bizarre reason, one process repeatedly prints out a line of question marks and the other prints out a line of ones. The printer is the shared resource; each process gains control of the printer by a request-printer call (which effectively starts a critical region), prints out a line, and releases the printer by a release-printer call (which ends the critical region). Again, the time ordering of statements of the program (which consists of both processes) varies. But, in this case, the time ordering of the output also varies nondeterministically. If one watches the printer, the lines of ones and question marks will be interleaved in arbitrary fashion. Note that this potential variance of output from program run to run does not imply that the program is "wrong". It depends on the intent/purpose of the program. However, the variance of output reinforces an important point. Trying to troubleshoot or test a concurrent program can be almost impossible because of the difficulty of trying to infer the program's behavior from a nondeterministic output.

3. Optimal Eight-Puzzle Example

In the previous example, the content of the output was deterministic, although the time ordering was non-deterministic. Consider a concurrent program which solves the eight-puzzle problem. It consists of some fixed number of processes, each of which is working on expanding a subtree of the problem. When a process finds a solution, it updates a shared data structure containing the value of the best solution found so far. The heuristic function which evaluates the potential of a node to be on a solution path ensures that the optimal solution is found (assume for implicitly that there is only one optimal solution). Periodically, processes check the value of the best solution found and throw away any leaf nodeswhich don't have as much potential.

Certainly, the time ordering of statements of the whole program is non-deterministic. Furthermore, what a process does from run to run may vary. Because a process periodically discards nodes, it may expand nodes in one program run which in another program run it discards because another process, running "faster", found a better solution sooner. However, because the solution method is optimal, the output of the program will always be the same.

4. Non-Optimal Eight-Puzzle Example

Consider the preceding program "complicated" by making the heuristic function such that an optimal solution is not guaranteed. (i.e., nodes which are on an optimal solution path may be thrown away based on a non-optimal solution already found). The program is further complicated by allowing a process to spawn other processes if it thinks it has a group of nodes with outstanding potential. This program is drastically different from a single sequential program. Of course, the time-ordering of statements within the program is nondeterministic. So is the time ordering of statements within a process. Moreover, the group of processes which execute is nondeterministic because dynamic process creation is allowed. But, the most dramatic difference of this program is that the program output is non-deterministic. Because the program is non-optimal, a process which in one program run found an optimal solution may in another program run throw the solution away because another process found a solution sooner that appeared better.

Here is a program which is correct (it does what's intended) which may give different, but correct answers from run to run depending on factors such as the number of processors, the loads on the processors, etc. This situation is contrary to the fundamental notions of computer science. Programs are supposed to give the same output for a given input. If they don't, something's wrong (such as a race condition). There are many reasons against writing programs such as this. What such programs do is difficult to conceptualize, hence the programs would be difficult to write, troubleshoot, and test. To depend on such a program would be dangerous--it might be impossible to know if the program were correct. (One might never want to put such a nondeterministic program onboard a space vehicle, for example).

But, there are also arguments for such programs. They might prove to be fundamentally more powerful in some respects. As an analogy, the tools of recursion and pointers are also difficult to understand, but they are very powerful (some would argue that recursion and pointers should not be used). The notion of a program with a nondeterministic output may closely mirror life. For example: a group of people are trying to decide an approach to take for a problem. During the discussion, Mary makes a suggestion which everyone thinks is good. Other paths which were being explored are thrown away. Eventually a solution is agreed upon. This solution reflects Mary's input. Now, assume the problem can be re-run; identical people are put together to discuss the problem, only this time, Mary has a headache. The discussion proceeds the same as before until the point at which Mary previously made her suggestion. She makes no suggestion this time because of her headache. The final result is a different solution, although it is a valid one.

As another example, a robot is built which consists of many different processes, each running on separate processors. The robot has a process which estimates the size of openings to determine if it can go through them.. If the sizing processor were slowed down by a weak battery, the robot might react differently to openings. Thus, the "output" of the robot for a given "input" may vary from run to run. An argument for using separate processors for a robot is reliability--if one processor fails, the robot can still partially function. This example of a robot is like a human being who may react differently to situations because of physical injuries or disease.

There are many problems which are so hard that optimal solutions probably do not exist. For those problems, the notion of doing the best one can, with resources available at the time, (such as number of processors) may be a valid approach. A non-deterministic program consisting of interacting processes may be a powerful, reasonable way to approach such problems.

5. Nondeterminism Recap

To recap, a single, disjoint process for a given input is deterministic in the set of statements it executes, the time ordering of those statements, the set of output and the time ordering of the output. A program of concurrent processes may be nondeterministic in some or all those categories and still be a "correct" program (i.e., the program does what is intended).

F. CHAPTER SUMMARY

An underlying theme of Chapter 3 has been the fundamental importance of structuring concurrent problems. It is important to distinguish between approaches to structuring and tools for implementing these approaches. To an extent, the tools available shape the approaches, but separation of the high level approach from the tools makes for a clearer solution to a problem. The tools and approaches discussed in Chapter 3 are synopsized in the following paragraphs.

Although it is important for understanding concurrency, mutual exclusion should not be used as a high-level approach to a problem. If it is needed, mutual exclusion should be relegated to a lower heirarchical level, preferably by incorporation into abstract data types.

The precedes relation provides a good example of a high-level structuring approach and is applicable to certain problems. It illustrates the advantage of externally coordinating a group of processes: by placing the code for concurrent control of the processes textually in one place, the program is easier to understand and maintan.

Often, the need for mutual exclusion can be avoided by ordering processes with respect to shared resources. A synchronization approach does this. It can be implemented using tools such as eventcounts and sequencers.

A high-level tool which can be used to structure some concurrent problems is message passng. The use of message passing can often eliminate the need for mutual exclusion by providing communications without shared variables.

Concurrent approaches to problems should provide a large increase in the power of the computer. Unorthodox practices such as programs with non-deterministic outputs may help realize that increase in power. Another consideration is the hardware upon which the software runs. Since the power of concurrency is realized by multiple processors, questions such as how well an architecture allows processes to communicate must be considered. Hardware considerations are addressed in the next chapter.

IV. DEVELOPING A CONCURRENT TREE SEARCH PROGRAM

With the concurrent programming issues of the last chapter in mind, this chapter discusses some of the considerations and approaches for writing a concurrent tree search program such as the eight puzzle. First, it is argued that it is necessary to consider the underlying architecture in writing the program. Next addressed is the fundamental question of whether the tree should be a shared data structure. Then, memory management problems are considered. The final three sections discuss approaches and problems in starting, doing, and finishing the tree search.

A. THE NEED TO CONSIDER THE UNDERLYING ARCHITECTURE

In considering the usefulness of concurrent programming techniques, it is helpful to review the context in which the techniques were developed. The initial problem was that of an operating system in which different programs had to use a single processor. Moreover, some of these programs competed for shared resources, such as a printer. A conceptual breakthrough in multiprocessing was the development of the notions of process and virtual processor. Each process, conceptually, ran on its own processor. This presented the problem of asynchronous processes accessing shared resources.

The initial tools of concurrent programming were developed to solve this problem.

These initial concurrent programming tools were based on systems in which there were one, maybe two, processors and there was one shared memory (even considering the notion of virtual memory, the correct picture was that of a shared memory). Thus, mutual exclusion ensured exclusive access to a <u>shared</u> data structure; a monitor consisted of a group of procedures located in a <u>shared</u> location, and which accessed <u>shared</u> data structures; eventcounts and sequencers synchronized processes with respect to <u>shared</u> data structures and eventcounts were <u>shared</u> variables accessible with certain operators. These approaches work well in a system in which only one or two processors access the shared memory, and in which processes are loosely coupled.

Consider now high level programs which are written to be concurrent, and which include, for instance, five or more cooperating processes. The value of such a program is the potential speed increase to be gained by doing a problem in parallel. This value is realized only by running the processes on separate, physical processors. But, if the program is written using tools which assume "easy" access to a shared memory, problems arise. For most architectures, access to shared memory is done by a common bus. If the processes are frequently accessing shared memory, a bottleneck occurs at the bus. Furthermore, this memory bus bottleneck may be coupled with a mutual exclusion bottleneck of shared data structures, making the problem worse. The effect of such bottlenecks is to slow down the program, i.e., to degrade the potential speed increase of concurrency.

Such problems imply that concurrency techniques based solely on easily accessible shared memory are insufficient for writing high-level concurrent programs. The problems also imply that somehow the concurrent tools should consider the underlying architecture. What is needed is a high-level view which is conducive to writing concurrent programs which run efficiently on the underlying architecture.

If concurrent programming tools are going to support an architecture, what architecture should they support? Perhaps the ideal approach is to develop good concurrency writing tools and then to develop hardware which best supports these tools. Some suggest that a major problem with most programming languages is that they were based on an underlying Von Neumann computer architecture [for example Back78].

One proposed language design for concurrency is based on the notion of data driven statement execution. A program would consist of a set of unordered statements, each of which is executed as soon as its operands are ready. Such a language, called a data flow language, has a high potential for concurrency. However, the architecture needed to support data flow languages is still being designed.

While concurrency tools which require new architectures are being developed, some tools are currently needed which make use of existing hardware. In order to discuss high-level concurrency approaches which facilitate efficient use of underlying architecture, a realistic hardware configuration will be assumed. This architecture consists of a number of processors which each have their own local memory, and which are all connected to a shared memory by a common bus. Figure 7 shows such a configuration. It is a typical architecture for multiprocessing systems and, thus, will be an appropriate vehicle for discussing concurrent approaches to the eight-puzzle problem. It should be noted that the conclusions developed are not necessarily limited to this architecture. In general, the discussion is relevant to any architecture in which processor communication is limited.

With the architecture of Figure 7 in mind, some issues in the design of a concurrent eight-puzzle program will be discussed. These issues involve fundamental considerations of concurrent programming, especially tree search, on a conventional architecture. The discussion assumes the availability of dynamic process creation, i.e., processes can create other processes at run time.

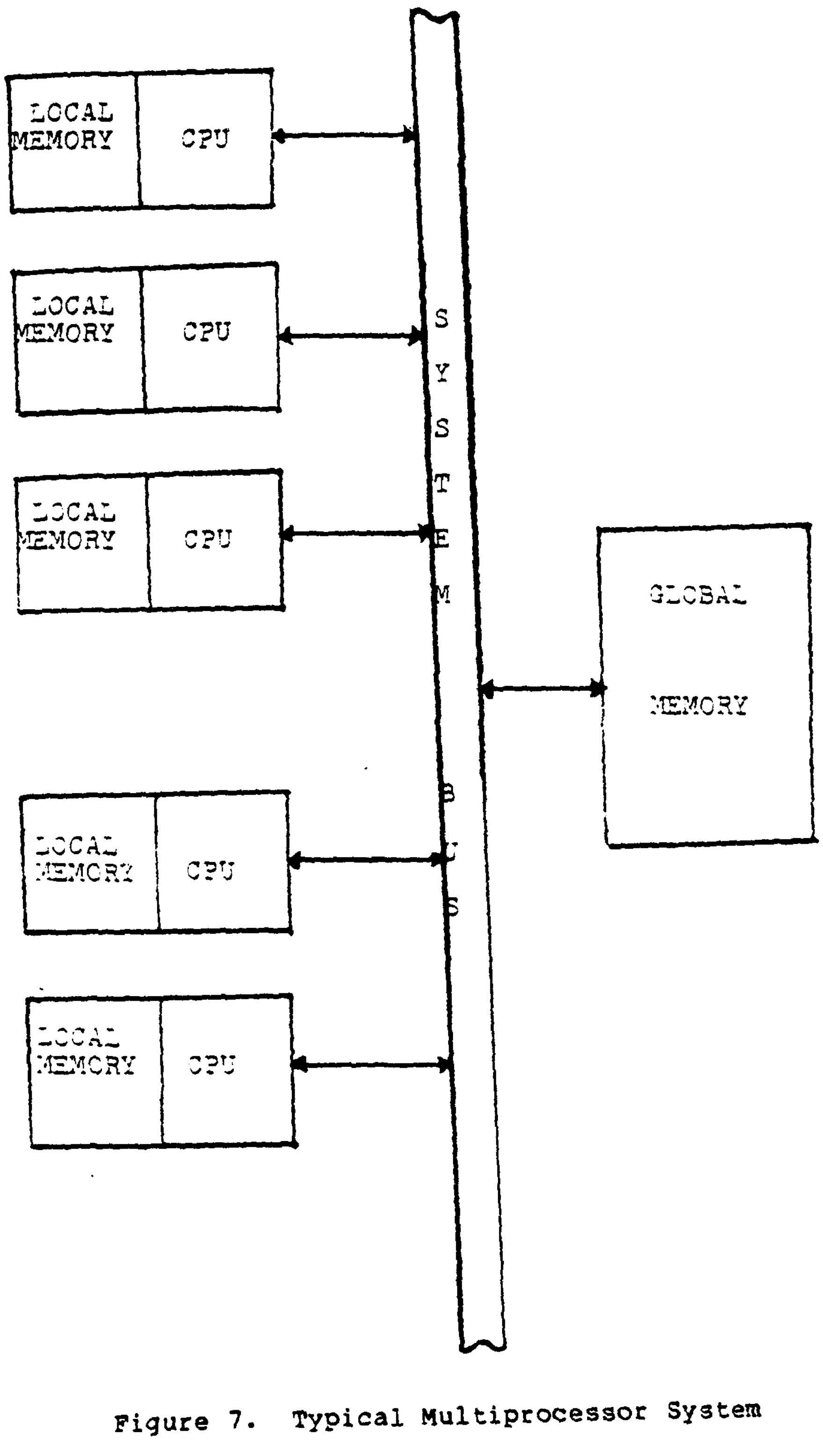


Figure 7. Typical Multiprocessor System

B. DIVISION OF TREE SEARCH AMONG PROCESSES

The first question to be addressed in a concurrent tree search is whether or not the tree should be a shared data structure.

1. Shared Frontier Approach

Perhaps the simplest approach is to place the tree in shared memory. Each processor would have a search process which retrieved a node from the frontier of the tree, expanded it, checked for a goal, and placed the children in the frontier. The problem with this approach is one of bottleneck. Processes compete for the common bus to access shared memory. With more elaborate hardware, simultaneous bus and memory accesses might be possible. This could reduce the hardware bottleneck, but there is still a mutual exclusion bottleneck with respect to the data structure. The data structure bottleneck could be reduced by allowing simultaneous operations on the frontier as discussed in the last chapter. At best, there is still a significant bottleneck for a shared tree structure. Avoidance of bottlenecks should be a fundamental consideration in concurrent programs.

2. Division of Tree Approach

To avoid a bottleneck, then, it makes sense to divide the tree among processes, letting each process search a subtree. An extreme of this approach is to let each process represent a single node. A process would expand a node and create a child process for each of the children nodes. The program would be started by expanding the root node far enough to place a process (representing a frontier node) on each processor. On each processor, as processes expand, a subtree is being explored. To control the subtree exploration, each frontier node process would have a priority based on the heuristic value of its node. In effect, the operating system of each processor would control the heuristic search of a subtree by controlling the order in which the frontier-node processes ran. One problem with this approach is the amount of overhead required for the numerous process creations. If process management were implemented at a hardware rather than software level, this approach might be efficient.

An approach between the two extremes of a shared tree and a single-node process is to have each process explore a subtree. As before, the root node is expanded until the frontier is large enough to divide among the processors. On each processor, then, a process explores a separate subtree.

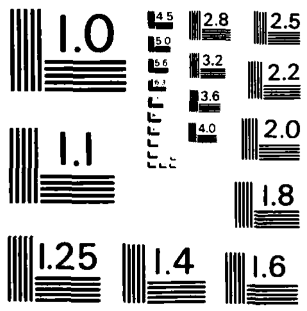
How good is this approach heuristically? If a process is passed a node to expand which has very little heuristic value, that process and its processor will be of little help to the overall problem. To ensure that each process gets nodes with enough potential, perhaps each

AD-A125 647 DEVELOPMENT OF A CONCURRENT TREE SEARCH PROGRAM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA C N POWLEY OCT 82

2/2

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
4 83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

process should be initially passed a number of nodes, rather than just one. Furthermore, it might be desirable to periodically recombine each process' frontier. This recombination, or collation, would be time-consuming, but might be worth the improvement realized by ensuring a uniform heuristic distribution of nodes. Questions such as how often to collate and how large a frontier to initially pass a process are difficult. They warrant extensive mathematical analysis coupled with empirical testing.

C. MEMORY MANAGEMENT PROBLEMS

Typically, in a tree search problem, a node is a record which includes a pointer (or some other reference) to its parent. This is necessary so that when a goal node is found, the path from it back up to the root node can be followed to construct the solution. (An alternative to using pointer in nodes is for each node to remember its ancestry--this approach won't be considered).

Consider a problem approach which allows dynamic process creation--a process may expand a frontier, then break it up and pass it to another process. But in doing this, a process can place in its memory space a node record which points to another process' memory space. Should that situation be allowed? If it is, the implication is that a process can retrieve a record from another process' memory space by following a pointer. This would require the local operating system associated with a processor to access another processor's memory space, a potential bottleneck. Furthermore, mutual exclusion is required so that a process cannot change a node while it is being read by the operating system.

A method for allowing inter-process memory pointers is discussed in the next chapter. It is based on message sending between processes to pass the value of a node. The programmer is presented with the high-level <u>view</u> that each process has its own memory space, or <u>bucket</u>, which is accessible only to it. Although perhaps not the most elegant solution, it requires little operating system support and presents a <u>view</u> to the programmer which correctly suggests that the operation is time-consuming.

Another consideration in moving records with pointers is dangling references. To avoid jumping among process/processor memory spaces to print out a solution (path from root to goal), it might be desirable to move all subtrees to the global memory space. This involves moving interior nodes. Interior nodes are pointed to by their children. Thus, when an interior node is moved, the pointer of the child node is left dangling. Either the child pointer must be changed or a reference to the old address must be converted somehow to the new address. One possibility is to record changed addresses in a hash table; everytime a pointer reference is followed, the hash table would be checked first to see if there were a substitute address. Clearly, allowing movement of interior nodes requires a significant amount of overhead. The benefits of collation may be great enough to warrant either movement of interior nodes, or changing the problem approach so pointers aren't required.

These problems show that memory management is a fundamental concern of concurrent programming. Often the tradeoff is between passing an enormous amount to a process so that no external references are needed, or requiring complex memory management capabilities of the operating system.

D. STARTING THE SEARCH

The desired configuration of the program when the search is in progress is that there will be one or more processes per processor, each process exploring some part of the tree. How is this configuration achieved? One approach is to have a controlling process, the director, which creates processes and passes them a subset of the frontier to expand. The director would start by expanding the root node for some time and then divide the frontier and pass sections of it to created processes.

Another approach is to let each process have the ability to create other processes. An initial process would expand in an amoeba-like splitting fashion until all the processors were full. This would create a tree structure of processes with parent-child relations among the processes.

So, two "start-up" approaches have been suggested. One involves a director overseeing a single level structure of processes, whereas the other approach involves a tree structure of processes. The tree structure might also have a director process which created the first search process. An advantage of the single level process structure is that it is simple. A disadvantage of the single level is that there is a potential bottleneck at the director process. For example, the single level director must sequentially create search processes, whereas processes can be created concurrently in an amoeba splitting manner in the other approach. A more difficult consideration is which of the two methods of start-up is better for ensuring an initial uniform heuristic distribution of the frontier. This consideration will not be analyzed in this thesis.

For either method of program start-up, there needs to be a way to stop process creation when all processors have been used. One approach is to know before program execution time the number of processors, and to program this as a constant into the program. Thus, a director process would know there were, say, seven processors, and each processor would have a process initially loaded on it. Sugi81, for example, solves the problem this way. This approach is considered too inflexible and too closely coupled to the hardware configuration.

What is needed is an interface with the operating system which either indicates the total number of processors available, or indicates if any processor is available. To prevent a race condition, the indication of an available processor must be coupled with allocation of the processor. For this reason, the approach suggested is a mechanism for indicating if a single processor is available. An elegant vehicle for implementing an operating system interface of processor availability is message-passing. A process could be allocated a processor by receiving a consumable message. Making the message consumable assures that the processor is allocated just once. Depending on the process' need for the processor, the message receipt operator used could be either a blocking or an existence type receipt operator.

For this suggested processor availability mechanism, the operating system must (conceptually at least) send a message for each available processor. When a message is received, the operating system would expect the receiving process to create a child process. The operating system would then allocate the processor to the created process. Furthermore, prior to allocating a processor for a reason other than message receipt, the operating system would have to "retract" one of its availability messages (conceptually, the operating system would receive a message itself). One further advantage of processor allocation by messages is that it allows dynamic changes in the number of processors.

E. WHILE THE SEARCH IS IN PROGRESS

Once the program has been started and processes are distributed among processors, the most important and time-consuming phase of the program takes place; the search for an optimal solution. The key considerations of this concurrent tree search will now be discussed.

1. Promulgating the Value of Best Solutions

In a heuristic search, after a solution is found, the search will continue as long as there are frontier nodes with more heuristic potential than the value of the solution already found. The search stops when there are no nodes left with more potential than the best solution found. It is apparent that expansion of nodes which have a lower potential than a solution already found is non-productive. To preclude such wasteful expansion in a concurrent tree search, it is necessary to somehow promulgate the value of each new best solution to other processes.

One way of keeping track of the overall best solution found is to use a shared variable. Periodically, processes check the value of the variable to see if it has changed. When a process finds a new solution, it would update the variable to reflect this (after, of course, comparing the variable's value to the value of the solution the process had just found). The disadvantage of the shared variable approach is that it presents a potential bottleneck.

Another approach for keeping track of the best solution found is to use message passing. The nature of a superceding broadcast message is well suited for this application. When a process finds a solution, it promulgates the value (cost) of the solution by sending a broadcast-supercede message. Similarly, each process periodically checks if a new best solution message exists.

On first glance, the message passing approach just suggested for promulgating solution values appears to work well. However, it has a flaw which should be understood, as it illustrates a crucial type of message-passing error. The reason for using a superceding type broadcast message to promulgate a new best solution is that only the value of the best solution is important. It is possible, however, for a message to be superceded by a message representing a solution of lesser value.

Consider the following example. Process A finds a new solution which has a higher value than the value of the last best solution message process A received. Process A checks for a more recent message, finds none, and sends a broadcast-supercede message to promulgate the value of the solution it just found. About the same time process A finds a solution, process B also found a solution. Process B's solution is also better than the last solution messaged process B received. However, process B's solution is not as good as process A's solution. Processes A and B both check for a new solution message at about the same time, and thus are unaware of each other's solution. Process B sends its best solution message just after process A sends its best solution message.

Because the message is a superceding broadcast type, the operating system promulgates only the most recent one. Thus process B's message is kept and process A's message thrown away. Unfortunately, process A had found the better solution. The effect is that other processes will receive a new best solution message which does not reflect the best solution; they may waste their time expanding nodes with more potential than process B's solution, but which are not as good as process A's solution.

The race situation just described is due to the nature of message passing. There are delays of unknown duration between the time a situation occurs, the time a message is sent to reflect the situation, and the time the message is available for receipt by other processes. Even if these delays did not exist, two messages could be sent simultaneously; if the messages are of the same superceding-broadcast type, the operating system must order them and discard one of the messages. In general, when writing programs using message passing, one should never assume that because a certain message does not exist that such a message has not been sent; it may be that such a message is about to be sent, or has already been sent but is not yet ready for receipt.

How can the problem with using a broadcast-supercede message to promulgate the value of a new solution be solved? The nature of the broadcast-supercede message type is such that it is well suited for promulgation by a single process rather than by a group of processes. With this in mind, one approach is to use one process (call it the best solution process) as a central point of contact for the best solution. Each process, upon finding a new solution, sends a consumable message to the best solution process. The best solution process keeps track of the best solution received and sends superceding-broadcast messages to all other processes. Although this may seem like a time-consuming operation, it is less of a bottleneck than the shared variable approach because of the nature of the messages. A process never has to wait to check the value of a shared variable; it only checks to see if there is a new best solution message. Note that in the message passing solution, delays may still occur in promulgating a new solution due to the time it takes to pass a message, but solution values will never be lost. Also, in both the shared variable and message passing approaches, the processes are delayed in receiving a new solution value by the intervals between checks for a new solution. How often a process should check for a new solution value is a tradeoff between the time required for excessive checks and the potential wasted time spent searching for a solution using nodes with less potential than a solution already found.

Another approach to avoiding the race problem which occurs if different processes use broadcast-supercede messages is to use broadcast-queueing messages instead. This approach requires that a process check for multiple best solution messages. It has the advantage that best solution messages go straight to search processes without passing through an intermediate best-solution process.

2. Ensuring Uniform Heuristic Distribution

Promulgating new best solutions is one way of improving the efficiency of a heuristic concurrent search. A consideration of probably even more importance is ensuring that the heuristic search is uniformly distributed among processes. Conceptually, at any given instant of the search, there is one global frontier associated with expanding (searching) the tree whose root node is the start node of the problem. This frontier is spread out among the search processes so that each such process has its own "sub-frontier" (subset of the global frontier). Ideally, the nodes of the global frontier should be spread out among the search processes so that each process has a set of nodes which are of about the same heuristic worth. If the global frontier is not uniformly distributed, then some processes may be expanding nodes of little heuristic value while other processes have so many "good" nodes that they need help in expanding them. Even if the global frontier is initially distributed in a uniform fashion (accomplishing that is a problem in itself), after a time the distribution may still become lopsided. The problem is how to detect a lopsided heuristic distribution and, once detected, how to remedy it.

a. Using Priorities

One approach to this problem is to use the operating system to aid in the distribution of processes and in the selection of which processes to run (assuming there are more processes than processors). Most operating systems use the notion of _priority_ in selecting processes to run. Processes with a higher priority are allocated more time on a processor than those with a lower priority. The relative time allocations are based on the policies of the operating system. For the assumed architecture, it is reasonable to assume that each processor has an operating system which allocates processes by priority. It is reasonable to further assume that when a process is created, its priority will be considered in determining which processor to place it on. Thus, if all processors have several processes on them and a process with a high priority is created, the new process will be placed on a processor having low priority processes rather than on a processor with high priority processes. Such placement of created processes helps in evenly distributing processes by priority. It should be noted that there may not be much of a common bus bottleneck as a result of creating a process on a different processor than the one on which the process creation request was made. Each processor/operating system should have a copy of the search process so that only the create-process request and the initial process parameters need to be passed along the common bus. In addition to distributing newly created processes, it might be useful if the operating system also redistributed running processes among processors. Whether process distribution is done any time or only during process creation, such actions require communication between local operating systems and introduce complexity and overhead (e.g., bus bottleneck) into the operating system.

If the operating system is to be used as a means of uniformly distributing processes among processors based on the priority of the process, there must be a mechanism for a process to communicate its priority to the operating system.

As suggested in discussing how a process can ascertain the availability of a processor, a good way to interface with the operating system is by message. To communicate its priority to the operating system, a process could send a message containing its priority. The priority of a process would be based on the heuristic worth of its sub-frontier, relative to the sub-frontiers of other processes. Note that a given process may have to change its priority numerous times during its search, and that therefore the operating system must accept dynamic priority changes.

Using an operating system to uniformly distribute search processes does little good if, say, all the good frontier nodes are in one process and the rest of the processes have poor frontier nodes. (When the terms "good" or "bad" are used to refer to a node, they refer to the heuristic value of the node). The operating system technique merely allocates processes uniformly when there are more processes than processors. Call a process with a disproportionate number of good frontier nodes a "hot" process. What is needed is a way to distribute some of the nodes of a hot process. Two ways a process can distribute some of its frontier nodes are by passing nodes to other existing search processes or by creating new search processes and passing nodes to them.

b. Distributing Frontier Nodes by Process Creation

If a hot process tries to distribute some of its frontier nodes by creating other processes, then it issues some type of create-process call to the operating system. Previously it was discussed how the entire program can be started by processes expanding in amoeba-splitting fashion until all processors were full. A process checked to see if a processor were available by checking the existence of an operating system message. Presumably, a create-process call made when all processors have running processes will create a process on a virtual processor. Actually, the process will be time-shared with other processes. The problem with creating a child process to help the parent process is that the child process may be placed on the same processor as the parent process. If that happens, there is no real gain in the concurrent search of the parent process' initial frontier nodes. By using a priority-driven approach, this problem is alleviated since the operating system will place a high-priority newly created process on a different processor than the high-priority parent of that process.

c. Distributing Nodes by Passing

The other approach for a hot process to distribute its frontier is for it to pass some of its nodes to an existing process. To do this requires some means of identifying a process to receive the nodes. This receiving process should be one which has a poor group of frontier nodes. Furthermore, once a receiving process is designated, some method is needed to do the transfer. One way is for each process to periodically check for the existence of transfer messages. The hot process could then send part of its frontier to a designated process and be guaranteed that *eventually* the receiving process would receive the message. However, "eventually" may be too long an interim for these "hot" frontier nodes to be unattended (in an unreceived message). One can think of more elaborate message passing schemes, such as one which "locks" the receiving process in a state of communication prior to sending so that reception occurs soon after sending. If relying on priorities to distribute processes, another potential problem is that a low priority process might never get a chance to receive a message because the operating system might not let it run due to its low priority. This problem indicates the danger of trying to rely on the operating system for some goal while at the same time attempting to achieve the goal at a higher level.

Two approaches for a hot process to distribute some of its frontier nodes have been discussed. One method was by creating new processes and the other method was by passing nodes to an existing process. Both approaches have problems. The danger in creating a process is that the child process may stay on the same processor as the parent. Coupling the create process approach with a priority driven approach avoids this problem. The other method of distributing nodes by passing to an existing process has the disadvantage that it requires some sort of communication hookup. This approach should not be used with priorities.

d. Detecting Hot Processes

A fundamental issue for both of the above approaches is how a process is initially determined to be "hot". That is, how does a process know that its frontier set has significantly better heuristic value than the frontier sets of other processes? For a process which intends to pass some nodes to another existing process, a similar question is how the receiving process is designated.

The first thing necessary to determine if a process is hot is some measure of the worth of a process' frontier set. Such a measure will not be developed.

Given that each process has a procedure for calculating the heuristic worth of its frontier, some means of comparison among processes is needed. A process should not divide its frontier unless it knows that the frontier is better than that of other processes. One means for comparison is to have a global data structure containing the worth of processes. The data structure could contain an average of the heuristic worth of all processes, or a table indicating the value of each process.

A global data structure presents a bottleneck. Another approach is to have a separate process, the heuristic director, in charge of evaluating the relative heuristic worths of the search processes. The heuristic director would keep track of the heuristic worth of each search process. Whenever a search process had a significant change in the heuristic value of its frontier (either an improvement or a degradation), it would send a message of its new value to the heuristic director. When the heuristic director detected a significant heuristic imbalance in the search processes, it would send a message to the hot processes *directing* them to break up their frontiers. If the method of breaking up a frontier by sending nodes to an existing process were used (vice creating a new process), the heuristic director would also determine the receiving process.

If a heuristic director method is used with the approach of breaking up a hot frontier by creating new processes (vice passing to existing processes), an important but subtle problem occurs. Breaking up frontiers by creating new processes implies that there will be more processes than processors, i.e., that each processor will have several processes. Say, for an example, that there are ten processors with twenty processes distributed among them. Let ten of the processes have *good frontiers* of about equal value, whereas the other ten processes all have much poorer frontiers. The processes are distributed so that each processor has one "good" and one "bad" process running on it. If the heuristic director assumes that each process is running on its own processor, then it will conclude that the processors with the bad processes are wasting their time relative to the processors with the good processes; accordingly, it will *direct* the ten good processes to split up. Such an action has the wrong effect. If a priority-driven approach was also being used, then the program was already running optimally; each processor had an equally good process which it was running most of the time because of its high priority. Dividing up each of the good processes does not cause any increase in the amount of processor time spent on the heuristically good processes. If a priority-driven approach was *not* being used, and the heuristic director directed the hot processes to split up, then some improvement does result. Without a priority approach, each processor was initially dividing its time equally between a good and a bad process. After the split, each processor spends 2/3 of its time with good processes and only 1/3 of its time with bad processes. This improvement is more by accident than by design. A more effective approach would have been for the heuristic director to send a message to each of the ten poor processes telling them to go to sleep (wait for the next message). This would have ensured only the ten good processes were left running.

The fundamental problem in the example above is that the heuristic director did not have any notion of how many processors there were. If the example were continued, the heuristic director would keep dividing up the good processes as long as they were better than the poor processes. What is needed is for the heuristic director to have a view of how many processors there are. This implies that, in addition to processor availability messages that might be needed for startup, it might be useful to have an operating system primitive which indicates the number of processors. It might further be useful to be able to determine which processor each process was residing on; this capability could become confusing if the operating system were dynamically shifting processes around based on priority.

e. Uniform Heuristic Distribution Summary

It has been discussed that a fundamental concern of a concurrent tree search is ensuring a uniform distribution of the (conceptual) global frontier among processes, and ultimately among processors. Two basic mechanisms for distributing a process' frontier have been suggested; passing the nodes to an existing process or creating a new process and passing nodes to it.

Both approaches require some means of determining when it is useful for a process to distribute some of its frontier nodes. It may be that it would be effective for a process to distribute whenever the value of its frontier increased by a preset amount. More likely, it would be necessary to compare frontiers of different processes since heuristic worth is of relative importance. Whether relative comparisons are more productive than internal comparisons is a matter for empirical and mathematical measures. Because of their nature, relative comparisons are probably best done by using a heuristic director process as a central point of contact.

The approach of creating a new process for distributing nodes (vice using an existing process) implies that there will be more processes than processors. This further implies that, if used, a heuristic director needs some notion of the number of processors so that new processes won't be created needlessly. Distributing processes by priority can also be coupled with the approach of distributing nodes by process creating.

When the number of processes can exceed the number of processors by an arbitrary and changing number, it gets very confusing trying to ensure that the global frontier is distributed so that it is being worked on uniformly by real processors. The approach of distributing frontier nodes to existing processes has the advantage of being conceptually simple with respect to distributing processes on real processors. Each process can be considered as running n its own processor and this is a sufficient view to achieve uniform global frontier distribution. A heuristic director can identify processes to distribute nodes as well as processes to receive the nodes. The difficulty comes in coordinating an effective transfer. Another advantage of passing nodes to an existing process is that no operating system interface is needed, whereas the process creation approach required as many as two interfaces (one for priority and one for number of processors).

3. Perhaps a Limited Global Frontier

Before proceeding, it is instructive to recall the underlying motivation for breaking the frontier up among different processes rather than having one global frontier. The reasoning was that a global frontier presented too much of a bottleneck. The primary advantage of using a global frontier is that it keeps all the frontier nodes uniformly distributed. Dividing the frontier among processes and then attempting to ensure uniform distribution has been shown to be a difficult undertaking. It may be that dividing the frontier among processes is so expensive that a global frontier is better. Before claiming that either one of these extreme approaches is best, one should consider trying an "in-between" approach.

The frontier of a tree search grows very large. However, the nodes of interest are only those with relatively high heuristic value; these nodes are a small percentage of the entire number of nodes. Much of the bottleneck in using a global frontier is the time it takes to re-insert newly expanded frontier nodes. Rather than re-insert all frontier nodes, it would be more efficient to re-insert just the best, say, ten percent of the nodes. Using this approach, processes could have their own sub-frontiers, but occasionally "re-mix" their best nodes to ensure uniform heuristic distribution among the processes. The global frontier in this case would not be a truly global frontier, but rather a limited global frontier which served as a mixing pot for the best nodes of each process.

4. <u>Memory Overflow</u>

When a program running on a single processor runs out of memory, that stops the program. But in a concurrent program running on different processors, when the processes on a single processor run out of memory, there are still other processors left as well as perhaps shared memory left. Because a tree search consumes so much memory, and because the processors are running different processes and are potentially running at different speeds, there is a good chance a processor will use all its memory before the program is finished. Some approaches for dealing with this problem are now discussed.

It may be that the operating system handles such situations automatically by transferring processes to other processors. This may not work because the processes take up so much space. Rather than switch a process to another processor, the operating system might place it in shared memory. This, however, may cause a bus bottleneck as the process must execute across the bus. A better operating system approach might be to transfer some of the process' variable space (such as nodes created) to shared memory and keep the process' execution code. This will still cause some bottleneck for across-the-bus memory references, but the bottleneck will be reduced. Operating system approaches to memory overflow have the disadvantage of requiring sophisticated memory management to handle pointer references.

Perhaps a better approach is to explicitly handle memory overflow at the program level. A process requests space from its processor each time it creates nodes. This request for space may be of a form similar to the "new" command of the language Pascal. It is reasonable to require the operating system to return some error condition when a request for space cannot be satisfied. When a process receives such an error code it could coordinate with the director process to send its frontier nodes elsewhere.

Memory overflow should be a consideration in any concurrent program which consumes a lot of memory. If a program is stopped because of lack of space on one processor, unused space elsewhere is wasted. By ensuring a contingency plan for memory overflow exists, then, a program is made more powerful by maximizing overall use of space.

F. FINISHING THE SEARCH

1. Recognizing that the Search is Completed

When each process has no frontier nodes with a better heuristic value than the best solution found, then the tree search is completed. How does the program recognize that the search is completed? A simple and elegant way is for each process to send a completion message to its parent. The completion message indicates that both the process sending the message as well as all of its children processes are done. This approach works for a program which was started by generating a tree-like structure of processes as well as for the simpler case of a single level of processes. When each process finishes, it waits for a completion message from each of its children, if any, then it sends a completion message to its parent. When the director process receives completion messages from all its children, the search is complete.

2. Identifying the Best Solution

Once the search is complete, it is necessary to identify the best solution. An incorrect attempt at identifying the best solution will be considered first to illustrate an important type of message passing error. Assume that, as discussed in the section on promulgating best solutions, the program uses a best solution process for keeping track of the best solution found. When the heuristic director has indication that all processes have finished searching, it asks the best solution process for the best solution (name of the process having a goal node that is part of the best solution). It might seem that the best-solution process would definitely have the best solution. Prior to sending its completion message, any process with a best solution has sent a solution message to the best-solution process. Since the director process has received all completion messages, it seems reasonable that the best solution process has received all best-solution messages. This supposition is wrong. One cannot make any assumptions about the relative speeds of transmission of different messages. It may be that the best-solution message from the process with the overall best solution had not reached the best-solution process when the heuristic director asked for the final best solution. (Note that this problem could nct occur if a shared variable instead of message passing were used for the best solution).

The problem described is an example of a race condition involving message passing (race was discussed in chapter three). A slightly different race condition involving messages was described in the section on promulgating the best solution. The problem discussed there resulted from using more than one process to issue the same type of broadcast message. Both these problems occurred because implicit assumptions were made about the relative ordering between messages sent by different processes.

A correct approach to identifying the overall best solution is easy. Each process can include in its completion message the best solution found between itself and its descendents. When a process receives completion messages from its children, it compares their best solutions against its best solution and forwards the best one to its parent.

3. Outputing the Best Solution

Once the goal node of the best solution has been identified, it is necessary to follow the path from the goal node to the root node to specify the solution. As discussed in the section on memory management problems, this may involve following pointers from the memory space of one process to another. Given that a method exists for doing this (one approach is suggested in the next chapter), the director process can coordinate the following of the path from goal to root node and can coordinate outputing the solution.

G. CHAPTER SUMMARY

Some of the tools discussed in the previous chapter on concurrent programming are inadequate for efficiently using typical concurrent architectures. For example, using the high-level view that each process runs on its own processor and that all processes have easy access to a shared memory leads to the fundamental problem of bottleneck. Thus a different high-level view is needed so that the programmer can achieve effective concurrency on available architectures. In probing different approaches to the tree search program, no particular operating system capabilities were assumed. Rather, for each approach, it was pointed out what operating system requirements would be necessary if that approach were to be used. In other words, in designing an appropriate high-level approach, operating system interfaces were also being designed. This is because the operating system interface determines whether or not the high-level view and its corresponding approach can achieve their potential.

Besides supporting a high-level view, the operating system interfaces that the view requires provide a measure of the worth of that view. For example, if the operating system interfaces become too complex and require too much overhead, then the high-level view is probably inadequate.

This chapter has leaned toward message passing as a high-level tool for supporting an adequate high-level view. One reason for this is that messages can distribute information without the need for a shared memory; hence, bottleneck is avoided. Like other concurrent programming tools, message passing has a potential for race that the programmer must understand. Interestingly, the nature of message passing which causes race also provides advantages. Race can be caused because there are delays between when a message is sent, when it is available for receipt, and when it is received. But these temporal delays also provide less restriction for processes. To-wit: a process can send a message and immediately proceed to do something else; and, a process can check for the existence of a message whenever it wants to--it does not have to be waiting when the message is sent.

The next chapter presents some candidate high-level algorithms for the eight-puzzle program which reflect the analysis of this and the last chapter.

V. A CONCURRENT TREE SEARCH ALGORITHM

In this chapter, a high-level algorithm for a concurrent tree search is presented. The algorithm is explained after first describing the approach taken and the high-level view adopted for writing the algorithm.

A. APPROACH

The approach taken for the algorithm is based on the discussion of the last chapter and is as follows. There is no global frontier; the frontier is divided among search processes. Initial distribution of the frontier is done by "amoeba-splitting" expansion of processes based on operating system messages for processor availability. A director process creates the first search process and then waits for a search complete message.

When a search process finds a new best solution, it promulgates the value of the solution directly to other search processes by a broadcast-queue message. Uniform heuristic frontier distribution is dynamically controlled by priority driven process creation. Processes detect the need to expand their frontier by internal comparison instead of inter-process comparison of heuristic worth.

Memory overflow is also considered. If there is no memory available when a process attempts to create a node, the process: (a) creates a new process and passes the frontier to it; (b) sends a completion message. The operating system will send the newly created process to another processor with sufficient memory, if any is available.

Processes signal completion of their search by sending a message to their parents. Completion messages include value of the goal node representing the best solution found. When the director process receives a search complete message, it creates an output-result process which outputs the solution by tracing the path from the goal node to the root node. The output-result process follows a node pointer by sending a message to the search process which has the node in its memory space; the search process follows the pointer and returns the node to the output-result process. This inter-process approach of following pointers precludes the need for operating system inter-processor memory references.

B. HIGH-LEVEL VIEW

As a basis for writing the algorithm, a high-level view is taken which includes the available means of process interaction as well as the available operating system interfaces. In turn, the basis for the high-level view is the (also high level) architecture view of Figure 7. However, the high-level view is valid for any architecture which supports the details of the view.

In the high-level view, shared data structures may be used, but accessing them causes a bottleneck. Processes may communicate by message passing. There is an arbitrary delay between the time a message is sent and the time it is available for receipt. However, sending a message and checking for the existence of a message (by the EXIST-RECEIVE( ) operation) causes a negligible delay.

Each process, conceptually, has its own memory space, or bucket. A process cannot access another process' memory space (e.g., by variable or pointer references). Although a pointer value cannot be followed outside a process' memory space, the value of the pointer may be passed between processes.

Processes each run on their own virtual processor. However, there may be more processes than physical processors. Physical processor allocation is done by the operating system interface of PROCESSOR-AVAILABLE messages. On the other hand, creating a process without first receiving a PROCESSOR-AVAILABLE message creates a process on a virtual processor; that is, the process may be created on a processor which has other running processes. To allow the programmer the ability to partially control the distribution of processes among physical processors, the operating system considers the priority of a newly created process in determining which processor the process will run on.

The operating system interfaces contained in the high-level view include: existence of operating system messages for allocating processors (PROCESSOR-AVAILABLE MESSAGE), the ability for a process to update its priority by sending a PRIORITY-UPDATE message to the operating system, capability for dynamic process creation, and availability of dynamic memory allocation. Additionally, if memory requested is not available, the operating system returns an error condition rather than stopping the requesting process.

C. ON THE STRUCTURED USE OF MESSAGES IN PROGRAMS

As a prelude to explaining how the algorithm works, it is appropriate to discuss a methodology for structuring message declarations which improves the clarity of how the processes interact.

Because the use of message passing involves spreading messages textually throughout the text of different processes, understanding the message passing interactions between processes becomes difficult. With mutual exclusion, this problem is conquered by the monitor/abstract data type notion of grouping the operations textually with the shared data. Because of their nature, messages need to be spread throughout the text. Hence, it becomes even more important to structure a program so that the use of message passing is clear. Since all messages are listed there, the declaration section of the program is a logical place to describe the intended use of the messages.

Each message in a program has a purpose. Sometimes several messages together serve a common purpose. For clarification, those messages serving a common purpose should be grouped together. An informal term describing a group of messages which serve a common purpose is "message system." It is useful to specify the purpose of each message as well as the purpose of each group of messages forming a message system. Additionally, it helps to name the intended receiver and sender of each message.

Such message structuring is done in the program type section of the algorithm of Figure 8. The only message system specified there is used to trace the solution from the goal node to the root node. For this message system the director process is "in charge"--it initiates a message request for the node pointed to by a pointer. A search process responds with a message containing the value of the node.

By studying the message declarations, then, a person should be able to glean an understanding of how the processes interact. If the message documentation is adequate, the message-passing aspect of the program should be clear and hence easy to maintain.

The process message interactions of the algorithm in Figure 8 are not complex, but one can imagine programs where the message interactions become very complex. For example, an approach not used in this algorithm is to have a heuristic-director process which directs a "hot" process to send part of its frontier to a specified process. This approach is used for maintaining a uniform heuristic distribution and was discussed in the previous chapter. A message system which accomplishes such a transfer can become complex.

A scenario for such a transfer might involve these transactions: Process A sends a message to a heuristic-director process indicating a change in Process A's heuristic value; the heuristic-director determines that Process A has a much better frontier than Process B; the heuristic-director sends a message to Process A telling it to send part of its frontier to Process B; Process A receives the heuristic director's message and reacts by sending a message containing frontier nodes to Process B; Process A sends a message to the heuristic director that the frontier transfer has been started; and, Process B receives Process A's message and sends a message to the heuristic-director that the transfer is complete.

The message system for effecting such a heuristic transfer would thus involve numerous message types, each with a particular purpose. If such a system were used without explaining the workings of the messages as a group, it would be very difficult to decipher the process interactions.

D. ALGORITHM SYNTAX

Figure 8 is the algorithm for a concurrent tree search. The algorithm is written in a Pascal-like ad-hoc algorithmic language.

Program structure of the algorithm is similar to that of Pascal. In the type section of the program, messages are declared as well as shared variables (there are no shared variables for this algorithm). Next, process types are declared. A process type looks like the declaration of a subroutine but is used differently. An instance of a process type can be created by a create-process call. A create-process call requires these arguments: name of the process type, any parameters the process type requires, and the initial-priority of the created process. Any number of instances of a process type can be created dynamically by the program or by any process.

Within the declaration of a process type, message bin-names as well as shared and local variables are declared. Except for operating system messages, each bin-type message is differentiated from the message-type declared in the program by appending a "1" to the program message-type name.

Thus, the program message-type "new-best-solution" becomes the bin-name "new-best-solution1" in the processes. Different bin-types are not needed for each program message type, so there is a one-to-one correspondence between program message-bin-types and process message-types.

In designating the receiver or sender of a message, a process may use the name of a process type. This is equivalent to specifying all existing instances of that process type. A process may also use the keyword "parent" in designating the sender or receiver of a message. This is equivalent to specifying the name of the process' parent (if there is no parent, using this keyword is an error). Another keyword available is "self"; when a process uses "self", it is translated into the name of that process.

The Pascal practice of specifying a field of a record by placing a period between the record name and field name is followed. This is required for accessing the contents of most of the messages. The algorithm uses the following conventions: /slashes enclose comments/ and {brackets enclose descriptions of code}.

```
                    PROGRAM CONCURRENT-TREE-SEARCH

TYPE

 /***********
 * MESSAGES *
 ***********/

 /SENDER/            /RECEIVER/          /MESSAGE/
 /Operating System/  /Search-Process/    PROCESSOR-AVAILABLE=MESSAGE
                                                             CONSUMABLE
                                                             NO-CONTENT

                                         /OS Interface-OS sends message when
                                         there is a processor available/

/-----------------------------------------------------------------------------/

 /Search Process/    /Operating System/  PRIORITY-UPDATE  =  MESSAGE
                                                             CONSUMABLE
                                                             RECORD
                                                              Proc-name:
                                                               {Process name
                                                               type}
                                                              Priority:
                                                               {Range of
                                                               priority}
                                                             END /RECORD/
                                         /OS  Interface - A  search  process
                                         sends message to operating system to
                                         update its priority/

/-----------------------------------------------------------------------------/

 /Search-Process/    /Search-Process/    NEW-BEST-SOLUTION = MESSAGE
                                                             BROADCAST-QUEUE
                                                             INTEGER
                                         /Used  to  promulgate  cost of a new
                                         best solution/

/-----------------------------------------------------------------------------/
```

Figure 8. Concurrent Tree Search Program

```
/Search-Process/   /Search-Process    FINAL-RESULT-OF-
                   or Director        PROCESS   =        MESSAGE
                   Process/                                CONSUMABLE
                                                           RECORD
                                                            Solution Found:
                                                             Boolean
                                                            Solution-Node:
                                                             Node-Record
                                                           END /RECORD/
                                      /Used to signal completion of a
                                      process to its parent, and pass
                                      best solution, if any/
/-----------------------------------------------------------------------------/

/Below Message System of two messages is used for tracing a solution from
goal to root node. The output-result process is in charge.  Search
processes respond to output-result process./
/Output-Result     /Search Process/   REQUEST-NODE-PTR =  MESSAGE
Process/                                                   CONSUMABLE
                                                           POINTER
                                      /Requests from a search process the
                                      node which is pointed to by pointer/

                   /-------------------------------/

/Search-Process/   /Output-Result     RESPONSE-NODE  =    MESSAGE
                   Process/                                CONSUMABLE
                                                           NODE-RECORD
                                      /Used for resonse to above request-
                                      node-ptr message/
/-----------------------------------------------------------------------------/

 /****************
 * NON-MESSAGES *
 ****************/

Node-Record   =    Record             /{ } means "applicable type"/
                      State: { }      /State representation of this node/
                      Operator: { }   /Operator applied to parent node to
                                                 produce this node/
                      Fhat: { }       /Evaluation of node/
                      Proc-Con-       /Process,if any, which contains the
                       taining                 parent of this node/
                       Parent: { }
                      Pointer-to-     /Value is meaningful only within
                       Parent:        "Memory Bucket" of Proc-Containing
                        Pointer                                 Parent/
                   End /Record/
```

Figure 8 contd.

```
  FRONTIER-TYPE = {Whatever data structure is used for frontier list of
                   nodes}

/***************************************************************************/

/******************
* DIRECTOR PROCESS *
******************/

TYPE

  DIRECTOR = PROCESS (ROOT-NODE, GOAL-NODE:  NODE-RECORD)

  VAR
    Final-Result-of-Process1:  Final-Result-of-Process /Message/
    Frontier:  Frontier-Type

  BEGIN /DIRECTOR PROCESS/
    CREATE-PROCESS (SEARCH-PROCESS, FRONTIER, GOAL-NODE, PRIORITY)
    AWAIT-RECEIVE (FINAL-RESULT-OF-PROCESS1)/Go to "Sleep" till search done/
    IF FINAL-RESULT-OF-PROCESS1.SOLUTION-FOUND = FALSE THEN
      {Output "Program Done, No Solution Found"}
    ELSE
      CREATE-PROCESS (OUTPUT-RESULT, FINAL-RESULT-OF-PROCESS.SOLUTION-NODE,
                                                                  PRIORITY)
    END IF
  END /DIRECTOR PROCESS/

/***************************************************************************/

/***********************
* OUTPUT-RESULT PROCESS *
***********************/

TYPE

  OUTPUT-RESULT-PROCESS = PROCESS (GOAL-NODE:  NODE-RECORD)

  VAR
    CHILD-NODE:  NODE-RECORD
    IS-ROOT:  BOOLEAN

  PROCEDURE GET-PARENT-NODE (CHILD-NODE, PARENT-NODE:  NODE-RECORD; IS-ROOT:
                  BOOLEAN)
    /This procedure is passed Child-Node.  It returns the Parent-Node of
    Child-Node and the boolean Is-Root which indicates if the parent-node is
    the root-node. The pointer to the parent of child-node is followed by
    using message communication with the process which contains the parent-
    node/
```

Figure 8 contd.

```
    VAR
      REQUEST-NODE-PTR1: REQUEST-NODE_PTR /Message/
      RESPONSE-NODE-1:  RESPONSE-NODE /Message/

    BEGIN  /GET-PARENT-NODE-PROCEDURE/
      REQUEST-NODE1 <-- CHILD-NODE.POINTER-TO-PARENT
      SEND (REQUEST-NODE1, CHILD-NODE.PROC-CONTAINING-PARENT)
      AWAIT-RECEIVE (RESPONSE-NODE1)
      PARENT-NODE <-- RESPONSE-NODE1
      IS-ROOT <-- (PARENT-NODE.POINTER-TO-PARENT = NIL)
    END   /GET-PARENT-NODE PROCEDURE/

/---------------------------------------------------------------------------/

  PROCEDURE STACK (NODE:  NODE-RECORD)
    /This procedure places nodes along the path from goal to root note on a
    stack.  This is done to get the solution in the correct order for
    output/
  END /Stack/

/-------------------------------------------------------------------------/

  PROCEDURE /OUTPUT-STACK/
      /This procedure outputs the solution  which was stacked. For the top
    node on the stack (the Root-Node), only the state representation is
    output.  For the rest of the nodes, the operator which produced that
    state is output followed by the state representation/
  END /OUTPUT-STACK/
/-------------------------------------------------------------------------/

BEGIN /OUTPUT-RESULT PROCESS/
    STACK (GOAL-NODE)
    CHILD-NODE <-- GOAL-NODE

    REPEAT
      GET-PARENT-NODE (CHILD-NODE, PARENT-NODE, IS-ROOT)
      STACK (PARENT-NODE)
      CHILD-NODE <-- PARENT-NODE
    UNTIL IS-ROOT

    OUTPUT-STACK

END /OUTPUT-RESULT PROCESS/

/*************************************************************************/
```

Figure 8 contd.

```
/*****************
* SEARCH PROCESS *
*****************/

TYPE

  SEARCH-PROCESS = PROCESS(FRONTIER, GOAL-NODE)

  VAR
    NEW-FRONTIER:  FRONTIER-TYPE
    FRONTIER-EMPTY:  BOOLEAN  /Indicates if frontier has any nodes in it/
    NUMBER-OF-NODE-EXPS-BET-COMM-CHECK:  INTEGER /Indicates No. of node
                                   exp's done before checking for messages/
    BEST-SOLUTION:  INTEGER  /Value of best solution found/
    NO-OF-CHILD-PROCESSES:  {POSITIVE INTEGER} /Number of processes created
                                                        by this process/

    NEW-BEST-SOLUTION1:  NEW-BEST-SOLUTION  /Message/
    FINAL-RESULT-OF-PROCESS1:  FINAL-RESULT-OF-PROCESS  /Message/
    /MESSAGE-SYSTEM/
      REQUEST-NODE-PTR1:  REQUEST-NODE-PTR
      RESPONSE-NODE1:  RESPONSE-NODE
    /END MESSAGE-SYSTEM/

/---------------------------------------------------------------------------/

PROCEDURE SPLIT-FRONTIER (FRONTIER, NEW-FRONTIER: FRONTIER-TYPE)
  /This procedure creates a new frontier with some of the best nodes from
  frontier.  New-frontier is used by a newly created process/
END /PROCEDURE SPLIT-FRONTIER/

 /--------------------------------------------------------------------------/

 PROCEDURE EXPAND-NODE
  /WITHIN EXPAND-NODE PROCEDURE IS INCLUDED THE FOLLOWING FOUR SITUATIONS/

    /(1) MEMORY OVERFLOW - WHEN REQUESTING SPACE FOR CREATING NEW NODES/

      IF {No memory available} THEN
        CREATE-PROCESS (SEARCH-PROCESS, FRONTIER, GOAL-NODE, PRIORITY)
        {EMPTY THE FRONTIER OF NODES}
        FRONTIER-EMPTY <-- TRUE
      END IF

    /(2) WHEN CREATING CHILDREN NODES/

      Proc-Containing-Parent <-- Self       /Indicate self as parent of
                                            node. Necessary for following
                                            pointers during output/
```

Figure 8 contd.

```
/(3) PROMULGATING BEST SOLUTION/
IF SOLUTION FOUND THEN
  WHILE EXIST-RECEIVE (NEW-BEST-SOLUTION1) DO
    {Update best solution and frontier}
  END WHILE

  IF {Value of solution found is better than best solution} THEN
    BEST-SOLUTION <-- {Value of solution found}
    NEW-BEST-SOLUTION1 <-- BEST-SOLUTION
    SEND (NEW-BEST-SOLUTION1) /Promulgate a New Best Solution/
  END IF
END IF

/(4) CHANGE IN HEURISTIC WORTH AND HOT PROCESS ACTION/

IF {Significant change in heuristic worth of frontier} THEN
  {Update Priority}
END IF
SEND (PRIORITY-UPDATE, OPERATING-SYSTEM)

IF {Process is "hot" and thus needs help with frontier} THEN
  Split-Frontier (Frontier, New-Frontier)
  {assign priority for new process}
  CREATE-PROCESS (SEARCH-PROCESS, NEW-FRONTIER, PRIORITY)
END IF

END /PROCEDURE EXPAND-NODE/

/-------------------------------------------------------------------------/

BEGIN /SEARCH PROCESS/

  /START UP/
    /Following takes care of initial process expansion to fill processors/
    WHILE EXIST-RECEIVE (PROCESSOR-AVAILABLE) DO      /OS Interface/
      {Expand-Nodes until frontier is large enough to split}
      SPLIT-FRONTIER (FRONTIER, NEW-FRONTIER)
      {Assign priority for new process}
      CREATE-PROCESS (SEARCH-PROCESS, NEW-FRONTIER, GOAL-NODE, PRIORITY)
    END DO
```

Figure 8 contd.

```
   /SEARCH/
    FRONTIER-EMPTY <-- FALSE
    REPEAT
      i <-- NUMBER-OF-NODE-EXPS-BET-COMM-CHECK
      REPEAT
        EXPAND-NODE
        i <-- i - 1
      UNTIL FRONTIER-EMPTY OR (i = 0)

      /Check for messages/
      WHILE EXIST-RECEIVE (NEW-BEST-SOLUTION1)  DO
        {Update best-solution and frontier}
      END DO
    UNTIL  FRONTIER-EMPTY


   /FINISH/
     /Promulgate to parent that search completed and best-solution found/
     IF NO-OF-CHILD-PROCESSES > 0 THEN
       FOR j <-- 1 TO NO-OF-CHILD-PROCESSES DO
         AWAIT-RECEIVE (FINAL-RESULT-OF-PROCESS1)
         /Keep best solution found between children and self/
       END DO
     END IF
     {Insert best solution into Final-Result-of-Process1 Message}
     SEND (FINAL-RESULT-OF-PROCESS1, PARENT)


     /OUTPUT BEST SOLUTION/
       /Wait for messages requesting node in solution path/
       DO
         AWAIT-RECEIVE (REQUEST-NODE-PTR1, OUTPUT-RESULT)
         RESPONSE-NODE1 <-- ^.REQUEST-NODE-PTR1)/Follow pointer to requested
                                                                       node/
         SEND (RESPONSE-NODE1, OUTPUT-RESULT) /Send node  found to  output-
                                                            result process/
       FOREVER
  END /SEARCH PROCESS/
/***************************************************************************/

 /**********
 * PROGRAM *
 **********/

 BEGIN /PROGRAM/
  CREATE-PROCESS (DIRECTOR, ROOT-NODE, GOAL-NODE, PRIORITY)
 END /PROGRAM/
```

Figure 8 conti.

E. EXPLANATION OF ALGORITHM

This section explains the algorithm. First, an outline of the algorithm is presented so that the reader will have a summary of the names and order of the processes and procedures. Next, a complete description of how the algorithm works is given.

1. Outline

An outline of the algorithm is given below:

```
Program Concurrent Tree Search
    Message Declarations
    Non-Message Declarations
Director Process Type Declaration
Output-Result Process Type Declaration
    Procedure Get-Parent-Node
    Procedure Stack
    Procedure Output
    Output-Result Process Code
Search Process
    Procedure Expand-Node
        (1) Memory Overflow Actions
        (2) While Creating a Node, Insert Self
        (3) Promulgating Best Solution
        (4) Change In Heuristic Worth and Hot Process
            Action
    Code For Search Process
Code for Concurrent Tree Search Program
```

Note that there are only three process types: the director process, the output-result process, and the search process. Only one instance of the director and output-result process are created, whereas numerous instances of the search process are created.

2. During Search

How the program functions while the search is in progress will be explained first. An explanation of how the program gets started and how it finishes will be given later.

Look at the declaration of the search process type. Under the word "type", the search-process is declared as a process type which is passed a frontier and goal-node when created. After the declaration is procedure Split-Frontier and procedure Expand-Node. Following the Expand-Node procedure is the code for the search process. Note that the search process code is divided into four sections: start up, search, finish, and output best solution. Look at the section for search. When the search is in progress, each search process executes this section of code. As long as the frontier has nodes in it (frontier-empty = false), the search process goes through the cycle of expanding nodes and checking for messages. The number of node expansions between checking for communications is specified by the variable Number-Of-Node-Exps-Bet-Comm-Check. In this algorithm, the only communication check is for a new-best-solution message; in the other tree-search approaches previously discussed, there may be more external communications.

A node is expanded by calling the procedure Expand-Node. This procedure expands a node from the frontier into children nodes and checks for the occurrence of a goal node among the children nodes. Only four situations within this procedure are shown.

First are memory overflow actions. If there is no memory available to create a child node, a new search process is created and the entire frontier passed to it. The motivation for creating a new process is that the operating system will place it on a processor which has available memory space.

The second situation shown in Procedure Expand-Node occurs during creation of a child node. The search process inserts its name into the proc-containing-parent field of the node record. This is necessary for following the child-node's pointer to its parent. To follow the pointer it is necessary not only to know the pointer value but also to know the name of the process containing the parent node. One can think of the pointer value as being a composite value consisting of a memory address and process name.

Procedure Expand-Node next takes action necessary if it finds a solution. First it checks if there are any new best-solutionl messages so it can compare the value of its solution with the most current best solution. If the solution found by Procedure Expand-Node has a better value than other solutions found, a message is sent to other search processes promulgating this new solution value.

The final situation shown in Procedure Expand-Node is that of a significant change in heuristic worth. If the procedure determines that the heuristic worth of its frontier has changed significantly it sends a message to the operating system to update the priority. This is done so that the operating system can allocate processor time accordingly among search processes on the same processor. If the procedure further determines that the change in heuristic worth was sufficient to warrant "getting help", it creates a new search process and passes part of the frontier to it.

After the search process has done the required number of node expansions by calling procedure Expand-Node, it checks for the existence of any new-best-solutionl messages. If any exist, the best solution is updated and the frontier purged of any nodes which have less potential than the new solution.

The search process continues the cycle of expanding nodes and checking for new-best-solution messages until there are no more nodes in the frontier to expand.

3. <u>Start Up</u>

Look at the program section of the algorithm, located at the bottom of the code. When the program is started, this section is executed. The program code consists of only one statement which creates an instance of the director process type, passing it the values of the root and goal nodes. Now look at the code for the director-process type to see what it does. It creates an instance of the search-process type and waits for a message from that search process signalling that it is done. Thus, the director process "goes to sleep" until the search is finished. With the director process asleep, look at the start-up section of the search process to see what is done next during the start up of the search. The search process created by the director process checks for the existence of a processor-available message (which is sent by the operating system). If such a message exists, the search process creates another search process and passes part of the frontier to it. As long as processor-available messages exist, each search-process will create new search processes. In this manner, search processes are shared on each processor with the frontier divided among them. When no more processors are available, each search process proceeds to the search portion of its code.

4. Completion of Search

A search process has completed its search when its frontier is empty. When that happens, a search process enters the section of its code titled "Finish". In the finish section, a search process first waits for receipt of completion messages from each of the search processes it created during start up. The completion message is called Final-Result-of-Process1, and it indicates not only that a process has completed its search, but also includes the value of the goal-node representing its best solution (if any). When a search process has received completion messages from all its children, it picks the best solution from among the solutions of its children and its own solution (if any). This best solution, if any, is then sent by the search-process to its parent. In this manner, all search processes report their completion until finally the director process receives a completion message from the search process it initially created. The director process then knows that all search processes have completed and also knows the best solution found.

Now look at the director-process code. At the point just described, it has received the message which it has been "asleep" waiting for since it created a search process. It now examines the final-result-of-process1 message to ascertain if any solution has been found. If so, it creates an output-result process, passing it the goal-node which is part of the best solution.

5. Solution Output

At the time when an output-result process is created, the search is complete and the best solution identified. All that remains to be done is to trace the solution from the goal-node to the root-node by following the pointers contained in each node record of the solution path. A pointer can be followed only within the process memory space to which the pointer applies. Thus, for the output-result process to follow a pointer, it must request help from the search process containing the node pointed to by the pointer. This is the reason that each search process placed its name in a created node record. The output-result process, using the get-parent-node procedure, sends a message containing the pointer value to the process specified in the "proc-containing-parent" field of the node record.

Look at the output best solution section of the search process type. After sending a completion message to its parent, each search process waits for a request-node-ptrl message from the output-result process. When a search process receives such a message, it follows the pointer value of the message (using the Pascal "follow-pointer" operation: "↑.") and retrieves the desired node. The value of this node is sent to the output result process and the search process again waits for another node request message.

In this manner, the output-result process traces the solution path from goal to root node. Each node is stacked (by the stack procedure) to place the solution in the correct order. The trace of the solution path is terminated when the root node is reached (determined by recognizing that since the root has no parent, its pointer-to-parent field has the value nil). After the entire solution path has been stacked, the solution is output from the stack. The output of the solution completes the program. (Note that the search processes never "finish"; they are in infinite loops waiting for node request messages).

The reader may wonder why the output-result process was written as a process and not as a subroutine of the director process. After all, only one instance of the output-result process was ever created. The output-result process was written as a process to allow flexibility in changing the program. Consider the situation when it is desirable for the concurrent-tree-search program to output not just the best solution, but several solutions. At the end of the search, the director process could receive several solution nodes. With only minor modifications to the program, the director process could create an instance of the output-result process for each goal-node found. The modification required is for each output-result process to specify its name in the request-node message it sends to search processes. A search process would address its response-node message to the output-result process that sent it. In this manner, any number of output-result processes could concurrently trace and output the same or different solutions. A little thought should convince the reader that this is possible, and that there is no chance of a race condition (such as two output-result processes "mixing up" the solution paths they were tracing).

6. Message Declarations

If the reader hasn't done so already, it is a good time to study the message declarations. The stated purposes of the different messages should be sufficient to give the reader an overall view of the process interactions which occur in the program. In fact, studying the message declarations is a good starting point for understanding a concurrent program which is based on message passing.

F. CHAPTER SUMMARY

One algorithm for concurrently solving a tree-search problem such as the eight-puzzle has been presented. It is again stressed that the design of the algorithm was part of of the design of a high-level view. Design of the high-level view also included the design of necessary operating system interfaces.

The algorithm illustrates one way of performing a concurrent tree search; it should not be construed as the best approach of those discussed in the previous chapter. Empirical testing is needed before final evaluation of the various approaches can be made.

VI. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

Presented in this chapter are a summary of the thesis, general conclusions, and recommendations for further study.

A. THESIS SUMMARY

Tree search, as typified by the eight-puzzle problem, is fundamental to the field of artificial intelligence. Even with good heuristic functions, the time it takes on a single processor to solve progressively more difficult tree search problems grows exponentially and quickly becomes constraining. It seems reasonable that the use of concurrency should significantly improve the speed of a tree search.

As background, Chapter Two discusses the fundamentals of tree search using the eight-puzzle problem for an example.

In Chapter Three, an overview of concurrent programming issues is presented. The problem of mutual exclusion illustrates key differences between sequential and concurrent programming. However, the use of mutual exclusion for structuring a problem should generally be avoided if other approaches are available. When used, mutual exclusion should be relegated to a lower hierarchial level, preferably by incorporation into abstract data types.

Precedes relation and synchronization are examples of high-level concurrent approaches to structuring certain types of problems. Problems can also be structured by using the high-level tools of message passing. The differences between sequential and concurrent programs is illustrated by the range of nondeterminism concurrent programs can exhibit.

With concurrent programming issues as background, Chapter Four develops some approaches for a concurrent tree search program. A good high-level approach should present a view to the programmer which is conducive to writing programs that run efficiently on the underlying architecture. As a result, development of a high-level approach includes development of operating system interfaces.

Based on one of the approaches in Chapter Four, Chapter Five provides a high-level algorithm for solving the eight-puzzle problem. The algorithm uses message passing as the basis for its high-level structure. It should not automatically be construed as the best algorithm for solving the problem. It, as well as other approaches, need to be evaluated empirically.

B. BACKGROUND SUMMARY

Nils71 is an excellent textbook on artificial intelligence problem solving and contains the fundamentals of tree search.

Dijk68A is the pioneering work on the mutual exclusion problem and introduces P and V, and semaphores. Brin73 includes further work on mutual exclusion. Monitors are described in Hoar74 and are based on some of the concepts of Brin73. For a discussion on the traditional (non-concurrent) notion of abstract data types, see Horo76. The heap structure is the basis for the concurrent priority queue example of Appendix A and is explained in Aho75.

The precedes relation specification is introduced in this thesis. Alternate methods of specifying a precedes relation are the FORK operation found in Conw63 and the CoBegin/CoEnd construct first introduced as ParBegin/ParEnd in Dijk68A.

Path Expressions are a means of synchronizing and are described in Habe75 and Andl78. The synchronization primitives eventcounts and sequencers are described in Reed79.

The message system developed in Chapter Three is not based specifically on any references and some of the notions discussed are new. Brinch-Hansen discusses messages in Brin73. He discusses the notion of messages with no content in his section on semaphores [Brin73: p. 93]. The type of messages discussed in Brin73 require an explicit queue and are consumable (although Brinch-Hansen doesn't use that term). Holt78 (p. 31) describes messages which are addressed by specifying the receiver. The UNIX Operating System

(Ritc74) uses messge passing via pipes; pipes are somewhat similar to the bin concept discussed in this thesis. The object-oriented programming language Smalltalk [Gold81] is based on objects which respond to messages. Languages based on actor semantics [Grei75, Hew77A, and Hew77B] are based on message passing to a greater degree than the message-system of this thesis.

For further reading which presents good overviews of concurrent programming, see Cali82 and Brya79. For a discussion of hierarchical structuring, see Dijk68B.

The problem of a bottleneck due to architecture was recognized as early as 1963 by Conway [Conw63]. Conw63, incidentally, presented some keen insights into concurrency. In the literature review done for this thesis, little detailed work was found on concurrent tree search issues. Consequently, most of the issues developed in Chapter Four are original. Although he solves a different problem, Kornfeld in Korn81 discusses some critical notions of concurrent heuristic search. For example, the notion of a "hot" process was gleaned from Korn81. Related articles on concurrent tree search are Fish80 and Imai79. Fish80 discusses an alpha-beta type tree search which uses message passing (principles of alpha-beta search are found in Nils71). Imai79 uses the idea of a current best solution.

Sugi81 was the one reference found which most closely related to the problem this thesis tries to solve. The article presents a concurrent Lisp solution to the eight-puzzle problem. In the solution, the frontier of the search tree is divided among processes and a director-type process (they call it a monitor) selects the best processes to run. Both mutual exclusion and message passing are used as a basis for the solution. The article does not addresss many issues of concurrent tree search, but rather is written primarily to present a solution.

There are some similarities between the syntax of the Chapter Five algorithm and the concurrent Pascal of Brin77. One primary difference is that concurrent Pascal does not allow dynamic process creation whereas the algorithm of this thesis does.

C. CONCLUSION

Foundations have been laid for evaluating and choosing an efficient high-level approach to a concurrent tree search problem. It is believed that the result of further work will not only be an effective program for concurrent tree search, but will also be a high-level approach to structuring concurrent programs that is useful for other applications.

D. RECOMMENDATIONS

It is recommended that empirical tests on an appropriate architecture be done to refine and evaluate some of the tree search approaches suggested. At a minimum, the following approaches should be evaluated:

1. Global frontier;

Combinations of the following

2. With and without a limited global frontier;
3. With and without best solution propagation;
4. With and without detection and correction of non-uniform heuristic distribution using combinations of
    a. Heuristic detection by internal comparison; by external comparison.
    b. Heuristic distribution by priority driven process creation; by passing to existing processes.

Evaluation considerations should include:

1. Strengths and weaknesses of message passing;
2. General usefulness of high-level programming views, including operating system interfaces.

In addition to empirical tests, mathematical analysis is recommended for questions such as:

1. How much frontier expansion is necessary for a good initial heuristic distribution?
2. How often should processes check for external communications such as promulgation of a new best solution?

APPENDIX A

IMPLEMENTATION OF A CONCURRENT PRIORITY QUEUE USING A HEAP

Aho75 (pp. 87-92) discusses the properties of a heap. A heap is a binary tree such that: values are "stored" at each node, all leaf nodes must be located at depth d or d + 1, and the leaves at the lowest level must be as far "left" as possible. Furthermore, the heap must satisfy the heap property: the value of each node is greater (less if the heap is ordered by the smallest rather than largest value) than the value of each of its children nodes (if the node has any children). It is convenient to use an array for representing a heap because of the ease of calculating the location of a child or parent of a node. The children (if they exist) of a node located at the ith slot of an array are located at the (2i) and (2i + 1) slots of the array.

An insertion into a heap is done by creating a new leaf node and placing in it the value to be inserted. To maintain the heap property, the newly inserted node must be "bubbled up" the tree. The new value is compared to the value of its parent node; if the new value is greater (less), the values of the two nodes are exchanged. A new value bubbles up in this manner until no exchange is necessary or the new value is at the top of the tree. In the array, this corresponds to calculating slot numbers of the array and exchanging values.

Concurrent insertions can be allowed by "locking" only the ncessary slots of the array as follows. The slot into which a new value is initially inserted is first locked (call it the child slot). Then the parent location is locked. A comparison is now made between the values in the two locked slots and, if required, a swap made. Following this, the child slot can be released. If an exchange was made, then the parent slot (which is still locked) is now the child slot, and a new parent slot is locked. This cycle of locking a parent slot, comparing values, releasing the child slot, and locking a new parent slot continues until an exchange is required or the value bubbles to the top of the heap. Any remaining locks are released when the "bubble-up" is completed. Some thought should convince the reader that any number of concurrent insertions can be done in this manner with no chance of deadlock or race.

A deletion from a heap is done by "removing" the value of the root node. This leaves an empty node in the heap; to fill it, the value of the rightmost of the lowest leaf nodes is placed in the root node and that leaf node deleted. The new value at the root must be "bubbled down" in the heap to preserve the heap property. This is done by comparing the value of the root node with the values of its children nodes. The value at the root node is exchanged with the greater (lesser) of the values of the children. In this manner, the new root node bubbles down the tree until no exchanges are necessary or the value arrives at a leaf node.

Deletions can be done concurrently in a manner similar to concurrent insertions. Instead of locking one node for an exchange as required for insertions, up to two nodes (the children) must be locked. However, there is a problem. With each deletion, the size of the heap decreases. Thus, when a value is bubbling down, the size of the heap may be changing if other deletions are concurrently taking place. Changing of the heap size wasn't a problem with an insertion because the size of the heap mattered only during the first part of the insertion (creation of a leaf node). In a deletion, however, the value bubbling down will not know where to stop. This problem is solveable (although the author doesn't know an "elegant" solution), but to simplify the solution shown in this appendix, only one deletion at a time will be allowed.

Allowing insertions and deletions to occur concurrently makes the problem more difficult. When an insertion value bubbling up "meets" a value bubbling down from a deletion, there is a deadlock. If the deadlock is somehow resolved in favor of one of the values, then another problem occurs. Consider an example. Let procedure Insert be in the process of bubbling value I up the heap. Let procedure Delete be in the process of bubbling value D down the heap. When values I and D "meet", there is a deadlock: procedure Insert is requesting a lock on the node containing D and procedure Delete already holds that lock; similarly, procedure Delete is requesting a lock that is already held by procedure Insert. Assume this deadlock is resolved in favor of procedure Delete, i.e., procedure Insert loses its lock to procedure Delete. Further assume that procedure Delete exchanges its value with procedure Insert's value. The problem is that procedure Insert has "lost" the location of its value because procedure Delete just moved it. One can think of solutions to this problem which require communication between procedure Delete and Insert. Because the author has not developed a nice solution to this problem, a solution allowing concurrent insertions and deletions will not be shown.

Based on the previous discussion, the restrictions placed on this problem are: any number of concurrent insertions are allowed, but deletions must be done separately. These restrictions are the same as reader-writer problems which allow concurrent reads but require separate writes. Cour71 gives two solutions to such a reader-writer problem. One solution allows read operations to delay write operations indefinitely and the other solution requires that a write request be honored as soon as possible (i.e., no read requests can be honored if a write request exists). The algorithm of this appendix incorporates the solution of

Cour71 which allows write requests to be delayed indefinitely. This means deletions are delayed as long as insertion requests exist. It is stressed that changing the algorithm to honor deletions as soon as possible would be simple. Such a change could incorporate the other solution of Cour71.

A solution based on P and V operations which allows concurrent insertions but separate deletions follows. It uses the following conventions: /slashes enclose comments/ and {brackets enclose descriptions of code}.

```
SHARED VARIABLES

insertioncount:  integer /initial value = 0/
insertion-sequencer, heapdoor:  semaphore /initial value = 1/
max: {max size of heap array}
N:  1..max /current size of heap/
heap:  array 1..max of integer
heaplock:  array 1..max of semaphore /semaphores initialized
                                                        to 1/
heapsize:  semaphore /initial value = 1; used to obtain
                                          current value of N/

------------------------------------------------------------

INSERTION /any number of concurrent insertions allowed/

VAR
    self:  1..max /current location of number being inserted/
    parent:  1..max /current location of parent of number
                                              being inserted/

/OBTAIN ACCESS TO HEAP/
P(insertion-sequencer)
insertioncount <-- insertioncount + 1
IF insertioncount = 1 THEN P(heapdoor)
V(insertion-sequencer)
```

```
/DO INSERTION/
P(heapsize)
N <-- N + 1
self <-- N
P(heaplock (self)) /lock location of self in heap/
V(heapsize)
Heap (self) <-- {value being inserted into heap}

REPEAT
    {calculate parent's address in heap}
    P(heaplock (parent))
    {do comparison and swap if necessary}
    V(heaplock (self))

    IF {swap not done or at top of heap} THEN
        V(heaplock(parent))
    ELSE
        self <-- parent /number being inserted has moved up
                                                       heap/
    END IF
UNTIL {done} /repeat cycle until no swap done or at top of
                                                       heap/

/LEAVE HEAP/
P(insertion-sequencer)
insertioncount <-- insertioncount - 1
IF insertioncount = 0 THEN V(heapdoor)
V(insertion-sequencer)
```

```
DELETION /only one deletion at a time/

P(heapdoor)
    {do entire deletion}
V(heapdoor)
```

EXPLANATION OF ALGORITHM

The semaphore heapdoor is used to gain access to the heap. Since any number of insertions may be done, only the first insertion of a group of insertions need lock the heap. To accomplish this, the semaphore "insertion-sequencer" is used to ensure only one insertion process (if implemented as a process) requests an insertion entry at a time. The variable insertioncount represents the number of insertions currently in progress. Similarly for "leaving" the heap, only the last insertion process need "unlock" the heap.

The "array" heap represents the heap. To allow locking separate slots of the array, another array called heaplock is used and it contains semaphores initialized to 1. When a value is initially inserted, the size of the heap array must be locked to prevent several processes from inserting values into an already occupied slot. The semaphore heapsize is used for this purpose.

Since the deletion must be done separately, its concurrency considerations consist only of locking and unlocking the heap.

APPENDIX B

DERIVATION OF POURER/TAKER ALGORITHM WITH EVENTCOUNTS AND SEQUENCERS

See the algorithm and explanation of Section III. C. 2.c.

Initial Conditions

An array (presented in Chapter Three as a racetrack) of size N is initially full.

Pourer (Producer)

Let i be the number of completed pours. Let T be the number of completed takes. Since the array is initially full, the number of pours can never exceed the number of takes. Said another way, the number of pours must be less than or equal to the number of takes:

$$i \leq T$$

If the pourer desires to make a pour, it must ensure that the number of pours already made plus the desired pour is less than or equal to T, i.e.,

$$(i + 1) \leq T$$

must be satisfied to proceed with a pour. This is equivalent to $T \geq (i + 1)$ which is specified by:

AWAIT (T, i + 1)

Taker (Consumer)

Let i (a different i than the previous one) be the number of completed takes. Let P be the number of completed pours. Since the array is initially full, the taker can take up to N times more than the producer. (When N times more has been taken than poured, the array is empty or, at best, a pour is in progress but not yet completed). Thus

$$(i - P) \leq N$$

must always be satisf ed. If the taker desires to take, it must ensure that this condition will hold after the take, i.e., after i is one greater. Thus,

$$((i+1)-P) \leq N \;\text{-->}\; ((i+1)-N) \leq P \;\text{-->}\; P \geq ((i+1)-N)$$
$$\text{-->}\; \text{AWAIT } (P,(i+1)-N)$$

specifies this requirement. This completes the derivation.

LIST OF REFERENCES

Aho75 Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysis of Computer Algorithms, Addison-Wesley, 1975.

Andl78 Andler, S., "Predicate Path Expressions", Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1978.

Back78 Backus, J., "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the ACM, 21-8, ACM, New York, Aug, 1978, pp. 613-641.

Brin73 Brinch Hansen, P., Operating System Principles, Prentice-Hall, New Jersey, 1973.

Brin77 Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, New Jersey, 1977.

Brya79 Bryant, R. E. and Dennis, J. B., "Concurrent Programming", Research Directions In Software Technology, ed. P. Wegner, The MIT Press, 1979.

Cali82 Calingaert, P., Operating System Elements, Prentice-Hall, New Jersey, 1982.

Conw63 Conway, M. E., "A Multiprocessor System Design", AFIPS Fall Joint Computer Conference Proceedngs, v. 24, Spartan Books, Baltimore, 1963.

Cour71 Courtois, P. J., Heymans, F., and Parnas, D. L., "Concurrent Control with 'Readers' and 'Writers'", Communications of the ACM, 14-10, ACM, New York, Oct, 1971, pp. 667-668.

Dijk68A Dijkstra, E., "Co-operating Sequential Processes", Programing Languages, ed. F. Genuys, Academic Press, New York, 1968.

Dijk68B Dijkstra, E., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, 11-5, ACM, New York, May, 1968, pp. 341-346.

Fish80 Fishburn, J. P., Finkel, R. A., and Lawless, S. A., "Parallel Alpha-Beta Search On Arachne", Proceedings of the 1980 International Conference on Parallel Processing, ACM and IEEE, 1980.

Gold81 Goldberg, A., "Introducing the Smalltalk-80 System", BYTE, 6-8, Aug, 81. (There are numerous articles on Smalltalk in this issue of BYTE).

Grei75 Grief, I., Semantics of Communicating Parallel Processes, Technical Report TR-154, MIT Laboratory for Computer Science, Cambridge, Mass., Sept 75.

Habe75 Habermann, A. N., "Path Expressions", Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburg, June, 1975.

Hew77A Hewitt, C. and Atkinson, R., "Parallelism and Synchronization in Actor Systems", Principles of Programming Languages, ACM, New York, Jan, 1977, pp. 267-280.

Hew77B Hewitt, C. and Baker, H., "Laws for Communicating Parallel Processes", Information Processsing 77, IFIP, North Holland Publishing Company, Amsterdam, 1977, pp. 987-992.

Hoar74 Hoare, C. A. R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, 17-10, ACM, New York, Oct 1974, pp. 549-557.

Holt78 Holt, R. C., Graham, G. S., Lazowska, E. D., and Scott, M. A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley, 1978.

Horo76 Horowitz, E. and Sahni, S., Fundamentals of Data Structures,Computer Science Press, Rockville, Md, 1976.

Imai79 Imai, M., Yoshida, Y., and Fukumura, T., "A Parallel Searching Scheme For Multiprocessor Systems and Its Application to Combinatorial Problems", Proceedings of the Sixth International Joint Conference on Artiificial Intelligence, Tokyo, August 20-33, v. 1, 1979.

Korn81 Kornfeld, W. A., "The Use of Parallelissm to Implement a Heuristic Search", MIT Artificial Intelligence Laboratory, AI Memo No. 627, March, 1981.

Mins68 Minsky, M., "Semantic Information Processing", The MIT Press, Cambridge, Mass., 1968.

Nils71 Nilsson, N. J., Problem-Solving Methods In Artificial Intelligence, McGraw-Hill, 1971.

Reed79 Reed, D. P. and Kanodia, R. K., "Synchronization With Eventcounts and Sequencers", Communications of the ACM, 22-2, ACM, New York, Feb 79, pp. 115-123.

Ritc74 Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, 17-7, ACM, New York, July 1974, pp. 365-375.

Sugi81 Sugimoto, S., Tabata, K., Agusa, K., and Ohno, Y., "Concurrent Lisp on a Multi-Micro-Processor System", Proceedings of the Seventh International Joint Conference On Artificial Intelligence, IJCAI-81, v. 2, p. 949-954.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Doug Smith, Code 52SC Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	William Shockley, Code 52SP Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	LCDR Curt Powley Naval Submarine School Code 20 SOAC Box 700 Groton, Connecticut 06349	4
7.	Curt Powley c/o John Powley 215 North Maple Gilman, Illinois 60938	3
8.	Linda Widmaier SMC# 2143 Naval Postgraduate School Monterey, California 93940	1
9.	John Hayes, Code 54HT Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1

10.	LT Brenda Selby Navy Regional Data Automation Center Building 8-2 Naval Air Station Alameda, California 94501	1