LANGUAGE FEATURES FOR PROCESS INTERACTION AND ACCESS CONTROL

James Russell McGraw

TR 77-319

A Thesis
Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

LANGUAGE FEATURES FOR PROCESS INTERACTION AND ACCESS CONTROL

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

by

James Russell McGraw

August 1977

BIOGRAPHICAL SKETCH

James Russell McGraw was born on October 20, 1951 in Clarksburg, West Virginia. He graduated from Grossmont High School, La Mesa, California in June, 1969. He received the Bachelor of Science degree in Computer Science from Purdue University in June, 1972, and the Master of Science degree in the same field from Cornell University in June, 1974. He is a member of the Association for Computing Machinery and Phi Beta Kappa.

To my parents

ACKNOWLEDGMENTS

I am deeply indebted to my thesis advisor, Greg Andrews; he has been a constant source of assistance and encouragement. His genuine interest in students and willingness to help have contributed much to this thesis and to my education as well.

Special thanks go to David Gries for his constructive criticisms and careful editing of this document. I am also grateful to the other members of my committee, John Dennis, Anthony Walters, and Norman Lyons for their help and guidance. The excellent typing is the work of Mrs. Roberta Ludgate.

Finally, I want to thank a long-time friend and advisor, Mr. Curtis Fauth, for providing my first opportunity to play with computers.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis examines the problems of representing process interaction and access control in a high-level multiprogramming language. Our work focuses on the task of writing an operating system; however, the results have a much wider applicability. Two basic goals have strongly influenced this research. First, we felt that the language facilities should provide a very flexible environment in which a system designer could easily work. Second, we wanted the language to prevent potentially harmful forms of simultaneous data access (e.g., overlapped modification of a variable) which could cause time-dependent errors. Our contributions to this field of research can be divided into three basic areas:

1. extensions to the monitor concept [33] to improve its flexibility and expressive power,
2. new static and dynamic access control mechanisms to improve control over the use of interaction facilities, and
3. an analysis of some language and program properties (data integrity, deadlock, and access safety) which are affected by interaction mechanisms.

1.1. Problem Framework

Many programming situations can be usefully cast in the form of processes and process interaction. The management of hospital monitoring equipment, the operation of a military defense network, and the control of a computing facility are only a few examples of these situations. In order to focus attention on process interaction we have selected one application which embodies most, if not all, of the problems that occur. Our choice is the control of a computing facility--i.e., the design of an operating system. In addition to providing difficult problems for evaluating various language features, it is sufficiently common so as to be generally understood.

Some operating systems have already been implemented in high-level languages. Burroughs [49] has been using a variant of ALGOL for writing all its systems programs for many years. Another example is the MULTICS [14,43] operating system which was written in PL/1. More recently, Concurrent Pascal [9,10] and Modula [63,64] were designed for this same purpose. We improve on deficiencies of these languages.

We view the operation of a computing facility as the execution of <u>one</u> program written in a high-level language. The structure of that program has the form outlined in Figure 1.1. The top part of the program includes most of the features of a normal operating system. The bottom part represents users currently in the system. This view puts all the problems of defining and controlling interaction directly in the programming language whether it involves users, the operating systems, or the hardware. One advantage of this approach is that the design of an operating system becomes, to a great extent, machine independent. We need not worry about concepts like user-executive modes of execution, memory locks, or address range checking. While a compiler may use them to implement the language design, they need not be present in the language itself.

```
SYSTEM:  begin

         system status information             ] shared data

         PRINTER:  procedure (    );           ]
                                               ] machine interface
                   end;                        ]   procedures
         :
         SPOOLER:  process;                    ]
                                               ] system processes
                   end;                        ]
         :
         CREATE_FILE:  procedure (    );       ] operating system
                                               ] provided facilities
                   end;                        ]
         :
         USER_1:  process;                     ]
                                               ]
                   end;                        ]
         :                                     ] system users
         USER_N:  process;                     ]
                                               ]
                   end;                        ]

         initialization

         end SYSTEM
```

Figure 1.1: Program structure for a computing system.

Within this framework, our areas of interest are the following. First, we examine language tools for permitting a system designer to specify how interaction should occur. We want to give him the flexibility to design, for example, a file system or message-passing system in any way that he chooses. The second area of interest is access control facilities for managing process interaction. The designer may only want certain processes to use a message channel, or he may possibly want them to use it in one specific way (e.g., only receiving messages). Restrictions could be permanent (never use this channel) or temporary (only use it when I give you permission). What language facilities can enforce these types of access policies? Finally, we are interested in some properties of programs which are affected by having interaction facilities. One example is deadlock--can we avoid it, prevent it, or detect it? Similarly we can ask questions about what objects and what information each process can access in a specific program.

In order to even consider these types of questions, we must have some basic assurances from the language; we call them integrity constraints. The first one is language integrity, which basically means that when we write some statement, it is properly translated by the compiler and cannot be later modified. In other words, the language cannot have any features which could be used to subvert the normal meaning and execution of a program. Without this type of assurance, we cannot be sure of what is actually happening. Strangely enough, language integrity is not often available in a sequential language let alone a multiprogramming language. What we will actually do is assume our base (sequential) language has this integrity, and then only add features which cannot be used to violate it.

The second property we need from the language is data integrity. Here, we are addressing the problem of time-dependent errors, which can arise in a multiprogramming environment. Although a language cannot prevent all time-dependent errors,[1] it can prevent one important class, namely simultaneous access to data by more than one process when one of the accesses is for modification. Errors of this type are extremely difficult to find and correct. But more important for our framework, these errors put a great burden on analysis of properties such as information flow and correctness. We will say that a language enforces data integrity if the only simultaneous access allowed is concurrent reading of a data object.

The final language property we need is abstraction integrity. The idea here is that the particular features of a language should support the concepts of modularity and hierarchical design. For example, if some system design uses a message-passing facility, then no language feature should permit a user to access its internal structure. Simply stated, if an abstraction is implemented in some part of a program, we do not want to be forced to examine the entire program in order to insure that it works properly. Abstraction integrity simplifies the understanding of process interaction (which is likely to be an abstraction) and also eases the problems of program and language analysis. This property has guided some recent developments in language design (e.g., CLU [44] and ALPHARD [67]).

---

[1]For example, no language can prevent errors such as designing a message-passing facility that fails to handle the possibility of a process attempting to receive a message before one is ever sent.

1.2. Goals

One of our major goals is the development of expressive language features for process interaction and access control. We are not attempting to design a new language. Our intent is to have features which could be imbedded in many different languages. Our other major goal is to be able to analyze these features with respect to deadlock, access safety, information flow, and correctness. From previous research we know that these properties are very complex, so our aim is mainly to see what can be said about them in one particular environment.

In designing language facilities, many different priorities are possible. Efficiency, clarity, simplicity, and verifiability are only a few examples of commonly used criteria. Except for the integrity constraints discussed earlier, our top priority for designing language tools is what we call expressiveness. In the next chapter we will be more precise in our use of the term; informally it means the ability to represent design concepts in a clear, understandable manner. This choice is based on our philosophy that the key to successfully writing complex programs (such as operating systems) is having an appropriate set of descriptive language tools. One limit we place on this priority is implementability; we do not want to include concepts which could not be incorporated into some language. All of the features we consider meet this requirement. However, if the tradeoff is between expressability and efficiency, we will favor the former. We feel that efficiency arguments become less and less important as the cost of computing time and memory decrease, and we see no substitute for understandable and clear programs.

In the area of analyzing language features, the key question is--What can we hope to accomplish? Properties such as deadlock [34] and safety [30] are almost certainly impossible to precisely characterize in the type of language environment we are considering. Our basic goal here is to develop models for these properties so that specific programs can be analyzed with respect to them. These models should permit us to answer questions like--Is execution of this system currently deadlocked? Can it avoid deadlock in the future? To which objects can this program block acquire access?

1.3. Thesis Overview

The rest of this thesis is organized into eight chapters. Chapter 2 reviews existing mechanisms for defining process interaction and specifying access control. An integral part of this chapter is the definition of our criteria for expressiveness, which is used to evaluate each of the mechanisms discussed. Chapter 3 describes the language base we will use as a foundation for our own language features. This base is needed in order to give programming examples, but it is only one of many suitable languages. In Chapter 4 we present our mechanism for process interaction, the <u>resource</u>. It is a variant of the monitor concept, with several extensions to improve its general utility. Chapter 5 defines our facilities for providing static access control. Our approach permits program control over access rights acquired by a block through scope rules. Chapter 6 defines complementary mechanisms for dynamic control. These facilities, based on the capability concept [24], permit the dynamic sharing of <u>resources</u>. Chapter 7 gives extensive examples of these language features and evaluates their strengths and weaknesses. Chapter 8 analyzes our facilities with respect to data integrity, deadlock, and access safety. Finally, Chapter 9 summarizes our work and mentions directions for future research.

CHAPTER 2

EXISTING INTERACTION AND ACCESS CONTROL TECHNIQUES

A significant amount of work has already been done on language facilities for process interaction and access control. In this chapter we summarize this work and show its relationship to our research. Early mechanisms for process interaction tended to be too primitive and powerful. Not surprisingly, they cannot satisfy our language integrity criteria. More recently, approaches like message passing [7,11,25] and monitors [33] have brought some coherence to process interaction. In order to evaluate the relative merits of these approaches, we develop some expressiveness criteria which puts the analysis in a more objective framework.

The second area of interest is access control mechanisms. Most previous work has attempted to improve a programmer's ability to enforce static access rights. This situation is particularly clear in Concurrent Pascal [9] and Modula [63,64]. Recently, some work has been done to permit dynamic access control, but much remains to be done. For both the static and dynamic cases we again develop expressiveness criteria to help evaluate the current state of the art.

2.1. Primitive Interaction Mechanisms

The earliest approaches to defining process intereaction were based on a simple philosophy: the language should provide sufficient mechanisms to allow a programmer to specify interaction. An important consequence of this philosophy is that the programmer has sole responsibility for making it work properly. We include in this category three basic approaches:

1. inline machine coding,
2. semaphores, and
3. regions.

All of these mechanisms fail to meet at least one of the integrity requirements which were described in Chapter 1. In this section we review each of these systems and discuss why they are inadequate for our purposes.

2.1.1. Inline Machine Coding

The simplest approach to handling process interaction, from a language designer's perspective, is to let the users of the language define it. The designer provides a basic sequential language with one extra feature which allows programmers to intersperse machine instructions with the code generated by the compiler. Using this approach, a programmer can work with high-level tools for general efficiency and understanding, and he can access the full power of the machine for special cases. In particular, he can build his own multiprogramming facilities. XPL [46], BLISS [65], and SUE [13] have made this choice. Since they are all similar, we will use SUE as the example.

In this language the key feature is called INLINE. It has the form:

CALL INLINE(<op code>,<operand-1>,<operand-2>)

This instruction is a message to the compiler to insert a machine operation into the generated code. Hence, it does not result in a procedure inovcation. The op-code indicates which machine instruction is to be inserted, and other fields specify the operands for it. These operands may be specified as variable names, or as base-displacement pairs yielding absolute machine addresses. For example, on an IBM 360 series machine the instruction:

CALL INLINE("82",0,0,2,0);

would generate a load PSW[1] command. The new value of the PSW would come from the contents of memory referenced by the address in register two. INLINE calls may be interspersed throughout a SUE program, and the compiler does no checking when the code is generated for them.

Using INLINE, it is possible to write programs in a multiprogramming fashion even though the language itself does not support the concept. The main program, through INLINE, can manage the states of all other processes and schedule them for execution as it desires. A significant aspect of this approach is that the main program is completely responsible for correctly implementing the concept; the language has no knowledge of the actions being taken.

The INLINE strategy for handling processes and process interaction assumes that these concepts are used too infrequently to justify their inclusion in the language. Each of the languages using this approach was designed to meet specific implementation goals like: a small compiler, efficient code generation, and direct access to all hardware features. Under these constraints, permitting machine code in a program may be reasonable.

This approach, however, has serious drawbacks. For our purposes, the major problem is that it is impossible to guarantee system integrity through the language. By using the absolute addressing feature. an INLINE

[1]PSW--program status word; this command loads the instruction counter and several associated protection keys.

instruction could modify code generated by the compiler. This action would subvert the semantics of the high-level programming features, causing a break in language integrity. Similarly, an INLINE could modify registers set by the compiler, leading to subtle violations of both data and abstraction integrity. This excessive potential for subversion makes the INLINE approach incompatible with our system requirements.

A more general drawback is that inserting machine code is rarely, if ever, understandable. The programmer must be thoroughly familiar with the characteristics of the hardware and the internal workings of the compiler. Naturally, the INLINE code in a program will reflect many of the problems caused by this interface. As a result, the machine code is difficult to write and very time-consuming to debug. The INLINE approach may be simple from a language designer's point of view, but it is far from simple for an operating system designer to use.

2.1.2. Semaphores

The second technique for process interaction permits direct sharing of objects, but under complete language supervision. In this strategy (and all succeeding ones) the language supports the creation and management of asynchronous processes. Interaction occurs by permitting different processes to access the same variables and procedures. The programmer can synchronize access to these objects through a new variable type, semaphore, and the operations associated with it. ESPOL [49] (the language used for Burroughs system programming) and PL/1 [37] both employ this technique.

A semaphore variable [20,28] can be represented as a non-negative integer which cannot be directly accessed by users. The only legal operations on semaphores are WAIT and SIGNAL. By definition, all invocations of these operations exclude each other in time. One definition for WAIT and SIGNAL is shown below:

```
WAIT: procedure (V: semaphore);
      if V > 0 then V:=V-1;
              else block this process on V;
      end WAIT
SIGNAL: procedure (V: semaphore);
      if some process is blocked on V
              then unblock one of them
              else V:=V+1;
      end SIGNAL
```

ESPOL and PL/1 use variations of this WAIT and SIGNAL, but the differences are not important to this discussion.

An example demonstrates the use of these language primitives to synchronize access to objects. Assume several processes wish to save the value of a variable A and then increment A by one. The following sequence of code can synchronize access to A.

```
WAIT(V);
SAVE:=A;
A:=A+1;
SIGNAL(V);
```

Proper synchronization is assured if three conditions are met: (1) the initial value of V is one; (2) each access to A in each process is bracketed by WAIT(V) and SIGNAL(V); and (3) no other calls of WAIT(V) and SIGNAL(V) are made in the program. Under these conditions, the value of V is either zero or one. If V equals zero, then exactly one process is accessing A. Otherwise, no process is using A.

The semaphore strategy for process interaction is a significant improvement over INLINE; by including the process concept in the language, it becomes possible to insure that the compiler cannot be subverted. Hence, this approach does permit the establishment of language integrity.

The weaknesses of the semaphore system can be identified from the previous example. A programmer is permitted to synchronize access to an object by logically associating it to some semaphore variable. Each process then gets direct access rights to both the object and semaphore. As long as each process satisfies the three conditions previously discussed, data integrity is insured. Unfortunately, languages employing semaphores cannot always enforce these conditions. In fact, in the example, the language has no way of connecting semaphore V with variable A. An untrustworthy user (or a simple coding error) could cause a violation of the conditions and thus a loss of data integrity. Similarly, each user that acquires access to a shared variable defines his own interface to it. Hence, a new process could easily destroy a completely safe interaction that had previously existed. This problem implies that the language cannot enforce abstraction integrity.

In summary, the semaphore approach supplies sufficient tools to permit safe interaction, but it does not require that they be used safely. The potential for misuse of the language is still too great for the integrity we need.

2.1.3. Regions

The critical region approach [7,20,31] to process interaction is based on language-controlled use of all shared objects. Each object that is accessible to more than one process must be declared with a shared attribute:

var A: _shared_ <type name>;

When a process wants to manipulate a shared object it must first enter a region block for that variable. The syntax is:

region A _do_ S

where A is a shared variable and S is any statement (or block of statements) allowable in the language. Under this structure, only the statements in S are permitted to access the shared object A. During program execution the language insures that at most one process at a time is inside a region for a particular shared variable.[2]

This approach gives the language sufficient information to insure data integrity. Any object that is not declared to be "shared" can be easily limited to one process domain. If an object is shared it is also easy to check that each access is within the appropriate region. The only other problem is for the language to insure that only one process at a time is inside a region for a particular shared variable. This is also simple to do using common hardware features.

We can illustrate the use of regions by solving the same problem we did with semaphores. Here, the variable A is declared shared by:

var A: _shared_ _integer_;

Each process wanting to save and update A would take the following actions.

---

[2]This synchronization technique often includes an extra feature, like _await_, for program-controlled scheduling inside a region. This type of addition does not alter our evaluation of the approach.

```
var SAVE: integer;            /* local var */
    :
region A do begin
            SAVE:=A;
            A:=A+1
            end
```

Notice that all references to A are inside the region for A. Outside the region, each process can access its own local variable, SAVE, to determine what value A had--but it cannot directly access A.

The weakness of regions is that each process defines its own interface in an interaction scheme. For example, in the above case each process has its own copy-and-increment code. This approach can be dangerous with untrustworthy processes because they may decrement A or add twenty instead of adding just one. A more difficult problem arises if a process goes into an infinite loop inside the region. It effectively shuts down all future interaction through that shared object. These problems actually reflect the more basic flaw that the interaction itself is not clearly defined. It is necessary to examine every region statement of a particular shared variable to understand one complete point of interaction. These problems indicate that regions cannot support abstraction integrity.

2.1.4. Summary

In the introductory chapter we enumerated three language requirements needed in order to safely design operating systems: language integrity, data integrity, and abstraction integrity. In this section we have reviewed the early approaches to process interaction and found that none of them satisfied all our requirements. The results of the analysis are summarized in Figure 2.1. The underlying problem with all three methods is that too much responsibility is given to language users, and too little accepted by the language itself. As a result, the facilities are too powerful, and make it difficult, if not impossible, for system designers to adequately control their use.

2.2. Advanced Interaction Mechanisms

The advanced mechanisms for process interaction provide a more controlled language environment. This extra control permits the compiler to detect and prevent interaction errors that might be difficult for the programmer to find himself. In this group we include three approaches:

1. message systems [7,11,25],
2. monitors [33], and
3. modules [63,64].

These facilities meet our requirements for system integrity, so we can begin to compare them for their effects on programming structure and style. We use the term "expressiveness" to represent this type of comparison.

Because expressiveness is an extremely subjective criterion, we begin this section by casting the evaluation in a more objective light. Several types of interaction which could arise while designing an operating system are identified. For each type of problem, we consider some specific solutions that expressive mechanisms should allow. If a particular interaction mechanism can only implement them by incurring excessive overhead, then we say that the mechanism is not very expressive.

	Interaction mechanisms		
	INLINE machine coding	Semaphores	Regions
Language integrity	No	Yes	Yes
Data integrity	No	No	Yes
Abstraction integrity	No	No	No

Figure 2.1: Summary of weaknesses in primitive process interaction schemes.

After we describe the types of interaction problems, we examine message systems, monitors, and modules to get a reasonably objective comparison of their expressiveness.

2.2.1. Expressiveness

In the design of an operating system many different examples of process interaction are likely to arise. Rather than select an arbitrary set as a basis for deciding expressiveness, we partition the set of interactions into several classes based on three different criteria:

1. Permanent versus temporary,
2. Single versus multi-level, and
3. Mutual exclusion versus simultaneous access.

These criteria give us a broad spectrum of problems for analyzing interaction mechanisms. Since the classes are relatively independent (e.g., if an interaction is permanent, that does not affect its placement in the other categories), we will treat each one as a separate type of problem.

The first partition of types of interactions is based on whether they are permanent or temporary. Permanent interactions are defined when the operating system begins execution and remain throughout its life. One example of this case is a disk resource. Interaction occurs because users compete for the right to access the disk. The resource should be considered permanent since the disk is permanent. This type of interaction is probably the most common one in operating systems and, as such, its representation should be as clear and efficient as possible.

Temporary interactions, although not as common, have some interesting characteristics with regard to expressiveness. Their lifetime may begin and end any time during the execution of the system. For example, the operating system may want each user to have his own virtual printer. While the user is executing, this printer queues up all output lines. When the user is done, the printer transmits all lines to the real printer and then dies. Thus, the lifetime of the virtual printer is the same as that of the user, not the operating system. The significant difference between permanent and temporary interactions is that in temporary ones the designer must consider termination. In the virtual printer case the output lines go to the real printer just prior to termination. In other cases, memory or disk space may need to be returned, or queued messages may be moved elsewhere. Since termination is an integral part of the interaction, it should be possible to handle it cleanly.

The second partition of interactions is based on design structure. If a particular interaction is defined by using more primitive ones as a foundation, we call it a multi-level interaction. Disk scheduling is one example. The primitive interaction facility is a disk resource which manages use of the disk. The higher-level interaction is a scheduler which implements a particular sequencing of access to the disk. Multi-level interactions are an important class of interactions because methodologies like structured programming and hierarchical design often lead to their use.

In partitioning the interactions in this way, we want to focus our attention on the interface between two levels. In particular, can two different levels operate independently or must they exclude each other in time? In the disk scheduling example it is critical that the scheduler be allowed to receive requests while the disk is operating; so in this case the levels should operate independently. However, in other situations we need the opposite. For example, we may define a message-passing system that is based on an existing buffer allocation scheme. If the message passer requests a buffer (i.e., control goes to the inner level) we do not want another process to begin passing a different message. Use of the buffer system should exclude further use of the message system. Since both of these forms of multi-level interaction represent useful operating system concepts, our expressiveness criteria demands that both be clearly programmable.

The third partition of interactions is based on the amount of simultaneous interaction that is possible. A particular interaction can fall into one of three groups:

1. processes must have mutually exclusive access,
2. processes may overlap access for read-only purposes, or
3. processes may overlap reading and writing access.

An example of an interaction in the first group is message-passing. If one process is sending a message, no other process can simultaneously send or receive a message over the same channel. Any such attempt by a process should be delayed until the previous action is completed. Most interactions fall into this group.

The second group includes interactions where simultaneous access is allowed, but only for reading. The classic example here is the readers/writers problem [15]. Several processes share read and write access to a data file. Each write operation must have exclusive access to the data in order to change it, but any number of reads may proceed simultaneously.

The third group of interactions permits a more generalized access overlap; namely, some reading and writing can execute simultaneously.

Linked-list manipulation (with operations SEARCH, INSERT, and REMOVE) is one example in this category. If coded properly, any number of SEARCHes (reading only) and one INSERT (which involves writing) can execute simultaneously on the same list without any possibility of incorrect results. Another example in this group is the on-the-fly garbage collector problem [22,27].

We want to be able to program solutions to problems falling in all three of these groups (without losing data integrity). The difficulty is that some correct programs, like INSERT for list-manipulation, come very close to violating data integrity. If links are altered in the wrong order in INSERT (even if the sequential result is unchanged), a violation is possible. This means that our goal here may be unattainable.

In summary, our criteria for expressiveness is based on the ability to implement a broad range of interaction problems. Figure 2.2 outlines the different types we have discussed, and identifies one sample problem from each. Our assessment of expressiveness is still somewhat subjective, however. Each of the mechanisms we analyze will be able to implement solutions to most of the problems in one way or another. Our concern is *how* it will be implemented. If implementation requires extra levels of procedure calls, extensive run-time checking, or similar inefficiencies we feel that this is evidence of poor expressive power. Since these things affect execution as well as human understanding, we must attempt to minimize them.

2.2.2. *Message Systems*

The first technique that satisfies all of our integrity requirements for process interaction is message passing. It does so by taking a very

Partition Basis	Groups		Example
Interaction Lifetime	Permanent		Disk resource
	Temporary		Virtual printer
Interaction Structure	One-level		Shared data files
	Two level	Mutual exclusion	Message system with buffers
		Simultaneous execution	Disk scheduling
Internal Overlap	No access overlap		Message passing
	Overlap reads		Readers/writers
	General access overlap		List manipulation

Figure 2.2: Summary of interaction types for evaluating expressiveness.

controlled view of the entire system. This approach prohibits all direct sharing of objects. Each process is completely isolated in terms of variables, procedures, etc., that it may access. Interaction occurs only through the transmission of information (messages) between processes, and it is completely controlled by language-defined mechanisms. This stragegy has not actually been built into any major languages, but it has been discussed in various sources [7,11,25] as a descriptive mechanism. Several operating systems such as THE [21], RC 4000 [4], MULTICS [14,48], HYDRA [65], and SUE [13] have been designed to support this type of interaction.

Message passing is a straightforward concept to define and implement. A process transmits a message by invoking the language-defined operation SEND, specifying the message and identity of the receiver as parameters. SEND records this information and then returns. A process acquires a message by calling RECEIVE, which returns a copy of a message and the identity of the sender. If RECEIVE is called and no messages exist for the caller, that process is delayed until a message can be returned. Many variations of this basic scheme exist. The sender may specify a set of processes which may receive the message. Or different algorithms may be used for selecting which message, of several pending, is to be given next to the receiver. These variations do not affect the basic strategy.

With this approach, the structure of an operating system is relatively fixed. Each process has its own permanent access rights. If it needs to perform any action not included in those rights, it must send a message to a process which can perform it. Presumably, when the action is completed the process doing it would send a return message. Hence, this approach favors designs which have a one-to-one correspondence between processes and potential actions. The minimum cost of any particular request is one SEND and one RECEIVE. If a response is necessary, the cost doubles to four message passing operations.

In analyzing message passing for expressiveness, the biggest problem is inflexibility. The language must define the message system that will be used by every process, even though the needs of various subsystems (and users) could vary widely. The algorithm for queuing unreceived messages and the mechanism for identifying potential receivers are only two of the policy decisions that must be made by the language. Once they are made, designers will be forced to work around unfavorable decisions. If the language permits messages to be sent to any process, security-minded processes will need to verify each sender before using a received message. Each type of interaction has its own needs for transmitting information, and they are all different. One message system cannot satisfy them all without significant overhead.

The other problem with message passing is its inefficiency. It requires four procedure calls in order to have a user process read one card from an input process because he must send a request and then receive a result. (The I/O process has complementary calls.) This is definitely expensive and unnecessary. The fact that the approach forces the designer into this organization for all interactions makes the cost that much greater.

In summary, message passing as a language feature is only effective if the designer wants exactly that. For other types of interaction, this system creates too much overhead and forces too many decisions to be a practical tool.

2.2.3. Monitors

The second advanced technique for process interaction is the monitor concept. Introduced by Brinch Hansen [8] and later developed by Hoare [33], it has been realized as an actual programming tool in Concurrent Pascal [9]. The underlying philosophy is that process interaction is a separate entity from any of the processes that use it. The methods of interaction should be defined solely by that entity, and not by the processes which access it. As a result, this approach significantly improves on message passing as an expressive tool for defining interaction.

The monitor concept is a derivative of the CLASS construct of SIMULA67 [16]. A CLASS can be used to encapsulate abstract data objects--i.e., define operations on objects while hiding implementation details from user blocks. The same approach applies to monitors. They define interaction operations available to processes without disclosing their internal details.

A monitor is a collection of data and procedures with the structure shown in Figure 2.3. In this system, the procedures are permitted to access the local monitor data. The key to this approach is that processes declared outside of a monitor may still invoke the monitor's procedures. They may not, however, directly access the monitor's data. This is a significant change from the scope rules of most languages. To insure data integrity (several processes may simultaneously call monitor procedures

```
MONITOR_NAME: monitor
      begin
          local monitor data;

          procedure proc_1(     );
              end;
          :
          procedure proc_n(     );
              end;

          initialization code

      end MONITOR_NAME
```

Figure 2.3: Syntax for the definition of a monitor.

which modify local data), a monitor only permits one of its procedures to be executing at a time.

Interaction occurs in this system by permitting several processes to access a particular monitor. Each process can invoke a monitor procedure using the statement:

MONITOR_NAME.proc_i(params)

The procedure can then access the local monitor data, which retain their value between calls. The effect of this approach is that the local monitor data are variables shared by processes. However, they are shared in a limited way since the monitor procedures define the only access paths. The interaction using shared variables insures data integrity in this case because the monitor allows at most one process to use the variables at any time.

We demonstrate the use of monitors with a simple example. Figure 2.4 shows a skeletal solution for the bounded buffer problem [7,33]. The monitor acts as a communication center which allows processes to add information (APPEND) or remove it (REMOVE). The procedures share three varibles--BUFFER, FIRST_OUT, and COUNT, which define a circular buffer. The last two lines of the monitor initialize the buffer prior to any access by processes. This code is executed once, <u>automatically</u>, when the monitor is defined.

After the monitor is defined and initialized, processes can use it to interact. One common use has one process putting information in the buffer (through APPEND) and another removing it. The logical integrity of the circular buffer is insured by the monitor procedures. Processes using the monitor cannot change its definition. If the interaction fails

```
BOUNDED_BUFFER: monitor
      begin
           BUFFER: array 0..N-1 of PORTION;
           FIRST_OUT: 0..N-1;
           COUNT: 0..N;
           :
           procedure APPEND(X:PORTION);

                   end APPEND;

           procedure REMOVE(RESULT:PORTION);

                   end REMOVE;

           COUNT := 0;
           FIRST_OUT := 0

      end BOUNDED_BUFFER
```

Figure 2.4: Skeleton of monitor solution to the bounded buffer problem.

to work properly, it is a result of code inside the monitor and nowhere else.

The last major feature of monitors is the option for program-controlled synchronization. Situations often arise when a monitor procedure cannot immediately complete its task. In the bounded-buffer example what happens if REMOVE is called and no information is in the buffer? One alternative is to return indicating failure. This choice would probably force users into busy waiting (repeated calls to REMOVE until it returns successfully). Another alternative is for the monitor procedure to delay the process until another process calls APPEND. In order to permit this type of program-controlled synchronization, Hoare defines condition variables and operations <u>wait</u> and <u>signal</u> [33].[3] We declare a condition variable by:

<u>var</u> NON_EMPTY: <u>condition</u>

as part of the monitor data. Then in the REMOVE procedure, if the buffer is empty we execute:

NON_EMPTY. <u>wait</u>

The code after the <u>wait</u> assumes that the buffer now contains data. In order to actually restart a process waiting for information, the APPEND procedure must issue:

NON_EMPTY. <u>signal</u>

every time it is called. The complete code for the bounded buffer problem is given in Figure 2.5.

---

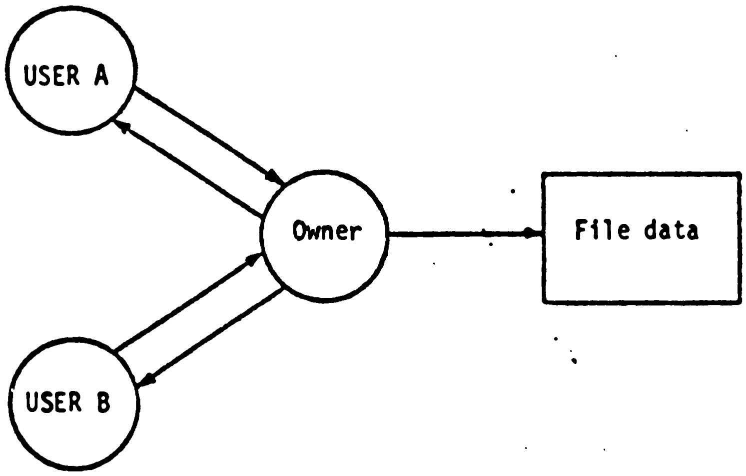
[3]These operations should not be confused with semaphore operations which, although similar, have different interpretations.

```
BOUNOED_BUFFER: monitor
            begin
               BUFFER: array 0..N-1 of PORTION;
               FIRST_OUT: 0..N-1;
               COUNT: 0..N;
               NON_EMPTY,NON_FULL: condition;

               /* Invariant: elements of BUFFER are in the          */
               /* order in which they were APPENDed, in locations   */
               /* BUFFER(FIRST_OUT),BUFFER(FIRST_OUT + 1 mod N),..., */
               /* BUFFER(FIRST_OUT + COUNT - 1 mod N).              */

               procedure APPENO(X:PORTION);
                      begin
                      if COUNT = N then NON_FULL. wait;
                      BUFFER(FIRST_OUT + COUNT mod N) := X;
                      COUNT := COUNT + 1;
                      NON_EMPTY. signal
                      end;

               procedure REMOVE(X:PORTION);
                      begin
                      if COUNT = 0 then NON_EMPTY. wait;
                      X := BUFFER(FIRST_OUT);
                      COUNT := COUNT - 1;
                      FIRST_OUT := FIRST_OUT + 1 mod N;
                      NON_FULL. signal
                      end;

               FIRST_OUT := 0;
               COUNT := 0
               end BOUNOEO_BUFFER
```

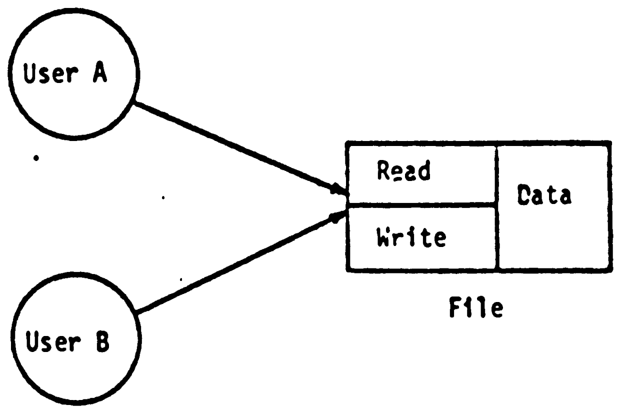
Figure 2.5: Complete monitor solution for the bounded buffer problem.

The condition system is implicitly tied to the monitor concept. When a process executes wait (which must be done inside some monitor), that process becomes blocked. The difficulty is that the process is still logically inside the monitor, and another process must be allowed to enter in order to unblock it. This problem is resolved by having the wait also relinquish control of the monitor so another can get it. Signals create similar control problems. If a process issues a signal and no process is waiting, the signal has no effect and the process continues. If, however, another process is waiting, the signaller is delayed and the waiter is unblocked and permitted to continue. This action guarantees that the waiting process gets immediate control of the monitor (so it can assume the condition is true). For a more complete treatment of monitors and the condition system see [33].

The monitor concept provides some significant improvements in expressiveness over the message system. If a process wants to take some action like reading an input card, it can call a monitor procedure which has direct rights to the device. The process itself performs the read. This is one procedure call as opposed to four calls using messages. Another good example is a shared data file. With messages the file must be kept by a separate process. Users then access the file by sending messages to that owner. This structure is shown in Figure 2.6.A. Alternately, a file could be implemented as a monitor. Then each user calls monitor procedures to access the files themselves. This structure is shown in Figure 2.6.B. This design simplifies the solution by eliminating an unnecessary process.



A) Using message systems



B) Using monitors

Figure 2.6: Two designs for a shared file.

Monitors also permit clear representation of some multi-level types of interaction, since they are permitted to call other monitor procedures. When this happens, the process making the calls has exclusive control over two monitors. For example, if process A calls M1.OP1 which then calls M2.OP2, then no other process may execute procedures in M1 or M2 until the calls from A are completed.[4] This rule permits monitors to easily implement a message system based on an established buffer scheme. When the message system (a monitor) needs a buffer, it can call the allocator within the buffer monitor. During this call no other process can enter the message system because it is still in use.

Monitors do, however, have some weaknesses. They are intended mostly for representing permanent interactions, so temporary ones like the virtual printer problem create difficulties. For example, insuring that no information is sent to the virtual printer after it has sent data to the real printer requires an expensive run-time check. Similar problems arise if the temporary interaction acquires and later releases objects like buffers. We need to insure that the objects are released only when the interaction is finished. Monitors make this difficult.

Another problem with monitors is in nested systems where the levels must execute simultaneously. Since calls out of monitors must retain control of the first level, interactions like disk scheduling must be structured in an unusual way. Figure 2.7 outlines this structure. The scheduler's operations are REQUEST_DISK and RELEASE_DISK. The first one returns when the calling process may access the disk. The second

---

[4] Actually, a call may be attempted, but it will be delayed until the other call completes.

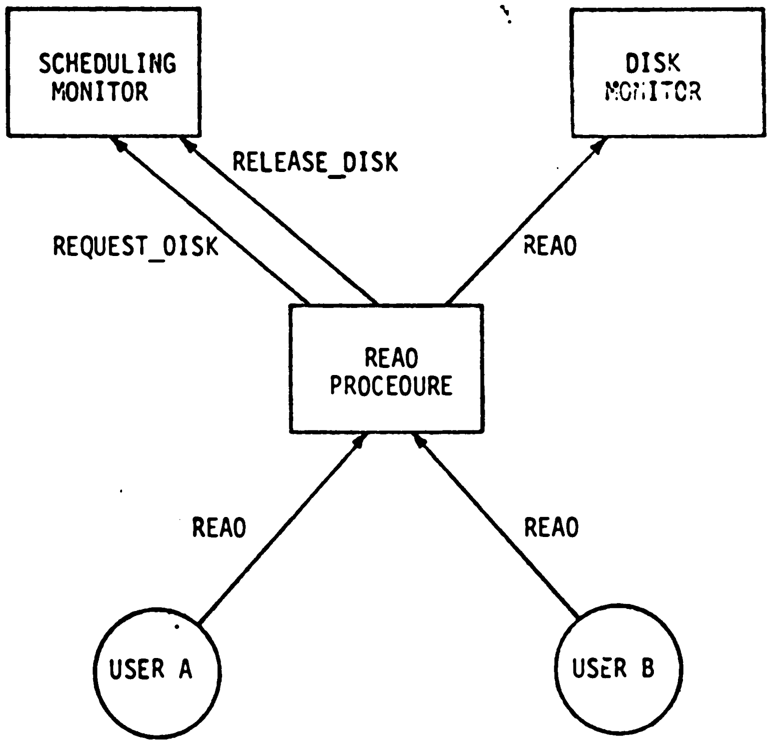


Figure 2.7: Disk scheduling with monitors.

procedure notifies the scheduler that a process has completed. To enforce proper use, an extra procedure must be inserted between users and the disk. This procedure enforces the policy of scheduling before use. The extra procedure overhead and the unusual structuring make this solution weak.

2.2.4. Modules

The last interaction mechanism we consider is the interface module developed in the language MODULA [63,64]. This approach is actually very similar to the monitor concept. We treat them separately in order to focus on the design differences and their effects on subsequent uses in interactions. According to Wirth [64], the purpose of an interaction module is to provide an adequate facility for defining large, complex entities like a scanner for a compiler or a disk store manager. He assumes that for these types of tasks only one instance of each is likely to be needed, and also that they are self-contained rather than built up through layers of hierarchical design. These differences in philosophy lead to Wirth's version of monitors.

The interface module has the same basic structure as a monitor--data blocks and procedures are grouped in a new type of block. Figure 2.8 illustrates the syntax. The access rules are roughly the same as they were for monitors; the difference is that outside the module access is not necessarily limited to the procedure names. The define-list option permits a programmer to specify exactly which objects are to be known outside the module--they may include procedures, data variables, and types declared internally. Any object not listed, is not known outside. If a variable is listed, by definition it is only visible for read purposes.

```
NAME: interface module;

      define ITEM_1,...,ITEM_k;

      module data and type definitions;

      procedure proc_1(    );

             end;
      :
      procedure proc_m(    );

             end;

      initialization code

      end NAME
```

Figure 2.8: Syntax for the definition of a Module.

Likewise if a type is listed, the name can be used to create instances, but the internal structure of that instance is invisible to the creator. However, the main use for define-list is to export procedure names to provide operations for various processes.

With interface modules the call of a procedure is slightly different than monitors. An invocation of an interface module procedure does not include the module's name. So for example, where a monitor call might have been: "MESSAGE.SEND( );", the appropriate form for an interface module would be "SEND( );". The immediate consequence of this approach is that all module procedure names must be unique (which was not necessary with monitors).

The internal synchronization of interface modules is also similar to monitors. Only one procedure at a time may be in execution within a particular interface module. This constraint is enforced by the language. Program-controlled synchronization is permitted through condition variables and operations WAIT and SEND, again similar to monitors with slight variations. In monitors when a signal is issued, the signaller is suspended to permit the waiting process to continue immediately. When that waiting process leaves the monitor, the signaller is reactivated to continue. In modules, the signaller may not be the first process to use the interface module after the waiter leaves.

The most significant new concept in interface modules is the treatment of real devices. Although monitors can be used to represent them, the one language using monitors, Concurrent Pascal, did not make that choice. In Modula a real device is represented by a special type of interface module, called a device module. Inside a device module it is possible to issue commands for initiating I/O to a particular external device. Details on the mechanisms for this system are complicated and unnecessary for our purposes, so they will not be discussed here. The interested reader can find the information in [63,64]. Device modules are important because they permit a unified approach for handling both real and virtual interactions. The syntax and conceptual framework is the same for all users whether the interaction is a "virtual" message-passing system, or a "real" disk device.

Most of the variations between monitors and modules affect the ways in which they are used only slightly. For the most part, if one system can be used to solve a problem then the other can also, in nearly the same way. For example, monitors may be used in a TYPE statement to create several identical instances. This option is not legal with modules. In order to get the same effect with modules, each instance must be defined separately. While this may be an inconvenience, it does not represent a significant difference in programming power.

The only major difference between monitors and modules is in nesting situations. According to Wirth, interface modules are not permitted to call other interface modules. This decision is in keeping with his philosophy that most interactions are self-contained and need not rely on other, more primitive, interactions. As a result, when nesting is the reasonable approach for organization, the design structure must be modified. The message system which uses a buffer allocation scheme is a good example. Normally, the message system could request a new buffer by calling out directly. See Figure 2.9A for that structure. With modules, an extra procedure must be added which first gets the buffer and then

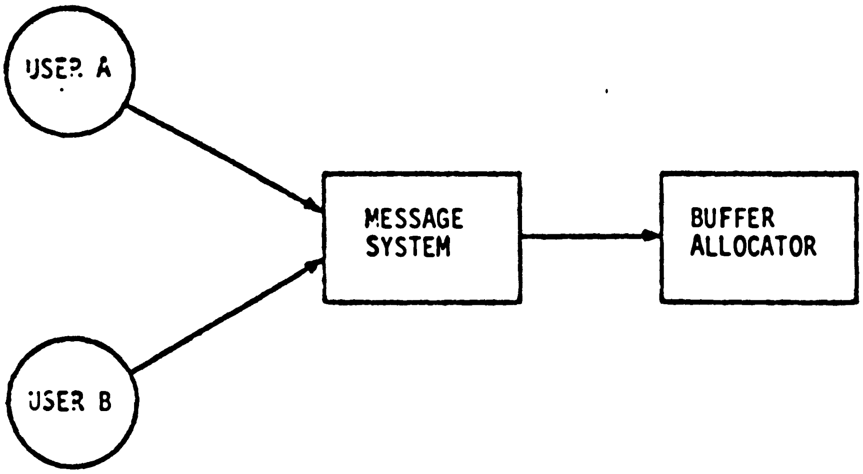


Figure 2.9A: Design structure for a message system.

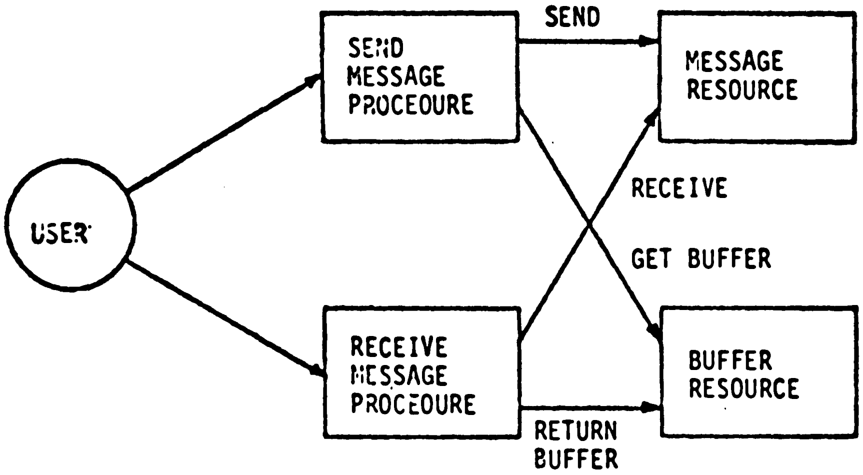


Figure 2.9B: Message system design with Modules.

invokes the message system. Figure 2.9B sketches this approach. Clumsy organizations such as this one, which are dictated by the language facilities, demonstrate the inherent weakness of forced monolithic designs.

2.2.5. Interaction Summary

We summarize the results of the previous discussion in Figure 2.10. A particular interaction mechanism is considered "good" for a particular type of problem if the solution using that mechanism is clean. If the problem can be solved, but only with extra procedure calls, extensive process switching, or special run-time checking then the mechanism is rated as "poor." If it cannot be done at all--then it is listed as impossible.

Based on this evaluation, monitors have the strongest rating but it is clear that there is room for improvement. Conceptually, monitors are strong because they allow processes to do more operations for themselves (through procedure calls) rather than incurring the overhead of passing messages and process switching. This can be done in safety because a process cannot violate the integrity of a monitor. The weaknesses in monitors are caused by several restrictions which prohibit some useful applications. However, we will show that these limitations are not inherent in the monitor concept and most of them can be fixed.

2.3. Access Control Systems

The second area of language design with which we are concerned is mechanisms for access control. In process interaction, particularly in a monitor-like environment, the ability to control access rights can have a strong effect on efficiency. For example, if a monitor procedure is

Partition groups	example	message passing	monitors	modules
Lifetime				
permanent	message passing	Good	Good	Good
	disk resource	Poor	Good	Good
temporary	virtual printer	Poor	Poor	Poor
Interaction structure				
one-level	shared data file	Poor	Good	Good
multi-level mutual exclusion	message system with buffers	Poor	Good	Poor
multi-level simultaneous execution	disk scheduling	Poor	Poor	Poor
Internal overlap				
no overlap	message passing	Good	Good	Good
overlap reads	readers/ writers	*	*	*
general access overlap	list manipulation	*	*	*

* impossible

Figure 2.10: Expressiveness of advanced interaction mechanisms.

invoked by a process, can it be assumed that the caller is a valid user? If a monitor must protect itself from illegal users then the interaction system will be slowed down significantly because of additional run-time checks. In this section we address the problem of access controls in the same way that we dealt with interaction. We first examine types of access control problems which often arise in operating systems. This establishes our expressiveness criteria. Then we discuss various existing access control methods and evaluate their relative merits.

2.3.1. Expressiveness

Access control problems are most commonly divided into two basic categories--static and dynamic. In the static case the subject (normally a process, monitor, or procedure block) has permanent access to some object (which could be a variable, procedure, monitor, type name, etc.). In the dynamic case the subject's rights for a particular object may vary while the program is executing. The one general constraint which we feel is important is that each static access right in a program should be enforced solely through the compiler. In other words, it should not require any run-time checking. For the dynamic case run-time checks cannot be avoided, so the goal here is to keep the cost of it as small as possible. The other important expressiveness criteria are related to the types of problems found in each category.

Within the static case, four different types of access control appear to be useful: no access, full access, partial access, and use without create access. The most common is to prohibit a subject from ever directly accessing some object. For example, in the disk scheduling problem of the previous section a necessary part of any solution is .

insuring that users cannot bypass the scheduler and invoke the disk directly. In the system environment we are considering (users and operating system in the same program) this type of control is probably the most critical.

Almost as important is the second type of control which permits a subject to have full access to an object. For example, a file system is one type of object that would probably be accessible to all users. At a glance, these first two problems may seem like two ways of saying the same thing. We separate them because mechanisms which may help in one, could actually hamper solutions in the other case.

The third type of static access control has to do with selective rights to an object. Some subjects should not have the right to invoke all operations on a particular monitor. For example, the operating system may use a message-passing monitor to transmit information to users. Hence, users must be able to receive messages along that channel. However, they should not be able to send messages because they could effectively clog the channel with garbage information. The solution we seek is independent control of access to each monitor operation.

The fourth type of useful static access control involves types and type instances. For example, assume a monitor is defined in a type definition for managing control of a remote terminal device. Since several terminals exist in the environment, the operating system could define one instance to control each of them. A user could then be given the ability to use one of those monitor instances. The problem is to insure that a user cannot create any new instances, thereby establishing an I/O link unknown to the operating system. In general, we want controls which allow a user to access an object while prohibiting him from defining new instances on his own.

The types of useful dynamic access controls are more numerous. First, all of the controls mentioned for the static case apply equally well to dynamic objects. Full access, no access, selective entry access, and use without create are all relevant dynamic control features. The additional problems associated with the dynamic case are related to how the rights move between different environments and how they are managed.

In considering rights movement, we are concerned with freedom to grant access and ability to later revoke that access. If a user holds access to some file (presumably a dynamic object) it should be possible for that user to grant access to other processes. The point here is flexibility. The operating system may not want such movement to occur and indeed it may prevent it, but the language facilities should not make that decision. Free movement of access rights should be supported. The other half of rights movement is revocation. A file owner should be able to get back all rights given out earlier. Or, more accurately, he should be able to prevent use of his file by others. Again, many policy questions are associated with this constraint--we only want the language to permit it in some form.

The last area of concern we have in the dynamic case is rights management. What types of objects should control movement of access rights? Two candidates seem reasonable: processes and monitors (or whatever interaction mechanism might be in the language). Monitors are appropriate for situations like handling buffer pools, where the rights are kept by the monitor until users request an instance. In fact the purpose of this type of interaction is strictly to manage access rights. Processes are appropriate for managing rights when special actions are necessary--like garbage collection. Monitors are only set into action when invoked by some process. If continuous garbage collection is needed (which involves changing access rights) then a process is the appropriate management tool. The point is that both processes and monitors could reasonably be used for managing rights, and that the language facilities for providing dynamic access should permit either form of control.

We summarize our criteria for expressiveness in access control facilities by again using a chart (Figure 2.11). Our goal is to be able to represent a broad range of policies reflecting the types of problems we consider important. If a particular access mechanism requires special overhead to enforce a certain form of protection, it is a good indicator of a weakness of that approach. We now use these criteria to evaluate existing access control systems.

2.3.2. Block-structured Languages

The first type of access control system we consider is that normally used in sequential, block-structured languages. In this approach, scope rules are the basis for controlling the static access rights of each program block. These rules are generally inflexible and promote the policy that the innermost block in a nested structure is the most powerful. Dynamic access rights are manipulated through parameter passing. When a procedure is invoked, it temporarily has the right to access objects passed in as parameters. An obvious question is--what must be change in this control system in making the transition to a multiprogramming language? One possible answer, which was taken by the PL/I designers, is to make no changes at all. This choice turns out to be a poor one.

Class	type	example
Static access rights	no access	prevent direct access to real disk
	full access	file system
	selective-entry	receive-only messages
	use w/o create ability	remote terminals
Dynamic access rights	no access	O.S. files
	full access	file system
	selective-entry	receive-only messages
	use w/o create ability	remote terminals
	general rights sharing	file system
	revoke rights	ownership situations
	monitors as managers	buffer pool
	processes as managers	garbage collector process

Figure 2.11: Expressiveness criteria for access controls.

The scope rules for PL/I are a major source of problems when the language is used for multiprogramming. By definition, each block in a PL/I program may access any objects that are defined in its heading, and also any objects defined in blocks which physically contain the block in question. This rule applies equally to all types of blocks (begin, procedure, etc.).[5] As we mentioned earlier (in Section 2.1) this rule prevents the language from insuring data integrity. Each process has the ability to freely access every globally-defined object.[6] As a result, all the desired restrictions on static control are difficult to achieve. The only way to block some piece of code from seeing a global object is to declare a dummy object with the same name. Needless to say, we consider this solution to be very poor.

In the area of dynamic rights movement, it becomes almost impossible to accurately evaluate PL/I. In general its mechanisms are very primitive, in fact the only mechanisms for dynamic rights movement are pointer and procedure type variables. In the context of the problems we are considering, these facilities are inadequate without making extensive assumptions about their use.

In summary, PL/I is mainly based on static access controls, particularly when multiprogramming is used. The major problem with PL/I's system is that it permits too much access by each block, with no real help for program control of access rights. It is this problem that later languages attempted to fix.

---

[5]PL/I has no process blocks for identifying concurrent program code. Concurrency is achieved by invoking a procedure as a task. As a result, it is difficult to identify which sections of code will ever execute in parallel.

[6]Assuming no name redefinition occurs through nesting.

2.3.3. Concurrent Pascal

The access control system of Concurrent Pascal provides a much more constrained and secure environment than PL/I, mainly because Concurrent Pascal requires explicit identification of all processes via process-type blocks. These blocks provide a maximum scope of execution for each process (i.e., it is illegal to branch out of a process block). Also, Concurrent Pascal deals strictly with a fixed type of environment. All processes and objects are considered to be permanent. Moreover, all access rights are static in nature. These limitations and the actual access mechanisms permit the language to enforce data integrity, while giving the programmer some control over access policy.

In Concurrent Pascal each process block may only access two types of objects--locally defined ones and those passed as parameters during initialization. Notice that this scope rule is much more restrictive than in PL/I. Furthermore, only monitors and constants may be passed during process initialization. This effectively isolates each process in a small environment. It is impossible for two processes to directly share any global variables or procedures. In fact, interaction can only occur between two processes if they are both passed a common monitor.

This control system yields a very simple programming environment. At the outermost program block, all processes and monitors must be defined. The only direct accesses permitted between these blocks are calls from one monitor to another. Processes can acquire access to a monitor only if it is passed as a parameter. Figure 2.12 gives a sample program to illustrate these points. None of the processes can use the names A, B1, or B2 to access the monitors. Notice, however, that the B monitor-

```
MAIN:    A: [monitor
            [
   type B = [monitor
            [
            [ A.PROC1(   );
            [
   var B1,B2 : B;

         X: [process (M1,M2 : B);
            [ M1.ENTRY1(   );
            [
         Y: [process (N1 : B);
            [ N1.ENTRY1(   );
            [
         Z: [process (P1 : B);
            [ P1.ENTRY1(   );
            [
            init X(B1,B2), Y(B1), Z(B2);
```

Figure 2.12: A Sample Concurrent Pascal program structure.

type can directly call the monitor A. The processes do get access to the B monitors through parameters passed at initialization. The actual access policy enforced by this program is sketched in Figure 2.13. The name attached to each arc is the name by which the monitor is known in each environment.

The two major strengths of this system are that data integrity is insured and that a program designer can easily limit the access rights of each process. The rights limitation is significant because it is so simple to do. In the sample program all processes were prohibited from accessing monitor A by default--no actions at all were needed.

On the other hand, there are some problems and weaknesses. First, dynamic access rights do not exist; all objects are permanent with access rights defined statically. Moreover, even some useful static policies are not available without run-time checks--like selective entry and use without create options. The former is impossible because an entire monitor must be passed; the latter because the process must be able to have the type name in order to properly format the formal parameter list.

The other weakness we see is the mechanism for allowing static access to a monitor. Parameter passing, which is traditionally used for dynamic rights movement, is being used to enforce static rights. There is no dynamic movement of rights in Concurrent Pascal, so why make it seem possible?

2.3.4. Modula

The first two access control systems we have examined define the extremes for protection; PL/I is too lenient and Concurrent Pascal is too restrictive. A third approach, taken by Modula, falls between these two

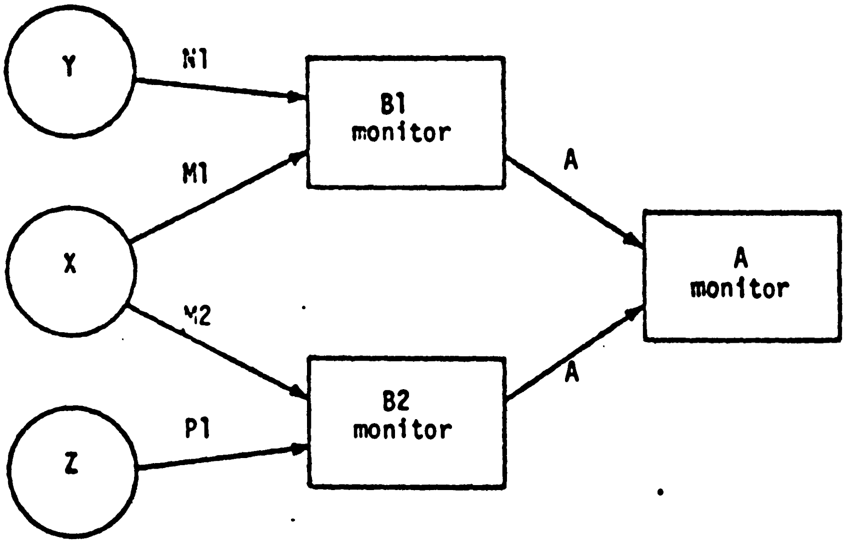


Figure 2.13: Enforcing some access policy in Concurrent Pascal.

and thus has advantages of both. Modula's program structure is comparable to that of Concurrent Pascal--permanent processes and modules (similar to monitors) with only static access rights. The scope rules, however, are closer to that of PL/I--a process may access objects defined outside of its execution scope. The key improvement is that a block may specify those objects which are to be visible inside of it. This extension permits a programmer to specify access policy to be enforced by the compiler.

The scope rules in Modula are intimately connected to a new mechanism called a USE-list. Each process and module block must specify in its heading (through a USE statement) all objects it will access that are defined outside its scope. Hence a block can access only objects that it declares and objects that it explicitly imports. Note, however, that data integrity is not insured by the language since two processes could import the same global variable.

We can illustrate the Modula approach to access control by solving a problem similar to the one we did in Concurrent Pascal. Since Modula does not permit nested calls to modules, we eliminate the monitor A from the policy sketched in Figure 2.13. The resulting policy can be implemented in Modula in two ways--both are illustrated in Figure 2.14.[7] Notice first that access to a module is based solely on knowledge of the operation names (e.g., OP_1, OP_2), not the module name (e.g., B1, B2, C). Each process identifies which objects it will reference within its scope.

---

[7]The two options arise since Modula does not permit a module to be typed. Part A defines each instance separately. Part B uses the type-exporting option to avoid the redefinition. The type B holds all data variables needed by a module instance. Two sets of data are then defined by B1 and B2. For more details on this approach, see [63,64].

Part A.

```
MAIN
  B1: interface module;
        define OP1,OP2;

  B2: interface module;
        define OP3,OP4;

  X:  process;
        use OP1,OP2,OP3,OP4;
          :
          OP1(    );

  Y:  process;
        use OP1,OP2;
          :
        OP2(    );

  Z:  process;
        use OP3,OP4;
          :
        OP3(    );
```

Part B.

```
MAIN
  C:  interface module;
        define OP1,OP2,B;
          :
          type B =...

  B1,B2 : B;

  X:  process;
        use OP1,OP2,B1,B2;

        OP1(B1,...);

  Y:  process
        use OP1,OP2,B1;

        OP2(B1,...);

  Z:  process;
        use OP1,OP2,B2;

        OP1(B2,...);
```

Figure 2.14: Two sample Modula programs illustrating USE-list.

The compiler easily checks that a process does not overstep its rights as specified. As a result, Modula can perform all access rights checking (except, of course, for variables passed as parameter) at compile time with no run-time overhead.

The main weakness of the USE-list option is that an inner block still dictates which objects should be available to it. So in the environment we are considering, we must somehow insure that a user process does not overstep his deserved rights. One possibility is for the operating system to define a user's process block for him, thus allowing it to specify a user's USE-list. This approach is difficult, and not generally useful for enforcing system-wide access policies. The alternative is to encorporate extra dummy blocks to handle the problem. This choice is sketched in Figure 2.15. The extra block limits the objects that the process Y can request. Using this approach, the outer blocks of a program can dictate which objects an inner block should see. Unfortunately, this inelegant solution doubles the number of USE-lists needed.

A second weakness involves selective entry access, as can be illustrated by part B of Figure 2.14. What happens of we want X to be able to perform OP_1 on B1 and OP_2 on B2? Under the design structure of part B (which is the approach recommended by Wirth [64]), we must give X all four objects. Hence we cannot prevent X from performing OP_2 on B1. Notice that under the strategy of part A we can get the necessary protection by giving X OP_1 and OP_4. (However, to do it we have to separately define two identical modules.)

The one remaining static policy to consider is use privileges without the ability to create. Because Modula does not permit modules to be

⋮

DUMMY: **begin**

  <u>use</u> $OP_1, OP_2, B1$;

  Y: process;

    <u>use</u> $DP_1, OP_2, B1$;

<u>end</u>;

⋮

Figure 2.15: Using a dummy block to enforce access policy.

TYPEd, this evaluation is somewhat difficult. However, both schemes for circumventing this rule seem to permit this type of access policy. We can see this by again referring to Figure 2.14. In part A, no type B is ever defined so no process can possibly create an object identical to B1 or B2. Indeed, as far as the language is concerned, B1 and B2 are not even the same. In part B, the type B is not known in any processes so again no new instances can be created. (They could if B were listed in USE.)

In summary, Modula provides more flexibility for specifying static access policies, without incurring any run-time overhead. The USE-list mechanism has a weakness in that inner blocks still dictate access policy, but it can be avoided at the expense of redundant coding. Except for the lack of data integrity (which was an explicit design decision, not something forced by other facets of Modula), the access control system is a definite improvement over Concurrent Pascal.

2.3.5. Managers

The major area of access control that has not been incorporated in either Concurrent Pascal or Modula is dynamic rights movement. Recent research by Silberschatz, Kieburtz, and Bernstein [57] has led to a proposal for permitting dynamic rights in Concurrent Pascal. Their approach introduces a new language facility, called a MANAGER, which controls all movement of dynamic rights. A manager is actually a monitor with special privileges for controlling access to some particular type of regular monitor. As a result, although all objects are still permanent, a process's access rights for a particular object may change during execution.

The manager mechanism is a rather complex approach for achieving dynamic control. Hence, we will only give a general overview of it,.

concentrating on the characteristics which are relevant to our analysis. For a thorough treatment of the system see [57]. A sample program structure using the manager concept is shown in Figure 2.16. In the definition of BOSS, an explicit connection is made to the monitor type BUF. This connection gives BOSS the exclusive rights to define and allocate all instances of that type. In order for a process to acquire access to an instance, it must first declare its intention to do so. Here, process X declares MY_BUF for this purpose. That name has a dual role. At first it provides an access to the manager, BOSS (see the invocation ALLOCATE). BOSS has the power to redirect that name to a monitor instance. Hence, after calling ALLOCATE, X can use MY_BUF to access an instance of the monitor. A manager also has the power to revoke a right, if the user uses MY_BUF to reenter the manager. (For example, X may want to return his right to the manager.)

This approach to dynamic allocation is reasonable for many types of situations. The definition of a manager for a monitor type precludes any other block from manipulating access rights. Since this authority is solely with a manager, access policies such as full vs. no access are simple to enforce. Moreover, use privileges without create is automatically applied to all processes. If the desired access policy falls into these categories then the manager concept can reasonably solve the problem.

Unfortunately, several useful policies are difficult, if not impossible, to implement with managers. When a manager gives out rights for a monitor, it must give full rights. Selective entry access control is not possible without run-time checks inside the monitor. Also, when a

```
MAIN:
   type BUF = monitor;
                 Read: procedure;
                    :

         BOSS: manager for BUF;
                    :
                 Allocate: procedure;

                 Release:  procedure;

            X: process;
               var MY_BUF : BUF from BOSS;

                      MY_BUF.ALLOCATE;

                      MY_BUF.READ;

                      MY_BUF.RELEASE;
```

Figure 2.16: Manager scheme for dynamic allocation.

process receives rights for a monitor the rights are for use only. A process cannot give his rights to anyone else. For example, if the monitor implements a file supposedly "owned" by a process, only that process can use the file. If he wants to share the file with someone else, it can only be accomplished by making a request to the manager. Finally, this approach prevents direct implementation of designs where processes control access rights. Only managers can manipulate rights. These restrictions tend to limit the useful applications of managers.

In summary, the manager concept permits dynamic access control to monitors but with a significant amount of policy attached. If a particular system fits that policy then managers are reasonable. However, some policies are difficult to handle. It seems that the concept is too high-level to fit our needs for general expressiveness.

2.3.6. Access Control Summary

We summarize the results of our discussion on access control mechanisms in Figure 2.17. A language mechanism is rated poor for a particular problem type if the solution requires some form of run-time overhead which is not inherent in the problem. (Notice that all dynamic accessing requires some run-time checking. This mandatory checking does not affect an evaluation.) The mechanism is rated good if its solution to a problem avoids unnecessary overhead. The two "fair" ratings for Modula indicate that the solution avoids most overhead, but it still has some weaknesses in its solution.

From this figure we can see that access control systems have developed at an uneven pace. Static access rights have been the major concern, with steady improvement being made. The Modula approach satisfies many

			PL/I	Concurrent Pascal	Modula	Managers
Language-enforced data integrity			No	Yes	No	Yes
Static Access Rights	no access	block real disk	Poor	Good	Fair	
	full access	file system	Good	Poor	Good	
	selective-entry access	receive-only messages	Poor	Poor	Fair	
	use without create	remote terminals	Poor	No	Good	
Dynamic Access Rights	full vs. no access	virtual disk				Good
	selective entry	read-only file				Poor
	use without create	remote terminals				Good
	general rights sharing	file system				Poor
	rights revocation	ownership situations				No
	monitors as managers	buffer pool				Good
	processes as managers	garbage collector process				Poor

Figure 2.17: Summary of access control mechanisms.

of our needs by enforcing all static control in the compiler. However, there is still room for improvement. In the dynamic case, work is only beginning at the language level. The manager system provides one highly-structured approach, but it lacks general flexibility. Certainly, much work remains to be done.

2.4. Our Work

Our contributions in the field of high-level multiprogramming languages are concentrated in three basic areas:

1. Extensions to monitors for improved interaction ability,
2. New mechanisms for static and dynamic access control, and
3. Analysis of program properties.

The work in the first two areas has been directed at eliminating the weaknesses of mechanisms discussed in this chapter. Our language features meet almost all the expressiveness criteria we have enumerated. The third area, analysis of program properties, relates concepts like deadlock [34] and safety [30] to the particular programming environment we have set up. For example, we want to know what can be done to avoid deadlock, or at least recognize it when it happens. Our work in this area hopefully provides a useful foundation for more extensive research.

Our modifications to monitors serve to unify the representation of physical objects (like disks) with virtual objects (like buffers and files). In this respect we are in agreement with Modula (i.e., its interface and device modules) but our approach is simpler. The most important modifications to monitors permit safe, simultaneous usage by multiple processes. As a result, we can solve problems like readers/writers and disk scheduling with no difficulty. In fact we can cleanly solve every problem in our expressiveness criteria for interaction, with one exception. We cannot permit general access overlap (as represented by an on-the-fly garbage collector) because of our constraint for insuring data integrity. We call our extended monitors RESOURCES in order to emphasize their general utility.

In the area of access control, we propose three new mechanisms. The RESTRICT pseudo-statement is used for controlling static access rights. It is similar to Modula's USE-list option in that both are handled strictly by the compiler. The key difference is that this feature gives outer blocks explicit control over the objects that inner blocks use. Instead of an inner block requesting USE of some object, an outer block dictates which objects can be seen. This feature easily meets all of our expressiveness criteria for static access rights.

The second access mechanism we propose permits dynamic access to resources. Our system is based on the concept of capabilities [24], and the use of parameter passing to move objects from one environment to another. In comparison to managers, our system is more basic, and thus avoids some of the policies that were imposed in that system. We can implement all but one of the problems we described under dynamic access control. In our system, unilateral revocation of rights by an "owner" is not feasible. However, the other problems have a much simpler solution than is possible with managers.

Our third access mechanism handles a special case of dynamic rights. If an object is moved among various domains in such a way that only one process at a time has access rights to it, then synchronization protection is not necessary. Moreover, in this case the object could safely be some data variable. An excellent example of this situation is a buffer scheme. If a buffer is implemented as a resource then unnecessary overhead is incurred for procedure calls and synchronization. To accommodate this simple type of application we introduce protected objects and pointers which can safely and efficiently solve the problem.

Our final area of research involves the analysis of several program properties. We first show that our language facilities enforce the property of data integrity. We also develop a model of our language for analyzing problems associated with deadlock. This model permits us to list conditions necessary for avoiding deadlock (which turn out to be excessively restrictive) and also develop an algorithm for detecting when deadlock has occurred. In a similar fashion we have developed a model for analyzing access rights in our system. It permits us to address problems such as safety [30] and information flow [18,45].

In the next four chapters we present our language proposals.[8] They are analyzed in Chapter 7 using the criteria developed here. Finally, in Chapter 8 we discuss the program properties mentioned above.

[8] The language facilities are also summarized in Appendix A.

CHAPTER 3

A BASIC LANGUAGE FOR PARALLEL PROGRAMS

We begin this chapter by discussing general language characteristics which have a bearing on our requirements and goals. In particular, our integrity constraints (language, data, and abstraction) are properties of an entire language--not just interaction and access control facilities. Carefully defined and controlled interaction facilities are ineffective unless all parts of the language provide the same level of protection. The rest of this chapter describes the specific language we use as a basis for incorporating our mechanisms. We have chosen to work from the sequential language PASCAL [33,62]. It is slightly modified to avoid conflict with features we will introduce later, and then extended with a system for defining parallel execution. The resulting language is one example of an effective environment for incorporating our interaction and access control facilities.

3.1. Language Characteristics

In the first chapter we enumerated three basic language requirements that are critical for meeting our design and analysis goals. To briefly summarize, they are:

1. Language integrity--the ability of a compiler to accurately translate a program into machine-executable form and prevent any attempts at subverting the intended semantics of the language.

2. Data integrity--the prevention of any simultaneous use of data variables, except when all such uses are only for reading.
3. Abstraction integrity--the support of modular design techniques (i.e., preventing one block of code from tampering with abstractions defined in another part of the program).

These three properties must hold throughout the entire language--not just in particular facilities. For example, data integrity in interaction mechanisms is useless if the multiprogramming mechanism permits misuse directly. As a result, these integrity constraints (particularly the first two) have some important implications for the types of facilities we can include in our basic language, and how they are implemented.

Language subversion is a very difficult action to prevent in any high-level language. Some forms of it are obvious, and easy to prevent. For example, we cannot permit a feature like the INLINE function in SUE, which permits machine code to be directly embedded in the program text. Unfortunately all cases are not this obvious. Most often, a language is subverted by an incomplete or incorrectly-implemented compiler. Legitimate language facilities (like pointers, arrays, parameter passing, and CASE statements) can be used improperly. For example, a pointer variable might be modified to reference a block of compiled code instead of the intended object. The pointer could then be used to change that code. Proving that a particular compiler prevents subversion is extremely difficult, if not impossible. However, most of our results are applicable to that implementation only if it is true.

The data integrity constraint also affects the acceptability of some language features and implementations. The obvious place for concern is in the mechanism for multiprogramming. PL/I's TASK option is one example that fails data integrity on this count. In PL/I a new process is created by calling a procedure and specifying that this invocation is a TASK. Hence, a procedure block may be invoked once as a process and another time as a procedure. Data integrity can be broken by passing a variable by reference to a task. In this case both the caller and the new task could simultaneously write to that variable. A more subtle violation can also occur since procedures can access globally-defined variables. Here the variable need not be passed, only known by scope rules to both the caller and new task. A third type of violation arises from interplay which is possible through the goto statement. This almost unlimited branching power could actually permit two processes to execute the same code operating on the same data.

Data integrity could also be violated through implementation failures. For example, a problem could arise with I/O procedures defined by the language. If two processes attempt to use the same device at the same time, some status data associated with the device could be completely scrambled causing many different types of damage. This form of violation is not always obvious, and yet it is critical to avoid in the operating system applications with which we are concerned.

The third constraint, abstraction integrity, is mostly related to the mechanisms we will add later, but it does place one restriction on the general language. We must be able to identify every access to an object (whether it is data, procedures, or whatever). Hidden accesses could destroy a particular abstraction. One example of this case (which actually violates all three integrity constraints) is the lack of a language requirement for array bounds-checking in FORTRAN. This property makes it possible for almost any variable or any program code to be indirectly accessed by an out-of-range subscript.

3.2. Modifications to Standard Pascal

This section describes the specific language context which is used in all of the succeeding chapters. Pascal was chosen as a starting point because it is a simple, well-defined language.[1] The difficulty with selecting an existing language as a base for defining new facilities is that incompatibilities invariably exist. We have that problem with Pascal. In order to avoid conflicts with some of our new facilities and also to conform with our language needs, we make some simple changes to sequential Pascal. We then add process blocks for identifying separate processes and add special procedures for creating and managing them.

3.2.1. Changes to Sequential Pascal

We have chosen to delete three features from standard Pascal. The first is the pointer type and all operations associated with it. (We introduce some similar facilities in Chapter 6, so these features are deleted to avoid syntax conflicts and general confusion.) The Pascal pointer concept is basically compatible with our interaction facilities; however, to incorporate it would require some extra rules which would decrease clarity. Since our examples do not need this facility we have chosen to avoid its use.

---

[1]We assume that the reader is familiar with it, but it is not critical to the understanding of this work. Detailed information on PASCAL can be found in [38,62].

The second deletion is the goto statement. This change is made for strictly practical reasons (although we sympathize with the philosophical ones as well!). The problem of controlling the scope of statements executable by a particular process is complicated by the presence of the goto. As with pointers, we could specify sufficient constraints to allow limited use of goto's, but it would serve no useful purpose. Pascal has a sufficiently rich set of control statements to make the goto unnecessary.

The last major omission from Pascal that we make involves its I/O control mechanisms. We assume that the language will have no operating system type of support, hence, the language must be able to control all external devices. We also have the problem of synchronizing access to devices, which is not a problem in a sequential language. Rather than expanding on Pascal's approach to I/O, we delete their system and, in Chapter 4, introduce our own system which treats I/O as a special case of process interaction.

Other modifications involve extending facilities such as parameter passing and scope rules. The changes are required to cover new situations created by multiprogramming. For example the scope rules must be expanded to handle process blocks. These modifications will be covered as the need arises. In general, the extensions will not affect the use or meaning of any Pascal facilities in situations that are purely sequential in nature.

3.2.2. Features for Multiprogramming

The next step in our specification of a basic language is to define facilities for multiprogramming. Since the next chapter will discuss interaction mechanisms, our purpose here is to define features for allowing concurrent execution without <u>any</u> interaction. Each process is completely isolated from every other one in the system. We prohibit shared variables and any other means that might allow communication. In this way, when interaction mechanisms are introduced, we will know that they are the only ones available.

The extentions we make here are similar to facilities implemented in EPL [43], Concurrent Pascal, and Modula. Tasks are separated into <u>process</u> blocks and specific commands are provided for creation and management of processes. Allocation of execution time to each process is done by the language-defined scheduler; however, scheduling policy can be administered within a program.

Processes are declared by using a new object type called <u>process block</u>.[2] We use the following notation to represent them syntactically:

```
P_NAME: process;
            <local declarations>
            <code body>
            end P_NAME
```

All process blocks must be defined in the declaration portion of a Pascal block. Each process block may declare instances of any type of object normally allowable in the language. The code body for a process specifies the instructions which can be executed in parallel with other portions of the program. The major restriction on process blocks is that they can not access any variables (simple or structured) declared outside their scope.

---

[2] We use the term "block" to refer to a named unit of a program delimited by a header and the key word <u>end</u>. It will only be used in reference to <u>program</u>, <u>procedure</u>, <u>process</u>, and <u>resource</u> (next chapter) units; not <u>begin</u> or <u>do</u> groups.

With one exception, this rule also applies to the access of globally-defined procedures. This restriction is necessary to enforce process isolation; it will be discussed in detail later.

An instance of a process may be defined in two ways: either directly by declaring a specific process, or indirectly by first declaring a new process type. Both options are displayed in Figure 3.1. Case A declares a process object named USER and associates it with the process block adjacent to it. Case B first declares a type of process called TERMINAL, and then creates CONSOLE as one instance of the type. We have extended Pascal's <u>type</u> feature to permit a restricted form of parameter passing to a process when it is created: all variables, simple and structured, must be passed by value. This extension allows several nearly identical processes to be declared with one specification of the code and yet maintains data integrity during execution.

This approach for defining simultaneous execution permits complete process isolation. Each process is limited to executing within a specific process block. It begins with the first statement of the block, and when it reaches the end of its block, it automatically dies. Since <u>goto</u>'s have been eliminated, flow of control cannot pass out of that block (except for some procedure calls). Also, as we have said, each process can only use variables which it declares itself. In the case of two processes defined through the same TYPE statement, each process has its own set of local variables. As a result, no variable may be directly shared by two processes.

The key remaining question is whether or not to permit a process to invoke a procedure defined outside its scope. In practice, it seems

```
SYSTEM: program;
          :
     type TERMINAL = process (DEVICE: device_id);
                          :
B                    end TERMINAL;

     var CONSOLE : TERMINAL(CONSOLE_ID);
          :
        USER: process;
A                 :
              end USER;

     <code body>

     end SYSTEM
```

Case A: simple process declaration.

Case B: process declaration using *type* feature.

Figure 3.1

useful.[3] But if we allow procedures to access variables outside their scope (which is also useful) we lose isolation. One solution would be to prohibit either one type of access or the other--but that is too restrictive. Our solution is that a process may call outside its scope to a procedure which accesses no global variables.[4] This rule can be enforced at compile-time.

Control of processes is achieved through the following four management operations which are automatically defined for each process:

NAME. <u>setstate</u> (priority, quantum, limit)
NAME. <u>readstate</u> (priority, quantum, limit, used, status)
NAME. <u>activate</u>
NAME. <u>suspend</u>

The NAME field identifies the process being controlled (it is the name used in declaring a process instance). <u>Setstate</u> gives a process an exec tion priority, time quantum, and time limit. A process cannot begin execution until an execution state is specified for it. Once a process has a state, it only requires an <u>activate</u> command to begin. (See Figure 3.2 for an example of a process initiation.) The <u>suspend</u> operation can used to temporarily halt the execution of a process; <u>activate</u> will rest it. Information about an executing process can be acquired through <u>readstate</u>. The current value of a process' state is returned in the parameters. The USED field indicates the total time that process has

---

[3]It could be viewed in our operating system environment as the equivalent of a supervisor call.

[4]To insure data integrity, such calls would probably need to be c piled reentrantly.

```
SYSTEM: program;
        :
    type TERMINAL = process (DEVICE: device_id);
                    :
                    end;

    var CONSOLE: TERMINAL(CONSOLE_ID);

          USER: process;
                  :
                  end;

        HIGH,LOW: integer;             /* priority variables   */
        SLICE,LIMIT: integer;          /* time limit variables */
        :
        HIGH := 1;
        LOW := 10;
        SLICE := 50;                   /* time slice limit     */
        LIMIT := 1000;                 /* some time limit, say 1000 ms */
                /* keyword 'infinite' indicates no limits on constraint */
        CONSOLE. setstate (HIGH, infinite, infinite);
        USER. setstate (LOW,SLICE,LIMIT);
        CONSOLE. activate;
        USER. activate;
        :
        end SYSTEM
```

Figure 3.2: Initial process activation.

used, and the STATUS field indicates its current execution state--which is either ACTIVE, SUSPENDED, or TERMINATED.

Processes are scheduled as follows.[5] Each process has a priority, time quantum, and time limit which affect scheduling decisions. The highest priority task that is ready to execute is always selected for execution. If more than one process satisfies this criterion, they are executed in round-robin fashion. The time quantum field for each process indicates the amount of time to be given to a process each time it is scheduled. Any process that exceeds its total time allotment is automatically suspended. (It may be restarted by a <u>setstate</u> followed by an <u>activate</u>.)

Finally, we permit the nesting of processes in our language. A sample program using this feature is outlined in Figure 3.3. The rules we use to cover this situation are relatively simple. The outermost block of a program actually defines a special process which is automatically activated to begin execution. Every process then has the ability to declare new processes of its own; however, only the process that declares a new process block may create and manage instances of that type. For example, only SYSTEM can use the management operations on USER and CONSOLE. This rule also implies that USER could not create any other TERMINAL instances. A process has complete responsibility for processes that it creates. In particular, a process cannot terminate (reach the end of its execution block) until all of its progeny have terminated. For example, Figure 3.4 shows the family tree that arises from the program in Figure 3.3. This

[5]This scheduling is accomplished by a run-time support package required by the language.

```
SYSTEM: program;
    type TERMINAL = process (DEVICE: device_id);
                    :
                    end;

    type USER = process;
                    var ALPHA: process;
                                :
                                end;
                        BETA:  process;
                                :
                                end;
                    :
                    ALPHA. setstate (    );
                    ALPHA. activate;
                    BETA. setstate (    );
                    BETA. activate;
                    :
                end USER;

    var CONSOLE: TERMINAL (CONSOLE_ID);
        USER_1,USER_2: USER;
    :
    USER_1. setstate (    );
    USER_1. activate;
    USER_2. setstate (    );
    USER_2. activate;
    :
    end SYSTEM
```

Figure 3.3: Sample program using nested process blocks.

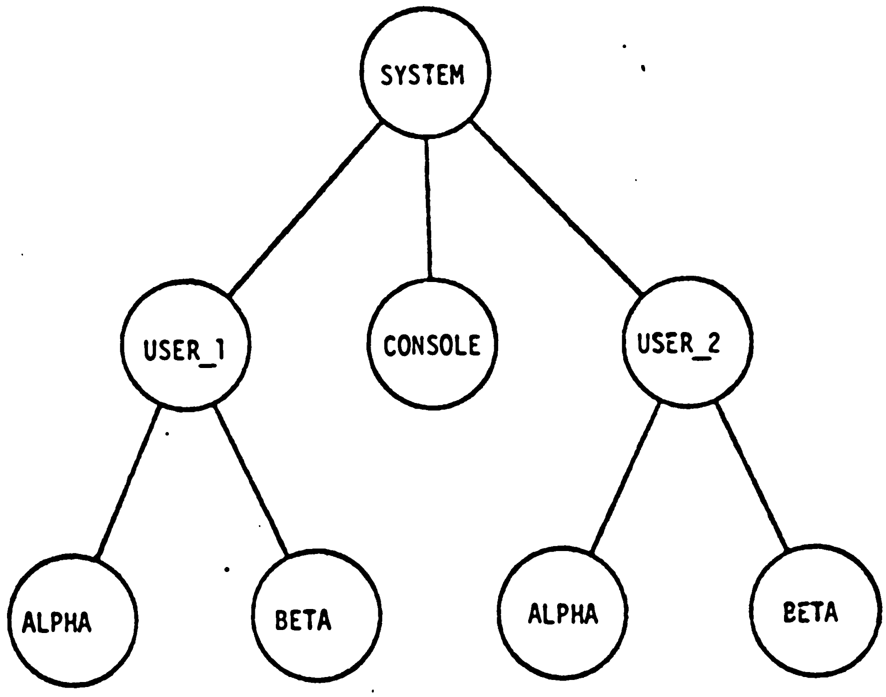


Figure 3.4: Run-time process structure for program in Figure 3.3.

termination rule specifies that a process cannot die until all processes in its subtree are gone.

Our major motivation for selecting this approach to multiprogramming is to provide the broadest possible environment for discussing interaction. This position is consistent with our philosophy that expressability is more important than efficiency of implementation. For example, we permit nesting of processes, which neither Concurrent Pascal nor Modula allow. They prohibit it because implementation would probably be complicated by a messier memory allocation system and because they assume that all processes and interactions are permanent. Unfortunately, one level of process structuring limits designs. For example, a user process would not be able to build up his own subsystem of processes. Since we are examining the general concept of process interaction and how it might be usefully applied, we have moved in the opposite direction. We have made our multiprogramming facilities as general as possible in order to provide a robust environment for embedding process interaction.

CHAPTER 4

THE RESOURCE CONSTRUCT FOR PROCESS INTERACTION

In the previous chapter we defined a language environment that allows a program to be a set of isolated processes. Every object in the language belongs to only one process with no possibilities for interaction. This chapter introduces a new type of program unit, a resource, which defines a meeting point where processes can interact in a controlled manner. It can own and control shared objects just as process units do with private objects.

A resource is a variant of the basic monitor concept [33]. In Chapter 2 we argued that although monitors are more expressive than any other existing mechanism, they still have some definite weaknesses. We keep the monitor framework but make the following changes to avoid these problems:

1. an optional POST_OP procedure may be defined--it is invoked automatically before a resource dies;
2. condition variables are replaced with a more expressive form of synchronization;
3. a parallel clause allows multiple processes to simultaneously enter a resource when it is safe to do so; and
4. a release clause permits multiple processes to be active in a nested resource structure.

These changes permit a resource to directly express interaction designs that were difficult or impossible before. As a result, the resource mechanism provides a unified approach for representing real objects (printers, disks, etc.) as well as virtual ones (messages, files, etc.).

4.1. Basic Resource Description

Figure 4.1 outlines the notation we use for defining a resource. In this section we discuss its basic features; the special extensions are discussed in subsequent sections. A resource associates a set of data with procedures (OP1,...,OPK) for operating on that data. Only the procedures within a resource may access its data. Interaction occurs when processes invoke the resource operations. In order to insure data integrity (of the resource variables) the default access policy is that only one process at a time may execute within a particular resource. Later, we will discuss methods for safety relaxing this rule.

The <access attribute> of a resource defines the mechanisms that must be used in order to create and use an instance of it. The static attribute allows a resource to be manipulated similar to normal variables. An instance is created via a declaration at the beginning of a block. Access to the resource within that block is controlled by scope rules. When execution inside the block is finished, the resource automatically dies. The only difference between the control of static resources and variables is that a static resource cannot be passed as a parameter.

An <access attribute> can also be dynamic. An instance of this type of resource is generated by executing a create statement rather than by a declaration. Its existence is independent of the block in which it is created, and it may be passed as a parameter. In this chapter and the next we will restrict our discussions and examples to static resources;

```
RES_NAME: <access attribute> resource;
          operations OP1,...,OPK;
          parallel* (PROC_L,...,PROC_M),...(PROC_N,...,PROC_P);
          release* PROC_R to PROC_S,
                      :
                   PROC_T to PROC_V;

          <resource data declarations>;

          OP1: procedure (   ); ... ; end OP1;

          OPK: procedure (   ); ... ; end OPK;

          <local resource procedures>

          PRE_OP*: procedure; ... ; end PRE_OP;

          POST_OP*: procedure; ... ; end POST_OP

          end RES_NAME


          * optional
```

Figure 4.1: General resource structure.

however, all the rules for defining resource blocks apply to both options. We will examine dynamic resources in Chapter 6.

A resource block is a new object type in the language (like integer, record, and process). Figure 4.2 gives a skeletal program structure demonstrating the use of static resources. SYSTEM contains three static resources, FILE_1, FILE_2, and LOG, which are accessed by processes USER_1 and USER-2. The log is used to record when a process accesses the file. This example demonstrates the two ways in which static resource instances can be generated. LOG is declared by associating the name directly with a resource block. The other method is to first declare a type of static resource (like FILE) and then use the type name to declare independent instances (like FILE_1 and FILE_2). Parameterized TYPING of resources is permitted, but only constants and variables may be passed (by value) as actual parameters. The scope rules permit static resources to be accessed from inside process blocks (unlike variables). For example, LOG is accessed by both processes even though it is declared globally to them. Calls to static resources use the same dot notation found in monitors.

FILE_1 and FILE_2 are separate, independent resources; they do not share variables and they have no common synchronization. USER_1 may enter FILE_1 while USER_2 is in FILE_2 with complete safety.

A static resource dies when the process that defined it terminates. In this example, when SYSTEM terminates, all of the resources disappear. In order to insure that USER_1 and USER_2 do not attempt to access a resource after it dies, we insist that a parent process cannot terminate until all of its progeny have terminated.

```
SYSTEM: process;

    type FILE = static resource (SIZE: integer);
                operations READ,WRITE;
                   :
                end FILE;

     var FILE_1: FILE(10);
         FILE_2: FILE(5);

             LOG: static resource;
                   operations ENTER;
                     :
                   end LOG;

             USER_1: process;
                       :
                      FILE_1.READ(   );
                      LOG.ENTER(   );
                       :
                     end USER_1;
             USER_2: process;
                       :
                      FILE_1.WRITE(   );
                      LOG.ENTER(   );
                       :
                     end USER_2;
              :
             end SYSTEM
```

Figure 4.2: Using static resources.

The rest of the basic resource description deals with the internal specifications of a resource block. In a resource only procedures listed in the operations clause may be invoked from outside. The only procedures which may not appear on the operations list are PRE_OP and POST_OP which will be discussed in the next section.

The scope rules for resources are basically the same as those for processes. They are:

1. a resource cannot access variables declared outside its block,
2. a resource can invoke resources declared outside its block, and
3. a resource can invoke pure[1] procedures.

In cases 2 and 3, when a resource calls outside, the process making the call retains control of the resource (so that another process cannot enter it). Later we will consider safe relaxations of this rule. The scope rules listed above (particularly 1 and 3) are necessary to insure data integrity. For example, in Figure 4.2 if SYSTEM declared a variable that LOG could access, then SYSTEM and USER_1 could simultaneously modify it, SYSTEM directly and USER_1 operating through LOG. Similar problems arise if LOG could invoke a procedure which could access the variable declared by SYSTEM; hence the need for pure procedures.

4.2. Initialization and Termination

One of the features in monitors is the optional specification of some initialization code. This code is automatically executed when a monitor comes into existence, thus insuring that it can be set up properly

[1] A pure procedure is one that accesses no variables outside its scope and only calls resources and other pure procedures.

before any accesses are made and also insuring that the initialization is only done once. In the resource concept this option is retained, with the extension that a similar block of code may be specified for execution just prior to the resource's destruction. In order to separate the two blocks we require that they be placed inside PRE_OP and POST_OP procedures, respectively. Neither procedure may be invoked by any process; each one is automatically invoked at the proper time (during creation or just before destruction).

We can use this mechanism to solve the virtual printer problem posed in Chapter 2. To briefly summarize, the problem is to define a facility for spooling a single user's output to a printer. The facility accumulates all the lines to be printed, then transmits them to the printer in one continuous stream. Most solutions to this problem require special code to identify when a user has completed so the spooler can begin the actual printing. The POST_OP option provides a clean solution without this need.

The general structure of the solution is shown in Figure 4.3. For now, we assume that the operating system has defined some resource, PRINTER, for controlling I/O to that device. The virtual printer is defined as a type of resource which uses the real printer. A user then defines his own instance of the type and uses it until he is done. For clarity, each program block will convey output lines through variables of type PRINT_LINE (some globally-defined structure).

The details of the virtual printer type are shown in Figure 4.4. Each time an instance is declared a maximum number of output lines is specified. Lines are added to the spooled file by calling OUTPUT, which

```
SYSTEM: program;
          :
          PRINTER: static resource;
                   operations OUTPUT,ACQUIRE,RELEASE;
                   :
                   end PRINTER;
          :
     type VIRTUAL_PRINTER = static resource (MAX_LINES: integer);
                            operations OUTPUT;
                            :
                            end VIRTUAL_PRINTER;
          :
          USER: process;
                var PRINT: VIRTUAL_PRINTER(100);
                    LINE: PRINT_LINE;
                    :
                    for I := 1 to 100 do
                         :
                         PRINT.OUTPUT(LINE);
                         end;
                         :
                    end USER;
          :
          end SYSTEM
```

Figure 4.3: Program structure for spooling solution.

```
type VIRTUAL_PRINTER = static resource (MAX_LINES: integer);
        operations OUTPUT;
        var LINE_FILE: array [1..MAX_LINES] of PRINT_LINE;
            CURRENT_LINE: 0..MAX_LINES;

        OUTPUT: procedure (LINE : PRINT_LINE);
                if (CURRENT_LINE = MAX_LINES) then return;
                CURRENT_LINE := CURRENT_LINE + 1;
                LINE_FILE(CURRENT_LINE) := LINE
                end OUTPUT;

        PRE_OP: procedure;
                CURRENT_LINE := 0
                end PRE_OP;

        POST_OP: procedure;
            var I: 1..CURRENT_LINE;
                PRINTER.ACQUIRE;
                for I := 1 to CURRENT_LINE do PRINTER.OUTPUT(LINE_FILE(I));
                PRINTER.RELEASE
                end POST_OP

        end VIRTUAL_PRINTER
```

Figure 4.4: Spooling printer output using POST_OP.

also prevents a process from exceeding its limit. When an instance is ready to be destroyed (i.e., the USER process is dead), POST_OP transmits all lines to the printer. In order to insure output continuity the real printer must provide an allocation mechanism (here we use ACQUIRE and RELEASE) which permits one virtual printer at a time to transmit all of its lines. The significant part of this solution is that neither the SYSTEM nor the USER is required to take any explicit action to cause the spooler to print the output; POST_OP handles it automatically. As a result, all of the spooling control is implemented in one resource.[2]

The POST_OP option is also useful in a more general type of problem. An operating system may want to give some object to a user, but with the assurance that it will be returned at a later time. We can extend the spooling problem to demonstrate this situation. Normally spooled output is stored in secondary memory to decrease main memory costs. The spooler must acquire drum space (on behalf of the user) and hold it until the transfer to the printer takes place. The symmetry of PRE_OP and POST_OP permits a simple solution. PRE_OP acquires the space and the last action by POST_OP is to return it. This strategy is encorporated in our design of a file system which is discussed in Chapter 7. The important facet of this approach is that the USER cannot "forget" to return the space when he completes execution.

[2] In this solution, the user is required to remain in core until the output is printed. In Chapter 6, we introduce features which will permit us to avoid this weakness.

4.3. Program-controlled Synchronization

We now introduce a mechanism for permitting program-controlled synchronization. The await primitive [33] is a special operation that can only be invoked inside a resource. Its form is:

await b

where b is an arbitrary Boolean expression. If await is executed and b is true, control immediately passes to the next instruction. If b is false, the process is delayed and temporarily leaves the resource so that another process can enter. The process may resume execution when two conditions are simultaneously met:

1. the Boolean expression b is true, and
2. the resource procedure may safely resume execution.[3]

When this happens, control passes to the instruction following the await.

Figure 4.5 illustrates the use of await in a resource which implements a bounded buffer. (Figure 2.5 solved it using monitors and condition variables.) Synchronization is needed in two places--one to insure that a new element is not added to an already full buffer, and the other to prevent a REMOVE from operating on an empty buffer. The major difference between this solution and the previous one is that there is no explicit signalling mechanism. If one process is stalled in REMOVE, it cannot proceed until another process calls APPEND and increments the COUNT variable. Notice that by the second condition discussed previously, the

---

[3]Usually this condition means that no other process is executing a resource procedure, but it is complicated by the possibility for parallel execution discussed in the next section.

```
BOUNDED_BUFFER: static resource;
    operations APPEND,REMOVE;

    var BUFFER: array 0..N-1 of PORTION;
        FIRST_OUT: 0..N-1;
        COUNT: 0..N;

            /* Invariant: elements are stored in the order        */
            /* in which they were APPENDed in BUFFER entries      */
            /* BUFFER(FIRST_OUT),BUFFER(FIRST_OUT + 1 mod N), ..., */
            /* BUFFER(FIRST_OUT + COUNT - 1 mod N).                */

        APPEND: procedure (X: PORTION);
                await COUNT < N;
                BUFFER((FIRST_OUT + COUNT) mod N) := X;
                COUNT := COUNT + 1
                end APPEND;

        REMOVE: procedure (X: PORTION);
                await COUNT > 0;
                X := BUFFER(FIRST_OUT);
                FIRST_OUT := (FIRST_OUT + 1) mod N;
                COUNT := COUNT - 1
                end REMOVE;

        PRE_OP: procedure;
                COUNT := 0;
                FIRST_OUT := 0
                end PRE_OP

        end BOUNDED_BUFFER
```

Figure 4.5: Bounded buffer problem demonstrating await.

stalled process must actually wait until the APPEND is completed even though the Boolean condition is now true.

This example brings out an important question about scheduling: if two processes are waiting in REMOVE for data, and one item is APPENDed, which process proceeds? More generally, if more than one process can be freed from a delay, which one goes first? An implicit part of each resource is a scheduler which must resolve these questions. Our only requirement is that this scheduler be fair--i.e., it cannot indefinitely discriminate against a particular process. The first-come, first-serve algorithm is one example of a fair scheduling algorithm, but other good choices exist. We refrain from specifying a particular one because this decision need not be made at this level. Also, resources should not be coded to take advantage of this algorithm. However, we must reiterate that this is still a language-level decision; it would be very difficult to pass it on to a user of the resource mechanism.

Our choice of the <u>await</u> mechanism instead of Hoare's condition system is based on clarity and understandability. In the condition system, a programmer must associate the actual event he wants to wait for with a specific condition variable. Moreover, he is solely responsible for signalling it at the proper time. It is possible that he will incorrectly signal a condition variable when the actual event has not occurred, or forget to signal when it has. In the <u>await</u> system the event can be directly represented by a Boolean expression. Since "signalling" is automatic, a statement following an <u>await</u> is absolutely certain that the Boolean is true. The only responsibilities left to the programmer are to insure that the Boolean expression actually specifies the condition he needs, and that at some later point it will become true. Based on these arguments we feel that the await system is much clearer and easier to use.

The argument that Hoare uses to support his choice is efficiency. The time spent in evaluating expressions could be significant. This point is definitely valid. In an implementation of await some precautions (such as only evaluating expressions of delayed processes where variables in the expression have been modified) may save some time, but it is not clear how much they would help.

Our choice in this case is based on our priorities for language design. Top priority is expressiveness, and in this light clarity must take precedence over efficiency.

4.4. Parallel Clause

In the monitor concept and the resource mechanism as described so far, only one process at a time is permitted to execute within the interaction facility. This rule insures that two processes do not simultaneously attempt to use the same resource variable. Unfortunately, this rule also precludes some completely safe situations where more than one process is actively working inside a resource. For example, the readers/writers problem [15] is based on permitting reader processes to simultaneously access a data base. In this section we introduce the parallel clause for resources which permits this type of concurrent activity, without sacrificing data integrity.

The parallel clause specifies exceptions to the general rule that only one process at a time may be executing within a resource. The syntax for this option is:

parallel (PROC_M,...,PROC_N),

⋮

(PROC_P,...,PROC_S)

If used, this clause must directly follow the operations list. Procedure named in this option must be selected from those declared as a part of th resource; PRE_OP and POST_OP are the only ones that cannot be named. The parallel clause lists groups of procedures which may simultaneously execute within the resource. For example, if one process is executing PROC_M, and a second process attempts to invoke PROC_N, the call is permitted to take place immediately. The simultaneous executions defined by parallel are the only ones permitted inside a resource--i.e., the concept is not transitive. If a clause specifies:

parallel (A,B),(B,C)

A and C cannot execute simultaneously. This form of specifying simultaneous execution is a very restricted version of path expressions [12].

We illustrate the use of parallel with a simple example. Figure 4 shows a solution to the basic Readers/Writers problem. This resource simply insures that several readers can proceed simultaneously and that when a writer executes, he has sole possession of the data base. For th example only one variable (STORE) will be shared, but in any practical data base situation, the sharing would involve large arrays and probably search procedures to determine the proper entry. The parallel clause, specifying "(READ,READ)", permits any number of processes to simultaneously call READ.[4] Since WRITE is not listed in the parallel clause, i

[4]This multiple use of READ has implications for data integrity. Each invocation of READ must be completely disjoint--e.g., each one mus

```
DATA_BASE: static resource;
           operations READ,WRITE;
           parallel (READ,READ);

           var STORE: integer;

           READ: procedure (VALUE: integer);
                 VALUE := STORE
                 end READ;

           WRITE: procedure (VALUE: integer);
                 STORE := VALUE
                 end WRITE;

           PRE_OP: procedure;
                 STORE := 0
                 end PRE_OP

           end DATA_BASE
```

Figure 4.6: Readers/Writers solution (default reader priority).

invocation of it will always execute by itself inside OATA_BASE. For example, assume the requests READ,WRITE,REAO,WRITE come in succession (from four different processes). Assuming the rescurce is free, the first REAO will proceed immediately. The WRITE will be delayed because READ and WRITE cannot execute in parallel. If the second READ come before the first finishes, it will be able to execute immediately. After they both finish, the two WRITES will be allowed to complete, but only one at a time. From this description it should be clear that by default this example implements readers' priority.

If a scheduling policy (e.g., writers' priority) is desired, it can be enforced by using parallel in a slightly different way. Figure 4.7 demonstrates a solution with writers having priority over readers. The resource defines four procedures; the two operations READ and WRITE order the calls to OOREAO and OOWRITE which accomplish the actual data access. The groups listed in parallel permit new requests to enter the resource while a DOREAO or DOWRITE is taking place. The double listing of DOREAD in the first group permits simultaneous reading, as before. Notice that two OOWRITES cannot proceed together, and that only one request at a time can be received (i.e., REAO and WRITE cannot operate simultaneously). The key to this solution is that when REAO calls OOREAO, the process taking the action is considered as being temporarily outside of READ (permitting others to call the resource operations). Consequently, when OOREAD returns to READ, the executing process may temporarily be delayed (because

---

have its own variable VALUE. In an implementation, this could be accomplished by reentrant coding.

```
DATA_BASE: static resource;
           operations READ,WRITE;
           parallel (OOREAD,OOREAD,READ),
                    (OOREAD,WRITE),
                    (OOWRITE,READ),
                    (OOWRITE,WRITE);

           var STORE: integer;
               WRITERS: integer;

           READ: procedure (VALUE: integer);
                 await WRITERS = 0;
                 DOREAD(VALUE)
                 end READ;

           WRITE: procedure (VALUE: integer);
                 WRITERS := WRITERS + 1;
                 DOWRITE(VALUE);
                 WRITERS := WRITERS - 1
                 end WRITE;

           DOREAD: procedure (VALUE: integer);
                 VALUE := STORE
                 end DOREAD;

           DOWRITE: procedure (VALUE: integer);
                 STORE := VALUE
                 end DOWRITE;

           PRE_OP: procedure;
                 WRITERS := 0;
                 STORE := 0
                 end PRE_OP

           end DATA_BASE
```

Figure 4.7: Readers/Writers solution with writers' priority.

some other process is in READ or WRITE). It is restarted as soon as that action is permitted, subject to the parallel constraints.

If the use of parallel is not restricted, our data integrity constraint could be violated. For example,

parallel (DOWRITE,DOWRITE) and

parallel (DOREADO,DOWRITE)

both allow simultaneously reading and writing. In order to prevent misuse, we impose some compiler-enforceable rules on the use of parallel. They are:

1. calls within a resource cannot pass resource variables as parameters; and
2. all procedures listed in a group must be interference-free [51]

Conceptually, insuring data integrity involves only the permanent variables of a resource (since only they are accessible to multiple processes). The first rule insures that we can accurately identify all accesses to those variables at compile-time (i.e., all accesses are static, using the variable name). The second rule, based on Owicki and Gries' concept of interference-free, insures that procedures which execu together cannot violate data integrity. We can define the concept as follows:

Definition 4.1: Given a procedure P_i, let R_i be the set of resour variables which P_i only reads, and let W_i be the set of resource variables that P_i modifies (include in W_i all resource variables passed as parameters).

Two procedures P_i and P_j are <u>interference-free</u> if and only if:

$$R_i \cap W_j = \emptyset \text{ and}$$
$$R_j \cap W_i = \emptyset \text{ and}$$
$$W_i \cap W_j = \emptyset .$$

In other words, P_i and P_j may only share variables that are read-only for both.

We can illustrate the interference-free concept with the resource defined in Figure 4.7. The analysis is done in Figure 4.8. We first compute the sets R and W for each procedure. Notice that we are only concerned with the permanent resource variables; local variables cannot be involved in simultaneous access because each procedure invocation has its own set of local variables. The table in Figure 4.8 shows which pairs of procedures are interference-free. From these entries it is clear that the <u>parallel</u> statement in Figure 4.7 satisfies our constraints. (We did not use two READs in the first group, even though it would be legal, because it would affect the priority of writers.) Moreover, from this chart it is clear that a compiler could detect an illegal use of <u>parallel</u>.

4.5. <u>Release Clause</u>

In the previous section we described a mechanism for relaxing the mandatory mutual exclusion rule within a monitor in safe situations. A second aspect of the mutual exclusion system involves calls from one monitor to another one defined outside the first. When such a call occurs, no process is permitted to enter the monitor that has been temporarily exited. As we saw in Chapter 2, this rule prohibits some useful system designs. In this section we introduce the <u>release</u> clause to alleviate this weakness, again without losing data integrity.

$$R_{READ} = \{WRITERS\}$$
$$W_{READ} = \emptyset$$
$$R_{WRITE} = \emptyset$$
$$W_{WRITE} = \{WRITERS\}$$
$$R_{DOREAD} = \{STORE\}$$
$$W_{DOREAD} = \emptyset$$
$$R_{DOWRITE} = \emptyset$$
$$W_{DOWRITE} = \{STORE\}$$

	READ	DOREAD	WRITE	DOWRITE
READ	Yes	Yes	No	Yes
DOREAD	Yes	Yes	Yes	No
WRITE	No	Yes	No	Yes
DOWRITE	Yes	No	Yes	No

Interference-free table.

Figure 4.8: Interference-free analysis of Readers/Writers solution.

The default policy on calls from one resource to another is the same as with monitors; the process making the call retains control of the procedure it was executing, even though it is temporarily executing elsewhere. The <u>release</u> clause permits explicit overriding of this policy. The syntax for this option is:

$$\underline{\text{release}}\ PROC_1\ \underline{\text{to}}\ PROC_2,$$
$$\vdots$$
$$PROC_{2N-1}\ \underline{\text{to}}\ PROC_{2N}$$

This clause must be specified in a resource's heading, just as with the <u>parallel</u> option. The procedures named as the first element of a pair must be ones defined inside the resource, and the others must be outside. The meaning of a <u>release</u> pair P_1 to P_2 is that if a process is executing in P_1, and calls out of the resource to P_2, then the process is temporarily leaving the resource. When P_2 returns to P_1, the process must be delayed until the state of the resource will permit a process to execute in P_1.

We can demonstrate the use of <u>release</u> by resolving the Readers/Writers problem (writers priority) with two resources (the solution is the same as in Figure 4.7, with READ and WRITE moved to a SCHEDULE resource). The actual data access is done by the resource defined in Figure 4.6 which gave reader priority. The SCHEDULE resource which reorders the requests into writers priority is defined in Figure 4.9. The <u>release</u> clause permits activity to occur in SCHEDULE and DATA_BASE simultaneously. For example, assume the following calls are made to SCHEDULE in succession: WRITE,READ,WRITE. Assuming that both resources are initially inactive, the first WRITE will immediately enter SCHEDULE, and then invoke the DATA_BASE. If the READ comes while the first WRITE is working in the

```
SYSTEM: program;
      :
   DATA_BASE: static resource;
              operations READ,WRITE;
              parallel (READ,READ);
                :  < body in Figure 4.6>
              end DATA_BASE;
      :
   SCHEDULE:  static resource;
              operations READ,WRITE;
              release READ to DATA_BASE.READ,
                      WRITE to DATA_BASE.WRITE;

              var WRITERS: integer;

              READ: procedure (VALUE: integer);
                    await WRITERS = 0;
                    /* following call gives up control of schedule */
                    DATA_BASE.READ(VALUE)
                    end READ;

              WRITE: procedure (VALUE: integer);
                    WRITERS := WRITERS + 1;
                    /* On call, give up control */
                    DATA_BASE.WRITE(VALUE);
                    /* On return, reacquire control */
                    WRITERS := WRITERS - 1
                    end WRITE;

              PRE_OP: procedure;
                    WRITERS := 0
                    end PRE_OP

              end SCHEDULE;
      :
   end SYSTEM
```

Figure 4.9: Use of <u>release</u> clause in a Readers/Writers solution.

DATA_BASE, that request will be permitted to enter schedule (since WRITE relinquished control). The READ will be delayed at the <u>await</u> statement, thus also giving up SCHEDULE. If the second WRITE comes next, it will update the variable WRITERS and call the DATA_BASE. The mutual exclusion in DATA_BASE will then delay it until the first WRITE completes. When the first WRITE returns from the second resource, it is delayed until no other process is using SCHEDULE. For example, if the return occurs while the second WRITE is in SCHEDULE it cannot complete the return until that WRITE calls DATA_BASE. Without the <u>release</u> clause, this simultaneous activity would not be possible. For the request sequence just described, each request would be processed fully (in and out of both resources) before the next one is recognized. There is one apparent weakness with this solution. Namely, a process may bypass the scheduler and call the data base directly. This problem will be resolved with access control mechanisms defined in Chapter 5.

In order to insure data integrity, we need one special rule: on calls out of a resource where the <u>release</u> mechanism is applied, none of the permanent resource variables may be passed as parameters. This rule is easily enforceable at compile-time so long as the resource variables cannot be accessed indirectly (as a result of parameter-passing) which we ruled out in the previous section. Without this rule data integrity could be violated.

The <u>parallel</u> and <u>release</u> facilities add a significant amount of flexibility to the resource mechanism, but not without increasing the complexity of interaction. One useful way of viewing these concepts is to examine their effect on the relationship between processes and resources.

Each process may be executing at most one procedure at a time within any one resource. When a process switches procedures within one resource (via call), it first gives up the old procedure and then starts the new one. When the call returns, the reverse holds true. Without the parallel option, these transitions can occur with no delay, since only one process at a time may execute in the resource (namely, the one that is already there). The parallel option permits some safe simultaneous executions, which introduces the possibility of process delays during call and return transitions. An example is shown in Figure 4.10. A resource is sketched which has some parallel clauses in it. The time line shows the use of the resource under one hypothetical calling sequence. In time periods 2, 5, and 7 processes are delayed on procedure transitions to avoid violating the simultaneous executions implied by parallel.

The release clause affects the number of procedures a process may simultaneously hold. Although it may hold only one per resource, it may involve several resources through nested calls. The standard call out of a resource permits the calling process to retain the first procedure, while using a second one. The advantage is that on return, no delay can be encountered. If the release is in effect for a call, the process gives up the first procedure before using the second. On return, a delay may be encountered for the same reasons we had in the parallel case.

Using this viewpoint, the meaning of parallels and releases in the same resource should be relatively clear. In monitors, the key concept was one process at a time inside; "control of the monitor" explained the necessary interactions. With resources, the key concept is "control of a procedure." The parallel clause specifies which control combinations are

```
RES: static resource;
     operations R1,R2;
     parallel (R1,R3),
              (R1,R2);
     :
     (assume R1 calls R3)
     :
     end RES
```

Hypothetical time sequence involving three processes P1, P2, and P3:

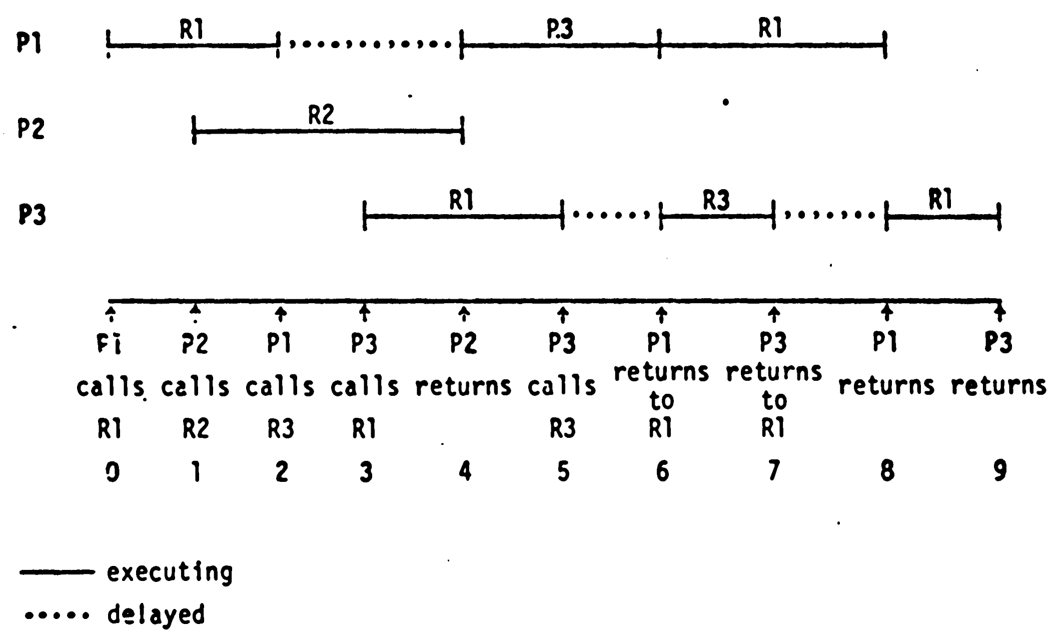


Figure 4.10: Effect of <u>parallel</u> on procedure transfers.

legal within a resource. As long as a particular situation meets these constraints, no process is delayed. The release clause assists in the execution phase by identifying special cases of relinquishing and later reacquiring procedure control.

4.6. Resources for Real Devices

This section discusses the use of resources to represent process interaction with I/O devices. In Chapter 3, while defining our base language, we removed the standard I/O procedures from PASCAL. The justification was that the procedure concept is insufficient to specify the synchronization that must be imposed in a multiprogramming language. The resource mechanism has the necessary concepts for representing real devices. The problem is in deciding how resources should get control over them. We consider several alternatives and discuss the reasons behind our actual choice.

From a descriptive view, an I/O device and our resource mechanism have very much in common. Both have local storage areas and operations which define how those areas may be manipulated. But more importantly, both require exclusive access and synchronization. Therefore, we propose that every addressable device be represented by a resource. One possible structure for such a resource is shown in Figure 4.11. The actual disk operations, READ and WRITE, are naturally represented by resource operations of the same name. The mutual exclusion necessary for the disk is automatically enforced by the language.

This close conceptual match between real devices and resources has a positive impact on our design for process interaction as a whole. It means we can use one language feature for handling interactions in both

```
DISK: static resource;
      operations READ,WRITE;

      READ: procedure (LOCATION: DISK_ADDR,
                       DATA: BUFFER);

            <operations on disk to accomplish READ>

            end READ;

      WRITE: procedure (LOCATION: DISK_ADDR,
                        DATA: BUFFER);

            <operations on disk to accomplish WRITE>

            end WRITE;
      .
      .
      .
      end DISK
```

Figure 4.11: Resource structure for I/O device control.

real and virtual objects. From a system designer's point of view, this fact is useful because problems like scheduling and hierarchical structuring are not affected by the type of object residing at the lowest levels. The uniform treatment of all interaction objects simplifies understanding and use.

So far, we have only stated that devices are represented by resources; we have not actually added anything to make this possible. In almost all computing systems, real devices deal with channel commands, interrupts, and other hardware-oriented concepts. Several feasible solutions are available for bridging the gap between resources and these aspects of real devices.

One approach is to permit specially designated resources to use machine code directly. This allows a system designer to access interrupt registers and status vectors in a simple and efficient manner. If the use of machine code is limited to resources, we get a simple device interface without giving users a means for subverting the system. Each special resource would be responsible for properly creating channel programs and fielding interrupts. The major problem with this approach is that language integrity is almost impossible to insure. While a general user may not be able to write in machine code, a systems programmer could still easily make a mistake. At this level, it would be extremely difficult for a compiler to detect errors. The other problem is the need for a precise definition of responsibility between the compiler and the program. Under our design, most interrupts are routinely handled by the language (e.g., CLOCK and ARITHMETIC). Giving all I/O interrupts to the program may be very difficult to do safely.

An alternative to this solution is to embed all I/O back into the language. For example, replace the I/O procedure calls in PASCAL with resource calls. In this case the responsibility of the language is to insure that all such calls look and act like resources as far as a programmer is concerned. Actual implementation by the language could take any form so long as it met this constraint. With this approach the language would handle all hardware interfaces like channel programs and interrupts. One justification for this strategy is that the language should be as machine-independent as possible. By keeping hardware details out, systems programs may be much more portable. The drawback is that some policy is being taken away from a system designer. He may want to specify operations on a device in a non-standard (possibly application dependent) way.

Our choice for giving device control to resources is somewhere between the two extremes presented. Interface with devices is done through resources defined in a program (as in the first option), but actual device access is done through procedure calls instead of machine code. So the language must define access procedures for each device in the computing system and insure that they are used only inside resources. For example, one procedure might be "DISK_IO (operation, location);". If several disks exist they must be represented by separate procedures. To insure integrity of a device, the language can require each procedure to be used in only one resource. The procedures (and hence the language) handle the actual control of the devices. When one is called, the proper linkage is created and the task is initiated. After the device signals completion (through an interrupt), the procedure returns to the resourc that made the call.

With this approach we get most of the advantages of the previous two. The system designer can structure his own device resources and decide access policies, but he cannot violate the integrity of the language or the device. Wirth [63,64] has recently made a similar proposa with his device modules in MODULA. Since MODULA has been implemented, know the approach is feasible. We refer the interested reader to Wirth papers for detailed information on an implementation of this type of device organization.

4.7. Summary

This chapter has described our resource mechanism for defining process interaction in a multiprogramming language. As a starting point we took Hoare's monitor concept. We then extended it in several directions in order to improve its expressiveness as a language facility, without sacrificing the property of data integrity. We proposed four modifications to monitors:

1. an automatically invoked POST_OP procedure to complement the initialization system,
2. replacement of the condition system by the <u>await</u> primitive for program-controlled synchronization,
3. a <u>parallel</u> clause for permitting safe, simultaneous accesses t a resource, and
4. a <u>release</u> clause for permitting nested monitors to overlap execution.

Most of this chapter dealt with the basic mechanics of these features--what they mean, how they can be used, and what restrictions are placed on them. The examples in this chapter were intended to convey a general understanding of the mechanism; in Chapter 7 we attempt to demonstrate the expressiveness of resources through a series of detailed examples.

This chapter makes two general contributions to the area of process interaction. First, it introduces several basic extensions to the monitor concept which give it more flexibility as a programming tool. Second, it carefully considers the enforcement of data integrity given these extensions. We show that it is possible for the language to insure data integrity, and we detail the constraints and run-time checks which are necessary to accomplish it.

CHAPTER 5
STATIC ACCESS CONTROLS

Our other area of interest in language design is mechanisms for providing access control. We see two levels of control as being important in operating systems. The first level is based on insuring data integrity. As we indicated in the introduction, the language should guarantee that simultaneous reading is the only allowable concurrent activity on a data object. Obviously, a language's access control mechanisms must play a key role in this endeavor. The second level of access control concerns the amount of flexibility given to a system designer. In Chapter 2 we enumerated several types of programming problems where it was useful to limit a block's access rights further than required by data integrity. Hence, we also need language mechanisms which will permit a programmer to enforce his own access policy.

This chapter defines our language features for providing static access control. By static access, we mean any access to an object permitted on the basis of scope rules. If a block of code inherits the right to use an object defined outside its scope, it cannot lose that right during execution. That right is static. The two previous chapters have touched on most of the important scope rules needed for data integrity. We summarize them here, and mention some of their implications which may not be obvious. The rest of this chapter presents the <u>restrict</u> pseudo-statement, which permits selective tightening of the basic scope rules.

Every language object can be restricted to a subset of the blocks to which it would normally be accessible. It can be applied to individual operations on a resource, as well as the entire resource. The philosophy behind this feature is that an outer block should dictate which objects an inner block can see.

5.1. Basic Scope Rules

The basic scope rules define a very lenient static access policy which still insures our goal of data integrity. These rules are used as the defaults for the language. Only two major language constraints affect the basic scope rules: (1) each data variable (i.e., Boolean, integer, array) must be permanently accessible to only one program unit (either a process or resource instance), and (2) only the process defining new process blocks may create and control instances of that new process type. Any type of static access which does not violate either constraint is permitted under the default scope rules.

The maximum scope of access for any statically-controlled object is the range of the block that defines that object. (The range of a block is the set of all statements inside it, including those in all nested blocks.) At a minimum, each object can be accessed within the reach of the block that defines it. (The reach of a block is the set of statements in that block, but not contained in any inner block.) The ability of an inner block to access an object is a function of the block and object types. Figure 5.1 summarizes the various possibilities using a matrix. Each row represents a particular object type and each column represents a block type. A "yes" entry for position (OBJECT,BLOCK) indicates that the basic scope rules permit an object of type OBJECT to be accessible to inner

		BLOCK TYPES		
		Procedure	Process	Resource
OBJECT TYPES	Data variables (Boolean, array, etc.)	Yes[1]	No[3]	No[3]
	Procedure declarations	Yes	Yes[2]	Yes[2]
	Static resources	Yes	Yes	Yes
	Resource type dcls,	Yes	Yes	Yes
	Process names	Yes	No[4]	No[4]
	Process type dcls.	Yes	No[4]	No[4]

[1]allowed if all calls to the procedure are from the reach of the process in which the procedure is declared.

[2]allowed if the procedure is pure.

[3]this restriction is needed to avoid simultaneous data modification.

[4]this restriction is needed for process control.

Figure 5.1: Default scope rules.

blocks of type BLOCK. In the case of nested blocks, the scope rules must be applied to each block separately. If a block is prohibited access to a globally-defined object, the restriction applies to the entire range of that block.

Some examples of static access are given in Figure 5.2 to help clarify the meaning of the scope rules matrix. Part (a) of the figure outlines a program in which all of the given accesses are legal. At the bottom of the figure, justification for allowing each access is based on the matrix of Figure 5.1. These accesses are all simple and straightforward.

In Figure 5.2b, all of the accesses violate the scope rules. Cases 1 and 2 combine to form one actual violation. The potential problem there is simultaneous modification of A by SYSTEM and GAMMA. SYSTEM declares A, so it has direct access to it. Also, GAMMA can call BETA which could call ALPHA. As a result, GAMMA could modify A. It is sufficient in this case to prevent either access 1 or 2. (An implementation would probably catch the second one.) Case 3 violates the nesting rule. Since GAMMA cannot access B, no internal block to GAMMA can access B.

The final case in Figure 5.2b is the most subtle. The name "GAMMA" is defined at the SYSTEM block level. It happens to stand for a block of type process. The applicable scope rule is that a process name cannot pass through a process block. The reason for this rule is to enforce the policy mentioned earlier that only the process defining new process blocks may create and control instances of it. In this case, only SYSTEM may control the process GAMMA.

```
                SYSTEM: process;
                        var A,B: integer;

                        ALPHA: procedure (COUNT: integer);
                                   :
1 -                                A := A+1;
                                   :
                                   end;

                        BETA: static resource;
                                   operations IN;
                                   :
                                   end;

                        GAMMA: process;
                                   :
2 -                                BETA.IN;
                                   end;
                     begin
3 -                     ALPHA(A);
4 -                     GAMMA. setstate (   );
                        :
5 -                     BETA.IN;
                        :
                        end

                     end SYSTEM
```

All of the static accesses in this program are legal, based on the default scope rules defined in the matrix of Figure 5.1.

1 - (DATA VARIABLE, PROCEDURE) = Yes, constraint holds;
2 - (STATIC RESOURCE, PROCESS) = Yes;
3 -
4 - access is within the reach of the block where declaration was made.
5 -

Figure 5.2a: Program using legal accesses based on scope rules.

```
          SYSTEM: process;
                  var A,B;

              ALPHA: procedure (COUNT: integer);
                        :
1 -                     A := A+1;
                        :
                        end;

              BETA: static resource;
                        operations IN;
                            :
2 -                         ALPHA(K);
                            :
                        end;

              GAMMA: process;
                            :
                            DELTA: procedure;
                                      :
3 -                                   B := B+1;
                                      end;
                            :
4 -                     GAMMA. setstate (   );
                        :
                        end;
              :
              end SYSTEM
```

The four accesses identified are all illegal, based on the matrix in Figure 5.1.

1 - condition violated on (DATA VARIABLES, PROCEDURE) call in BETA to ALPHA is not in reach of SYSTEM.

2 - condition violated on (PROCEDURE DCL, RESOURCE) [same as 1].

3 - nesting rule causes violation of (DATA VARIABLE, PROCESS).

4 - (PROCESS NAME, PROCESS) = NO.

Figure 5.2b: Program with violations of scope rules.

5.2. Program-controlled Access Restrictions

The scope rules as given are probably too permissive for most appli cations. This point has been raised as a criticism of Algol-like scope rules in general [56]. In our view of a computing facility, where the user and operating system are considered together as one program, some extra controls are necessary. The organization outlined in Figure 5.3 would normally permit USER blocks to access resource objects that are intended for use strictly by the operating system.

Our solution to the need for improved control is the restrict command, which controls access to an object on a block by block basis. The general form of the command is:

restrict <object list> to <block list>

The meaning of the statement is that only those blocks named in the bloc list may access the objects specified in the object list. Any staticall controlled object can be named in a restrict command. The elements of block list must be blocks defined at the block level where the restrict command appears. The only exception to this rule is that the name of t current block may be included in the block list. The presence of this name in the list permits the reach of that block to access the object. Restrict commands may only appear in the declaration portion of a block By definition, this command can only be used to restrict the default sc rules, it cannot be used to permit an access that would otherwise be illegal.

We can demonstrate the use of the restrict command by completing solution to the readers/writers problem of the previous chapter. The solution in Figure 4.9 had the drawback that readers and writers could

```
                SYSTEM: program;

                   CONSOLE: static resource;
                                operations READ,WRITE;
                                    :
                                end CONSOLE;
System
resources
                   BUFFER_ALLOCATOR: static resource;
                                        operations REQUEST,RELEASE;
                                            :
                                        end BUFFER_ALLOCATOR;

                   SPOOLER: process;
                                :
                                CONSOLE.WRITE(   );
System                          :
processes                       BUFFER_ALLOCATOR.REQUEST;
                                :
                                end SPOOLER;
                   :
                   USER_i: process;
                                :
Users                           end USER_i;
                   :
                   end SYSTEM
```

Figure 5.3: Hypothetical system structure.

potentially by-pass the scheduler and call DATA_BASE directly. The solution outlined in Figure 5.4 eliminates this weakness. The *restrict* command has the dual effect of permitting SCHEDULE to access the DATA_BASE resource and preventing the USER blocks from seeing its definition. Also, since SYSTEM is not in the block list of the *restrict* command, the reach of SYSTEM cannot access DATA_BASE even though that block declared the object.

The *restrict* command can be applied to all object types, not just resources. It can be used to control each block's access rights to variables and procedures in order to improve the security of the system. Control of type definitions can also be very useful. For example, an operating system may require several identical terminal-controller resources which are accessible to users. The simplest way to achieve this structure is to define one resource type called TERMINAL_CONTROLLER and then create several instances. Figure 5.5 demonstrates our solution. The first restrict command prohibits the users from using the type name (since they are not in the block list), so they cannot declare new instances of the resource. The command does not affect the instances defined by SYSTEM, so the users can still access them.

An extension to the basic *restrict* command permits control over individual resource operations. The extension has the following form:

restrict R_NAME{OPERATION_LIST} *to* <block_list>

When an operation list is specified as part of resource object name, the blocks in the list may access the resource but only through the operations named. Obviously, the list can only include operations which are defined for the resource.

```
SYSTEM: program;

    var DATA_BASE: static resource;
                   operations READ,WRITE;
                   :
                   end DATA_BASE;

    restrict DATA_BASE to SCHEDULE;

    var SCHEDULE: static resource;
                  :
                  end SCHEDULE;

        USER_1: process;
                :
                end USER_1;

        USER_2: process;
                :
                end USER_2;
    :
    end SYSTEM
```

Figure 5.4: Restrict used in solution to readers/writers problem.

```
SYSTEM: program;

    type TERMINAL_CONTROLLER = static resource (DEVICE: ID);
                                   operations READ,WRITE;
                                   :
                                   end TERMINAL_CONTROLLER;

    restrict TERMINAL_CONTROLLER to SYSTEM;

    var TERM_1: TERMINAL_CONTROLLER(CONSOLE);
        TERM_2: TERMINAL_CONTROLLER(REMOTE_LINE);

    restrict TERM_1 to USER_1;
    restrict TERM_2 to USER_2;

        USER_1: process;
                :
                TERM_1.READ;
                :
                end USER_1;

        USER_2: process;
                :
                TERM_2.WRITE;
                :
                end USER_2;
        :
        end SYSTEM
```

Figure 5.5: Using restrict to control access to a type definition.

We can use this extension to enforce some normally difficult accessing policies. Figure 5.6a sketches one hypothetical situation. Four processes communicate with each other by passing messages. The access policy we want to enforce is represented by the arcs connecting the processes. For example, USER_1 can send messages to USER_2, but he can only receive them from USER_3 and USER_4. Our solution to the problem is shown in Figure 5.6b. The message resource (defined in Chapter 4) accepts messages from any process calling SEND and transmits them to any process calling RECEIVE. We use three instances of the resource in our solution. The access policy is enforced via three <u>restrict</u> commands, giving each process only those operations he needs.

The use of multiple restrict clauses in this example requires some clarification. In the default case (i.e., no <u>restricts</u> given for an object) access is granted to the object if it does not violate the scope rules. If one restrict is applied to an object, all blocks named in the block list are permitted access and all others are prohibited from using it. When multiple <u>restricts</u> are applied to the same object (e.g., all three LINKS in Figure 5.6), each access that is identified for a block is allowed. In this case, a block is prohibited access only if it is not specified in any of the clauses. For example, USER_1 cannot use LINK_2 at all, and it cannot RECEIVE on LINK_1 or SENO on LINK_3. The main point here is that in the single restrict case the clause specifically grants access and denies it to all unnamed blocks. In the multiple restrict case acceptance/denial is based on the union of rights in the clauses.

The examples and description of <u>restrict</u> have, so far, been limited to situations where it was used immediately after an object was declared.

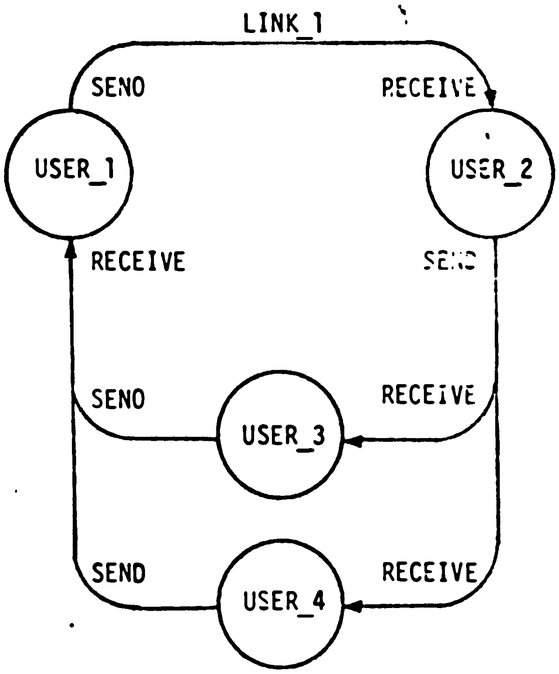


Figure 5.6a: A process communication policy.

```
SYSTEM: program;

   type MESSAGE_CLASS = static resource (SIZE: integer);
                        operations SEND,RECEIVE;
                        (body defined in Figure 4.4).
                        end;

   var LINK_1,LINK_2,LINK_3: MESSAGE_CLASS(20);

   restrict LINK_1(SEND),LINK_3(RECEIVE) to USER_1;
   restrict LINK_2(SEND),LINK_1(RECEIVE) to USER_2;
   restrict LINK_3(SEND),LINK_2(RECEIVE) to USER_3,USER_4;

   var USER_1: process;
                    :
                    LINK_1.SEND(   );
                    :
                    LINK_3.RECEIVE(   );
                    :
                  end USER_1;

       (similar code for other users)
   :
   end SYSTEM
```

Figure 5.6b: Using restrict to enforce communication policy.

It is also possible to use a restrict command inside blocks which have been granted access to a globally defined object. For example, an operating system may give a user the right to access some resource. However, within that process, the user may only want certain trusted portions of the code to access the resource. In this case the USER can enforce his own internal access policy with restrict just as if he had declared the object. The obvious constraint is that a block must have been granted access (explicitly or by default) to an object before it can control it.

A subtle question with respect to restrict can arise when name redefinition is permitted (as it is in Pascal). Consider the example in Figure 5.7, where two nested blocks, SYSTEM and USER, define similar resources called BUFFER. Inside USER, a restrict clause is used to prohibit SUB_TASK from accessing the nearer definition of BUFFER. The relevant question is--if SUB_TASK is blinded from the nearer definition, can it still access the outer one? If we actually view the restrict mechanism as hiding particular definitions from a block, then the answer should be that SUB_TASK can still access the outer definition of BUFFER. We have chosen to look at name redefinition in a different light, resulting in different answer. If a block A redefines some name B, the new definition of B is the only one that can possibly be used in the range of A. In effect, the new definition blocks out the outer one from ever being seen anywhere in A. So our answer to the problem posed in Figure 5.7 is that SUB_TASK cannot access either definition of BUFFER.

We have taken this approach for several reasons. First, it is a simple, understandable option. Second, it provides some protection against inadvertent accessing of objects. In the situation described

```
SYSTEM: program;

    var BUFFER: static resource;
                  operations ADD,REMOVE;
                  :
                  end BUFFER;

        USER: process;

            var BUFFER: static resource;
                          operations ADD,REMOVE;
                          :
                          end BUFFER;

            restrict BUFFER to USER;

            var SUB_TASK: process;
                          :
Is this legal? →          BUFFER.ADD(    );
                          :
                          end SUB_TASK;

            :
            end USER;
        :
        end SYSTEM
```

Figure 5.7: A name redefinition problem for restrict.

Figure 5.7, it is likely that USER forgot to include SUB_TASK on the list of blocks to get buffer. If SUB_TASK can use a system-defined one instead, no error will be detected until execution fouls up in some way. But the main reason for our choice is that we could see no interesting applications that would justify the added complexity that other options required. In most cases, the easiest alternative is to avoid name redefinition in the first place.

5.3. Summary

Our approach to static access control is based on two general goals: (1) permitting the clear expression of static access policies, and (2) enforcing them without incurring any run-time overhead. The default scope rules provide the most lenient access environment possible without allowing violations of data integrity. Static access policies for some particular program can be introduced on top of these constraints through the restrict pseudo-statement.

By the nature of block-structured languages, static access policies have one basic form--limiting the access of inner blocks to objects defined at outer levels. Unless other control tools are available, this situation makes inner blocks more powerful than outer ones. Unfortunately in operating systems, we almost certainly do not want this type of power structure. If some block defines an object, it should be able to control which other blocks can use it. These are the types of static policies we want to be able to express.

The major effect of the restrict feature is to allow a designer to reverse the default power structure of blocks. Each outer block can dictate to each immediate inner block precisely which objects that block may use. This same control carried through any number of levels of nesting; a block may be granted access to some global object and choose not to let a further nested block use it. As a result, it is a simple, yet powerful mechanism for expressing access policies. And, since its effect is limited to modifying access rights determined by scope rules, it can be implemented without any run-time checking.

CHAPTER 6

DYNAMIC ACCESS CONTROLS

The main weakness of static access control systems is that they cannot by themselves accurately describe some commonly used access restrictions. For example, a system designer may want his operating syste to be able to allocate files to users on a temporary basis. We want to represent the fact that a user can access the file only during the time between its request and release. With static access controls, if a bloc (or process) has any access to an object, it has that access permanentl Clearly, to get the desired level of protection, we need a dynamic set o access controls.

This chapter presents our language facilities for specifying dynar access control over objects. To avoid confusion between static and dynamic controls, we introduce two new types of language objects--*dynami resources* and *protected* variables; dynamic controls *only* apply to instances of these two types. Access to a *dynamic resources* is made throu a *capability* variable [24] which holds a reference for that object and specifies authorized accesses. Capabilities can be passed as parameters thus allowing *dynamic resources* to move through process environments. I controlling *protected* variables (simple data objects) we define a simila but more restrictive system using *pointer* variables to hold their references. These facilities for dynamic access control give our language descriptive power that is not available in any existing languages.

6.1. Dynamic Resources and Capabilities

The purpose of this section is to describe a simple system for allowing resources to be dynamically accessible to various process environments. We introduce a new type of resource (dynamic) which must be accessed indirectly through a capability variable. By defining operations like assignment and parameter passing on capabilities, we are in effect, defining the mechanisms for allowing dynamic movement of resources.

6.1.1. Dynamic Resources

Our plan for presenting dynamic resources and capabilities is to solve a dynamic control problem while we introduce the various aspects of the features. By interleaving language details with a problem solution, the description of the features should be easier to follow. The specific problem, taken from [57], is to design a system for allocating a fixed number of message channels to various users on a temporary basis. We assume that many more channels will be requested during execution than are actually in use at any one time, so dynamic control of the channels makes sense. The structure of our solution is displayed in Figure 6.1. Each channel is an instance of a dynamic resource called MESSAGE_CHANNEL. A static resource, ALLOCATOR, is responsible for granting users access to a channel. To keep this example simple and to focus on the mechanics of dynamic access control, we assume that users will not attempt to disrupt the system by taking devious actions. (In the next chapter we will deal with the more difficult problem of untrustworthy users.) As we describe the parts of dynamic resource control, we will fill in the blocks of code to get a complete solution.

```
SYSTEM: program
        :
        type MESSAGE_CHANNEL = dynamic resource;
                               operations SEND,RECEIVE;

                               Each instance of this resource can hold one
                               message (of type MESSAGE) at a time.  SEND
                               stores it, and RECEIVE removes it.

                               end MESSAGE_CHANNEL;

        ALLOCATOR: static resource;
                   operations ACQUIRE,RELEASE;

                   This resource holds capabilities for all instances of
                   MESSAGE_CHANNEL.  A user may ACQUIRE one by specifying
                   a channel number.  RELEASE gives the capability back
                   to ALLOCATOR.

                   end ALLOCATOR;

        USER_K: process;
                :
                end USER_K;
        :
        end SYSTEM
```

Figure 6.1: Program skeleton for allocating message channels.

The simplest part of the dynamic system to describe is the definition of dynamic resources. All resources, whether static or dynamic, must obey the same rules regarding their internal structure. The accessing method does not affect how a resource may operate. For example, all static resource definitions made in the previous chapters could be redefined with the dynamic option. (Of course, we would have to change the scheme for using the resource, but it would not affect the resource definition itself.) The only real difference between dynamic and static resources is that all dynamic ones must be declared as types. The purpose of this rule is to separate the specification of a dynamic resource block from the creation of any of its instances. (We discuss dynamic resource creation later.) The type definitions for dynamic resources obey the same scope rules that applied to static resources (e.g., they are visible through process and resource block definitions). Similarly, we prohibit dynamic resource types from being passed as parameters, just as all other passing of types is prohibited.

We can now specify the first part of our solution to the message channel allocation problem. In Figure 6.2 we define the dynamic resource type, MESSAGE_CHANNEL. For simplicity, each channel holds at most one message at a time. The actual definition is similar to other message systems presented earlier so we will not discuss it further here. The important point is that the type definition will be visible inside the USER and ALLOCATOR blocks.

6.1.2. Capability Variables

The key to the entire dynamic access control system is the use of capability variables to hold references to dynamic objects. A capability

```
type MESSAGE_CHANNEL = dynamic resource;
                       operations SEND,RECEIVE;

                       var MESS: MESSAGE;
                           MESS_EMPTY: Boolean;

                       SEND: procedure (INFO: MESSAGE);
                             await MESS_EMPTY;
                             MESS := INFO;
                             MESS_EMPTY := false
                             end SEND;

                       RECEIVE: procedure (INFO: MESSAGE);
                                await ¬ MESS_EMPTY;
                                INFO := MESS;
                                MESS_EMPTY := true
                                end RECEIVE;

                       PRE_OP:  procedure;
                                MESS_EMPTY := true
                                end PRE_OP

                       end MESSAGE_CHANNEL
```

Figure 6.2: Definition of the MESSAGE_CHANNEL resource.

variable is declared using the notation:

<cap_name>: <dynamic resource type> capability

Each capability variable is only permitted to contain a reference for one type of dynamic resource (namely, the type specified in the declaration). In addition to a dynamic resource reference, a capability variable contains access rights for that resource instance. Among the possible rights a capability may have are the rights to call the resource's operations (each one separately controlled). A dynamic resource may be accessed by using a capability which references it. The notation is:

<cap_name>.<op_name>(<parameter list>)

When this statement type is executed, the operation is invoked for the resource currently referenced by the capability. If the capability does not contain a reference or does not contain the proper access rights when the call is attempted, it is an error.

It is important to distinguish between a capability variable and the object it may reference. During execution a capability variable may reference different instances at different times. Also, several capability variables may reference the same object at the same time. Moreover, while a dynamic resource is controlled through dynamic access mechanisms, capability variables are controlled by static rules. In fact, the static scope rules for capability variables are exactly the same as the ones for data variables. As a result, a capability variable can only be accessed in the process or resource block that defined it. (The reason for these tight scope rules on capabilities is to prevent any implicit sharing of dynamic resources among processes. All movement of access rights among processes will be accomplished by parameter passing.)

We can now show how a USER can access and use an instance of a MESSAGE_CHANNEL. Figure 6.3 outlines a possible user structure. At the beginning of the process, the user declares a capability, CHANNEL, which can hold a reference to a MESSAGE_CHANNEL instance. Initially this capability is empty. When the user calls the allocator, he acquires an instance of the dynamic resource in CHANNEL. The user can then access the SEND and RECEIVE operations through his capability name. When the user is finished, he calls the allocator to release his access to the instance. Notice that if the user attempts to use the CHANNEL before acquiring an instance, an error will occur.

Creation of a dynamic resource requires three steps. First, the definition of the dynamic resource's type must be given. Then a capability must be declared to hold a reference to that type of resource. Both of these steps have already been covered. The final step is to execute a statement of the following form:

<cap_name> := <dynamic_resource_type>. create (<params>)

The effect of this statement is to create a new instance of the specified type of dynamic resource and give the capability a reference to it, along with full access rights. The access rights permit the capability to be used to call any operation for the resource. In addition, the capability gets the COPY right (which will be explained momentarily). For simplicity, we impose the restriction that a capability used in a create statement must be empty prior to the creation. The actual parameters to a create depend on the resource's type declaration; if it specifies formal parameters, then the corresponding actual parameters must appear in the create

```
USER_K: process;
        var CHANNEL: MESSAGE_CHANNEL capability;
            MESS: MESSAGE;
            REQUEST_NUMBER: integer;
            :

        do forever
            :
            /* Get a channel capability by specifying its number */
                ALLOCATOR.ACQUIRE(REQUEST_NUMBER,CHANNEL);
            :
            /* Access the channel by using the capability */
                CHANNEL.SEND(MESS);
                :
                CHANNEL.RECEIVE(MESS);
                :
            /* Return the channel by passing back the capability */
                ALLOCATOR.RELEASE(CHANNEL);

            :
            end

        end USER_K
```

Figure 6.3: A sample user's code for accessing a message channel.

statement. For the moment, only data variables may be passed, and they are passed by value. (Shortly we will extend this rule to permit capabilities to be passed.)

Movement of access rights for a dynamic resource instance is accomplished by assignment and by passing parameters. The assignment statement takes the following form:

<cap_name_1> := <cap_name_2>{<rights list>}

Execution of this type of statement copies the reference held in the right-hand side capability to the one named on the left-hand side. Further, only those rights named in the rights list are given to cap_name_1. We place the following three constraints on the use of the assignment:

1. cap_name_1 must be empty prior to the assignment,[1]
2. cap_name_2 must have the "COPY" right in its possession, and
3. cap_name_2 must possess all rights named in the rights list.

The assignment statement does not affect the right-hand side capability.

We also use the assignment statement to empty a capability of its current resource reference. This option is effected by assigning the keyword <u>null</u> to the target capability. No special right is necessary to take this action.

Although the assignment statement permits the specification of multiple access paths to a resource, the scope rules for capabilities prevent two processes from sharing those paths. The only way that resource sharing among processes can be accomplished is through parameter passing. Passing a capability on a procedure call (or resource call) has

---

[1]This constraint is not absolutely necessary; its presence avoids the inadvertent loss of access rights.

the effect of assigning all rights contained in the actual parameter to the formal one, and then making a null assignment to the actual parameter. When the call returns, the reverse operation is carried out (i.e., assignment from the formal to the actual parameter). This formulation of the rules for rights movement makes it very simple for a process to either share or transfer his access rights to another process.

We can now complete our solution to the message allocator problem. In this example, the static resource, ALLOCATOR, creates and controls all instances of the message channels. When ALLOCATOR is initialized, all instances are created at once, with tne references stored in capabilities of its global data. When a process requests a message channel, ALLOCATOR gives it a copy of the appropriate capability with the necessary rights (always keeping a copy for himself). When a process returns a capability, the ALLOCATOR nullifies the formal prameter to take away the user's rights.

The complete coding for the allocator is shown in Figure 6.4. We have selected a simple convention for permitting a group of users to share the same message channel. When a user requests a message channel, he must specify some channel number. All users giving the same number are granted access to the same message channel.

We must reiterate that in this example the users are assumed to be trustworthy. Among other things, they will not create their own instances of message channels (and pass them to the allocator) and they will not return any empty capabilities in a RELEASE call. We have made these assumptions only to simplify the presentation of this example. With some extra coding we could remove these assumptions and still have a safe

```
var ALLOCATOR: static resource;
               operations ACQUIRE,RELEASE;

               var CHANNELS: array 1..N of MESSAGE_CHANNEL capability;

               ACQUIRE: procedure (INOEX: integer,
                                   MC: MESSAGE_CHANNEL capability);

                       MC := CHANNELS(INOEX){SEND,RECEIVE}

                       end ACQUIRE;

               RELEASE: procedure (MC: MESSAGE_CHANNEL capability);

                       MC := null              /* empty user's rights */
                       end RELEASE;

               PRE_OP: procedure;
                       var I: integer;
                       for I := 1 to N do
                               CHANNELS(I) := MESSAGE_CHANNEL. create
                       end PRE_OP

               end ALLOCATOR
```

Figure 6.4: Resource for allocating message channels.

system. In fact, examples in Chapter 7 will take exactly this point of view. Our purpose now is to convey the use of the dynamic resources and capabilities in the simplest possible manner.

The remaining points to be made about dynamic resources and capabilities involve areas that were not necessary to solve the previous example. We define two Boolean operations on capabilities so that their contents can be interrogated without actually using a specific access right. We also extend parameter passing to permit capabilities to be passed to processes or resources during their creation. And finally, we discuss the destruction of dynamic resource instances.

Capabilities may appear in Boolean expressions in two possible ways. The equality operation can be used to determine if two capabilities currently reference the same object. The access rights of each capability are irrelevant in this evaluation. For example, if MC_1 and MC-2 reference the same dynamic resource (even if MC_1 has SEND and MC_2 has RECEIVE rights), the Boolean expression:

(MC_1 = MC_2)

evaluates to true. The only other relational operator for capabilities is an extension of the PASCAL set containment operator, <u>in</u>. The Boolean expression:

({<rights list>} <u>in</u> CAP_NAME)

evaluates to true if and only if the capability contains a reference to a resource and it possesses all access rights named in the list. In the message channel allocation solution, we could have used these operations (mainly the equality) to insure that a user was actually returning a reference to one of the channels.

The other extension we make is to permit capabilities to be passed as parameters when processes and resources are created. Previously, only data variables could be passed (by value) in this situation. This extension allows a process (or resource) to acquire some dynamic object rights without going to a static resource later during execution. The only difference between passing during creation and passing during a procedure call, is that in the former case there is no "return phase" where capabilities may come back. Any time a capability is passed during a creation, it is guaranteed to be empty the next time it is used.

The major remaining question about dynamic resources that we have yet to answer is when and how can an instance be destroyed? Our convention is that an instance can be destroyed only when no capabilities reference it. The life-time of dynamic resources is therefore different than the one normally defined by static scope rules. It is possible for dynamic object to outlive the process (or resource) which created it. T[illegible] only limitation (caused by scope rules) is that an instance cannot outli[illegible] its type definition. This statement is obvious from the fact that any capability holding the instance must be declared using the type name, so the capability's scope cannot be greater than that of the type.

6.1.3. Implementation

Incorporating dynamic resources and capabilities in a language ad[illegible] some implementation cost. It naturally requires more work than the sta[illegible] case, but this extra work can be kept at a reasonable level. A capabil[illegible] variable could be represented as a pair (pointer, bit vector) where the pointer is an address for the resource, and the bit vector holds the access rights. Operations on capabilities, like assignment and parameter passing, are then completely straightforward. Using a capability to access a resource requires three steps: (1) verifying that the pointer contains a valid reference, (2) testing the bit vector to see if the operation is permitted, and (3) invoking the procedure call. With most existing hardware, steps one and three can be done with little trouble. The only unavoidable overhead is the bit vector test. But in comparison to the work necessary for a procedure call, this extra test is not significant.

We can recognize when a resource should be destroyed by associating a simple reference counter with each one. Any time capability operations take place, the appropriate counter is adjusted accordingly. The only difficulty would be caused by a resource which held a reference to itself. It may not be accessible to any object, yet the reference count will be non-zero. A special test can be added by the compiler to cover this situation in the event that a program makes it possible. Also, if recovery of space is not necessary, the reference count can be omitted.

The one implementation problem we cannot ignore is the requirement for dynamic memory allocation. Since a dynamic object may survive the process or resource which created it, simple stack allocations of memory will not be sufficient. Separate areas of memory must be available when an object is created at run-time. Likewise, when an object is destroyed the space allocated for it should be recoverable. This recovery process requires some general run-time garbage collection scheme which introduces additional overhead.

One possible solution to this problem is to modify the language, removing the facilities for dynamic creation and destruction. In our design we have tied allocation to access control. In other words, every dynamically controlled object must also be created dynamically. This connection is not absolutely necessary. (We made this choice to conceptually separate the static and dynamic system, and also to examine the potential gains in descriptive power for the language.) It would be possible to design a similar system with standard block allocation, but using dynamic access control. We only need to insure that a resource is not deallocated while some block has access to it. With this approach the problem of memory management is decreased considerably.

An alternative solution to having dynamic memory management would be to specify an upper limit on the number of instances for each dynamic resource type. This could be done by having some kind of limit phrase as a required portion of the type definitions. In this way, the maximum space requirements can be computed and allocated at compile-time. (However, this approach may involve more memory than is actually necessary.)

To briefly summarize, our basic system for providing dynamic access control introduces the concept of capabilities into the language. The logical object for this type of control is a resource, since we designed it for handling interactions. Each program unit (process or procedure) declares its own capability variables which must be used in accessing instances of dynamic resources. The movement of rights between different environments is accomplished by passing capabilities as parameters.

Run-time checking for this system only involves one inexpensive rights test prior to each invocation of a dynamic resource operation.

We feel that this approach provides a simple, yet powerful environment for writing programs that require dynamic access control. In the next chapter we show how these mechanisms meet nearly all our expressiveness goals.

5.2. Protected Variables and Pointers

For most designs employing dynamically controlled process interaction we feel that dynamic resources and capabilities can provide a simple, concise problem solution. However, in some situations these tools may force a system designer to incur unnecessary overhead. For example, what if a dynamic object may be used by several processes, but at any given time only one process (or resource) may have access rights to that object? Clearly, in this situation the synchronization overhead of resources is not needed. We could go one step further and note that in some cases even the overhead of the calling resource operations may be unnecessary. As a specific example, consider the problem of allocating buffer space to several processes. While doing I/O, a process may need space for temporary storage. If he only needs it for a short time, it would be grossly inefficient to make him define and keep a permanent block of space. A useful alternative is to dynamically allocate space from a fixed pool. If we use resources to represent buffers, synchronization is unnecessary since only one process at a time has access rights. Moreover, the procedure calls are wasted effort since the only operations we need are direct reads and writes of buffers (via assignment statements).

For these situations, we introduce a new type of dynamic access control system (using protected variables and pointers) which avoids th unnecessary overhead. In this section we define operations that are c plementary to those in the capability system, but in this case the con trolled objects are data variables (e.g., integers, arrays, and record instead of resources. To avoid having a designer misuse these access facilities (by giving a data object to two processes simultaneously), language mechanisms insure that only one reference ever exists for eac dynamically controlled data variable. Moreover, that one reference ca be shared by two processes.

Most of the system for permitting dynamic control of data objec is analogous to that used in the capability system. A dynamically co trolled data object (a protected object) must be specified through a declaration, using a new attribute called protected. An example of u this attribute follows.

```
type BUFFER = protected array 1..N of integer
```

This attribute can be added to any type of standard data variable. (PASCAL supported the CLASS [9,16] concept we could also allow it to b protected.) The object corresponding to a capability is a pointer. holds a reference for a protected object, and is declared as follows

```
var BUF_1: BUFFER pointer;
```

Once the protected type and an appropriate pointer are declared, an instance of the protected data object can be created via:

```
BUF_1 := BUFFER. create;
```

After this statement is executed, the pointer BUF_1 contains a reference to a new data object of type BUFFER.

One of the major differences between capabilities and pointers involve when and how they can be used to access the objects they reference. When a capability's object is accessed, it must be for calling a resource operation; hence the call always stands as an independent statement. It is always clear from the statement context how a capability is being used. With pointers we do not always have such a clean separation. To avoid ambiguity, we use the convention that a pointer followed by an up-arrow gives access to the object. So for example if we are passing a parameter, BUF_1 means pass the pointer, and BUF_1↑ means pass the values stored in the array referenced by the pointer. Using this convention, a pointer followed by up-arrow may appear in any expression where a variable of the basic object type could appear. If a pointer references an array, specific entries in the array are accessed via the standard subscripting mechanism. For example,

```
BUF_1↑(I) := BUF_1↑(I) + 1;
```

increments by one the value of the I^{th} entry in the array referenced by BUF_1. Similarly, if the object is a record definition, record fields are accessed by specifying the field name after the up-arrow.

The other major difference between capabilities and pointers involves movement of access rights. With pointers the language definition insures that only one reference to a protected object ever exists. As a result, the assignment operation for pointers must destroy the reference in the right-hand side operator in order to give it to the left-hand side pointer. Rather than use the normal assignment command which may cause confusion on this point, we introduce a different symbol for pointer assignment. The statement:

BUF_2 ← BUF_1;

transfers the reference rights contained in BUF_1 to BUF_2. By definition, BUF_1 is always empty after the transfer. We make no provisions for transferring sub-rights of a protected data object since it would imply that the non-transferred rights would be permanently lost. Hence, any pointer reference automatically has full rights.

The <u>null</u> assignment can also be used with pointers, but in this case it is much stronger. If a pointer is nullified, the object is effectively being destroyed. To accurately represent this action we use the key word <u>destroy</u>, which only operates on pointers. The assignment:

BUF_1 ← <u>destroy</u>;

destroys the protected object referenced by BUF_1 and also empties the pointer variable. (The only other way a protected object can be destroyed is if an active pointer variable dies because execution of the block in which it was declared, is terminated.)

In parameter passing, pointers and capabilities follow exactly the same rules. On calls to procedures or resources a pointer reference is transferred from the actual to the formal parameter. On return the transfer is in the reverse direction. Pointers may also be passed at the creation of processes and resources. As with capabilities, the transfers in this case only go from the actuals to the formals.

The last area of the pointer system we must detail is the possibility for interrogation of a pointer to determine its status. With capabilities we could ask many questions because of the possibility for multiple object references and different access rights. For a pointer, the only relevant question is--does it currently contain a reference? We define a Boolean function <u>active</u> to permit this interrogation. The expression "<u>active</u> (BUF_1)" evaluates to true if and only if BUF_1 contains a reference to a protected object.

We now illustrate the use of the protected data object system with an outline of a solution to the airline reservation problem. In this problem several airline clerks need to access the records for future plane flights. The access may only be to confirm a previous reservation (no writing), or it may be to make a new one. However, to insure the integrity of the flight information, the airline company insists that only one clerk at a time be able to examine and change the records of a particular flight. Our solution to the problem is shown in Figure 6.5. Each flight record is implemented as a separate instance of a protected record type FLIGHT_STATS. The FLT_MANAGER resource is responsible for holding all records that are not being used by a clerk. On a request or release, the manager uses the pointer assignment to move a record to or from his FLT_BOOK. Notice that in this solution, unlike our earlier message channel allocation solution, the FLT_MANAGER does not keep a reference to a record being used by a CLERK. (Even if the manager were coded improperly, the language prevents a particular protected record from having more than one reference.) The RELEASE procedure demonstrates the use of the <u>active</u> feature; in this case it verifies that a clerk is actually returning a flight record.

```
SYSTEM: program;

    type FLIGHT_STATS = protected record
                        FLT_NUM: integer;
                        ORIGIN,DEST: CITY_CODE;
                        OPEN_SEATS: integer ;
                        end

    FLIGHT_MANAGER: static resource;
                    operations REQUEST,RELEASE;

                    var FLT_BOOK: array 1..MAX_FLTS of
                                          FLIGHT_STATS pointer;
                    :
                    REQUEST: procedure (FLIGHT: integer,
                                      FLIGHT_REC: FLIGHT_STATS pointer)

                          Search FLT_BOOK for appropriate FLIGHT.
                          Store entry index in variable IND.

                          FLIGHT_REC ← FLT_BOOK(IND)
                          end REQUEST;

                    RELEASE: procedure (FLT_REC: FLIGHT_STATS pointer);
                          :
                        /* Make sure a record was passed in */
                          if ¬ active (FLT_REC) then return;

                        Find an empty entry in FLT_BOOK.
                        Store index in variable IND.

                        FLT_BOOK(IND) ← FLT_REC
                        end RELEASE;
```

Figure 6.5: Flight reservation system (continued on next page).

```
                Initialize FLT_BOOK, etc.
                end FLT_MANAGER;

FLT_CLERK: process;
           var INFO: FLIGHT_STATS pointer;
           :
           while CUSTOMER_PRESENT do
                Get ticket request, including FROM (origin),
                TO (destination), and SEATS (number of tickets).
                Find the proper FLIGHT.

              /* Get the record for the appropriate flight */
                FLT_MANAGER.REQUEST(FLIGHT,INFO);

              /* If we have the proper flight, and there */
              /* is still room, then reserve the seats.  */

                if (TO = INFO↑DEST) &
                   (FROM = INFO↑ORIGIN) &
                   (SEATS <= INFO↑OPEN_SEATS) then
                   begin
                       INFO↑SEATS := INFO↑SEATS - SEATS;
                       CONFIRMED := true
                       end;

                FLT_MANAGER.RELEASE(INFO);
                :
                end
           end FLT_CLERK
end SYSTEM
```

Figure 6.5 (cont.): Flight reservation system.

The last part of our solution shows how a flight clerk might access the data stored in a flight record. He first makes a request to the flight manager to get access to a particular record. It is returned through the pointer parameter INFO. If the flight has sufficient room, the ticket request is noted by decreasing the number of open seats on the flight. The clerk then returns the flight record to the manager.

In this example we have assumed that the clerks are trustworthy (i.e., they will not modify the flight number or change the destination, etc.). If more protection is necessary, it could be achieved by changing the representation of a flight status from a <u>record</u> to a <u>class</u> definition (à la Concurrent Pascal). (We have not included <u>class</u> in our language, but it is a trivial extension.) This change would permit controlled use of the data by reintroducing the overhead of procedure calls. However, this solution still avoids the unnecessary synchronization required when using a resource for holding the data.

The implementation of protected variables and pointers is straightforward. Each variable only needs to be represented by the address of the referenced data object. The operations on pointers, such as <u>active</u> and parameter passing, are simple to handle. The only necessary run-time check verifies that a pointer actually contains a valid reference when it is used. Hardware address-checking available on most machines can make this test easy. The only drawback is the dynamic creation and destruction of protected objects, which usually implies recovery of memory. We covered this problem in our discussion of capabilities, and our comments apply equally here.

6.3. Discussion of Dynamic Access Control

The concept of dynamic access control has already been discussed in the literature, but mostly from the perspective of operating systems [1,4, 19,26,39,42,58]. Research has even included the development of hardware that supports capability-based access control [24,29]. But so far, very little has been done to bring this concept to realization in a programming language. To our knowledge the language facilities presented in this chapter are some of the first ones to provide the access-control expressiveness normally associated with the capability concept.

In designing these language facilities, two specific concerns influenced our work. Since this language is one of the first to incorporate general access controls, we felt it was important to conceptually separate, as much as possible, the static from the dynamic system. With a maximum separation, the resulting language becomes easier to use and understand. Also, it gives us a clear view of the effect that including dynamic access control can have on the overall language design. The other concern was to avoid imposing policy decisions on language users. In particular, the "ownership" concept was excluded from the design for this reason. Both of these positions had a strong influence on the final language design.

Our desire to separate static and dynamic controls led to several key design decisions. Dynamic (and protected) objects provide the focal point for the new access controls. As we stated earlier, we linked the object allocation mechanisms with the access control mechanisms. As a result, static resources cannot be passed as parameters, and capability variables cannot be shared by processes. Neither of these constraints is absolutely necessary for the safe execution of a program. The return we get from these design decisions is added language clarity. If some program uses a static resource, we can examine the program text and determine exactly which blocks can use it. If instead, the object is a dynamic resource, we know that the access rights must move by passing a parameter. Having an explicit representation of this movement can greatly simplify understanding of the program. Since our major goal is to define expressive facilities for systems design, we feel that this language clarity is important.

A noteworthy alternative to our scheme for putting capabilities in a language has been recently proposed by Jones and Liskov [40]. The major difference is that in their scheme each capability has a fixed set of access rights. Every time an object reference is put in a capability, the appropriate access rights must accompany it. The advantage of this approach is that all access rights checking can be done at compile-time through tests on all capability assignment and parameter passing statements. It eliminates the need for the run-time check of an access right that we encounter when a dynamic resource operation is invoked. The disadvantage of their approach is that it could require many more capability variables--one for each subset of rights that some process could receive.

The more difficult design decisions were affected by the concept of ownership, often included in discussions of dynamic access control [53,53]. The philosophy behind ownership is that one distinguished subject has special privileges with respect to each object. These privileges may include the ability to unilaterally revoke another subject's access rights, or to destroy the object (possibly while other subjects are trying to use it). The question for us is--do we include some form of the ownership concept in our design? If we reject it, a system designer may have difficulty trying to introduce it himself. On the other hand, if we include it we must make many policy decisions (like which rights are special, what action should be taken if a destroy is issued while a subject is using an object, etc.). Either of these decisions could handicap some designs.

It should be evident from our language specification that we have rejected the ownership principle. In this regard we maintain the same position advocated by the Hydra group [65]. Many protection situations are not accurately modelled under ownership, and forcing a designer to identify an owner when there really is none is wrong. The main effect of this decision on the language is that rights cannot be revoked from a process without its permission. A process gives permission for revoking a capability by passing it as a parameter. So long as an active capability is not passed (or nullified) it is certain to remain active. In example 3 of the next chapter we illustrate one way for a designer to introduce the ownership concept. The key is to write a resource which can, in effect, quit functioning. We cannot unilaterally take a process's capability away, but using this idea we can make his rights useless. This is, however, an expensive approach.

CHAPTER 7

APPLICATIONS OF RESOURCES AND ACCESS CONTROLS

The three previous chapters have presented each of the language features in isolation, using very simple examples to help convey their meaning. In this chapter we present four detailed examples in order to demonstrate their utility and expressive power on realistic examples.

The four programming problems in this chapter are all drawn from the area of operating systems design. Our solutions are based on a system structure that combines the user and system code into one program for execution (see Figure 1.1). We assume that the users in this environment may be untrustworthy and hence they may attempt actions to disrupt or cripple the operating system. So a primary concern in every solution is that all system components are adequately protected from user tampering.

First we implement an efficient disk scheduling algorithm. It demonstrates the use of the release clause in resources to permit simultaneous scheduling and use of the disk. Next we design a dynamic buffer allocation scheme. An interesting aspect of of the solution is that a user can access instances of a buffer type but cannot create any new ones himself. The third problem is to design a simple file system. This example shows how the ownership principle can be introduced at the system design level. The final problem is to design an efficient system for managing interactive terminals. In this case we introduce the possibility of allowing one resource to manage several identical I/O devices.

We conclude this chapter with a detailed evaluation of our features based on the expressiveness criteria developed in Chapter 2. The examples of this chapter help establish the fact that we have met all but two of our initial goals. We also use them to directly compare our facilities with the existing systems discussed in Chapter 2. This comparison emphasizes the value of our extensions and new mechanisms in specific, practical situations by considering how equivalent solutions could be achieved without them.

7.1. Disk Scheduling

Consider the problem of scheduling access to a disk, where at any point in time several users may want to use the disk. Since the service time for a (moving-head) disk request increases with head movement, we must schedule the users in a manner which controls that movement if we want to give efficient service. A variety of algorithms have been proposed to solve this problem and most have one common characteristic--that new requests are scheduled while the disk is servicing some user. This characteristic is critical to an efficient solution.

From a protection point of view, the main problem is to insure that the scheduling algorithm is not subverted. We must force each user to give his requests to the scheduler and also verify that each request generates the proper disk access (i.e., a user cannot request cylinder A, and then somehow cause cylinder B to be accessed instead). The reason for this protection should be obvious. The efficiency of the scheduling algorithm is based on always knowing the exact location of the disk head. If a user could move the head without the scheduler's knowledge, the purpose of scheduling would be defeated.

In the solution to the disk scheduling problem outlined in Figure 7.1, two resources are used to satisfy the design constraints. The real disk is accessed via the R_DISK resource. Since its internal structure is highly machine-dependent we omit it here. The virtual disk resource, V_DISK, implements the scheduling algorithm. All users make their requests to this resource. The release phrase in V_DISK permits scheduling and disk operations to proceed simultaneously.

An interesting aspect of this solution is that we do not need to see the specific scheduling algorithm to understand how the required protection is insured. The use of the R_DISK resource is limited to V_DISK by the single restrict command. Therefore, a USER's only option is to call V_DISK for each disk request. At the proper time V_DISK invokes R_DISK to execute the I/O, so a user has no way of altering a request before the disk access is made.

In Figure 7.2 we give the details of the scheduling resource, a variation of the elevator algorithm described by Hoare [33]. The disk head moves in complete sweeps across the cylinders. All requests for a particular cylinder are serviced when the disk head moves over that cylinder. We implement the algorithm by keeping two sorted lists of disk requests--those to be serviced on the next up-pass, and those for the next down-pass. An element on either list contains a cylinder number and a count of users requesting that cylinder. When a new request enters V_DISK the information is stored in one of these lists. Each process waits for the disk status variables to take the proper values. When a new cylinder is selected for service, all tasks waiting for that cylinder may proceed to R_DISK simultaneously. The tasks actually restart one at a time from

```
SYSTEM: program;

    var R_DISK: static resource;
                operations READ,WRITE;
                :
                end R_DISK;

    restrict R_DISK to V_DISK;

    var V_DISK: static resource;
                operations READ,WRITE;
                release READ to R_DISK.READ,
                        WRITE to R_DISK.WRITE;
                :
                end V_DISK;

        USER: process;
                :
                V_DISK.READ(   );
                :
                V_DISK.WRITE(   );
                :
                end USER;
    :
    end SYSTEM
```

Figure 7.1: Structure of the disk scheduling solution.

```
V_DISK: static resource;
   operations REAO,WRITE;
   release REAO to OISK.REAO,
         WRITE to OISK.WRITE;
   var HEAO_POSITION: 0..NUM_OF_CYLINOERS;      /* OISK STATUS VARS.    */
      GOING_UP: Boolean;                        /* TASK COUNTS PENDING */
      TASKS: 0..NUM_OF_USERS                    /* JOBS ON CURRENT     */
                                                /* CYLINDER            */
   type LIST_ELEMENT = record;
                     CYL_NUM: 1..NUM_OF_CYLINDERS;
                     REQUESTS: 0..NUM_OF_USERS;
                     LINK: integer
                     end;
   var LIST: array 1..NUM_OF_USERS of LIST_ELEMENT;
      FREE_LIST,                                /* INDEXES INTO        */
      UP_LIST,                                  /* LIST IOENTIFYING    */
      DOWN_LIST: 0..NUM_OF_USERS;               /* FIRST ELEMENT       */

   READ: procedure (CYLINOER: 1..NUM_OF_CYLINDERS,
                  TRACK: 1..NUM_OF_TRACKS,
                  OFFSET: 0..TRACK_SIZE,
                  INFO: BUFFER pointer);

      if HEAO_POSITION > 0                     /* i.e., disk is in use */
         then if CYLINOER > HEAO_POSITION
            then begin            /* SERVICE REQUEST NEXT UP PASS      */
               INSERT_UP_LIST(CYLINOER);
               await CYLINOER = HEAD_POSITION & GOING_UP
               end
```

Figure 7.2: V_OISK resource for disk scheduling solution (continued on on next page).

```
        else begin              /* SERVICE REQUEST NEXT DOWN PASS */
            INSERT_DOWN_LIST(CYLINDER);
            await CYLINDER = HEAD_POSITION & ¬ GOING_UP
            end

    else begin
        HEAD_POSITION := CYLINDER;
        TASKS := 1
        end;

DISK.READ(CYLINDER,TRACK,OFFSET,INFO);            /* exit V_DISK */
TASKS := TASKS - 1;
if TASKS > 0 then return;  /* unfinished requests this cylinder */
if GOING_UP then REMOVE_UP_LIST(HEAD_POSITION,TASKS);
            else REMOVE_DOWN_LIST(HEAD_POSITION,TASKS);
if HEAD_POSITION > 0 then return;            /* found new tasks    */
GOING_UP := ¬ GOING_UP;                      /* reverse direction */
if GOING_UP then REMOVE_UP_LIST(HEAD_POSITION,TASKS);
            else REMOVE_DOWN_LIST(HEAD_POSITION,TASKS);
end READ;

WRITE is identical to READ except for call to R_DISK
```

The Insert routines search down the appropriate list adding the new request in the proper place. For UP_LIST cylinder numbers increase moving through the list. For DOWN_LIST they decrease. The Remove routines take the top element off the appropriate list and return the CYLINDER number and task count for that cylinder. If there are no tasks on a list, the procedure returns a cylinder value of zero.

Figure 7.2 (cont.): V_Disk resource for disk scheduling solution (continued on next page).

```
PRE_OP: procedure;
    var I: 1..NUMBER_OF_USERS;
    /* put all list elements on Free-list */
    for I := 1 to NUMBER_OF_USERS - 1 do
            LIST(I).LINK := I + 1;
    UP_LIST := 0;
    DOWN_LIST := 0;
    FREE_LIST := 1;
    LIST(NUMBER_OF_USERS).LINK := 0;
    TASKS := 0
    end PRE_OP

end V_DISK
```

Figure 7.2 (cont.): V_DISK resource for disk scheduling solution.

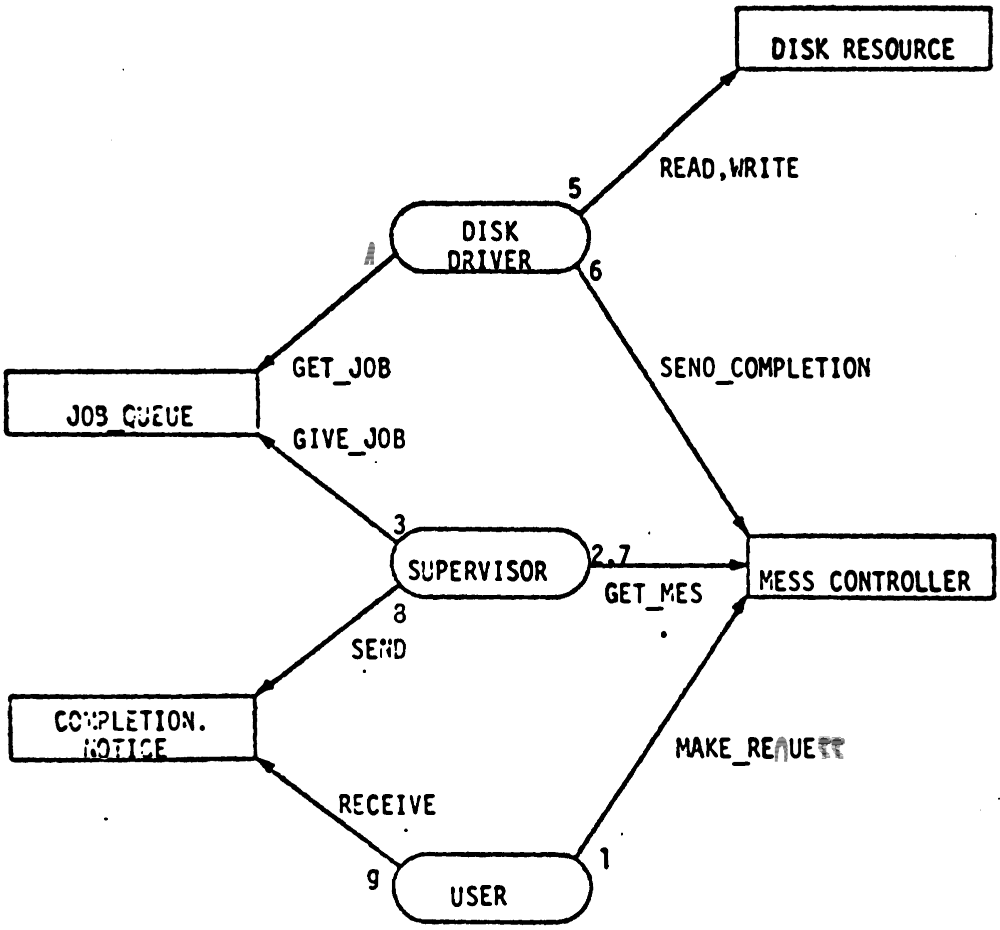
the <u>await</u>, but the <u>release</u> clause permits another to go as soon as the previous one calls out to R_DISK. Hence all the tasks will queue up at R_DISK. The last returning one sets the status variables for the next cylinder that has requests.

The purpose of this implementation is to maximize disk utilization. We do this by minimizing the amount of time necessary to select the next task for R_DISK. V_DISK permits all tasks waiting for the current disk head position to call R_DISK simultaneously (R_DISK will serve them one at a time). When it is time for the disk head to move, V_DISK finds the next position without having to perform a search. The next position must be identified in the first element of either the UP_LIST or DOWN_LIST. All scheduling (and searching) overhead is incurred when a process initiates a request. At that time an INSERT procedure must search down a list to find the proper location. But this execution time probably overlaps usage of the disk, so it does not affect disk utilization.

The main significance of this solution is its straightforward realization of a systems design. It accurately implements the policies (synchronization and access control) without extra procedures or run-time checks. The <u>release</u> clause permits the virtual disk to be scheduling at all times (particularly while some tasks are using the actual disk), while the <u>restrict</u> phrase effectively makes the virtual disk resource the owner and only possible user of the real disk. From a user's view, the virtual disk <u>is</u> the "real" disk. When he invokes a disk operation, the work on the disk is finished before the call returns. In fact the user process, executing within a system-defined resource, is operating on the disk.

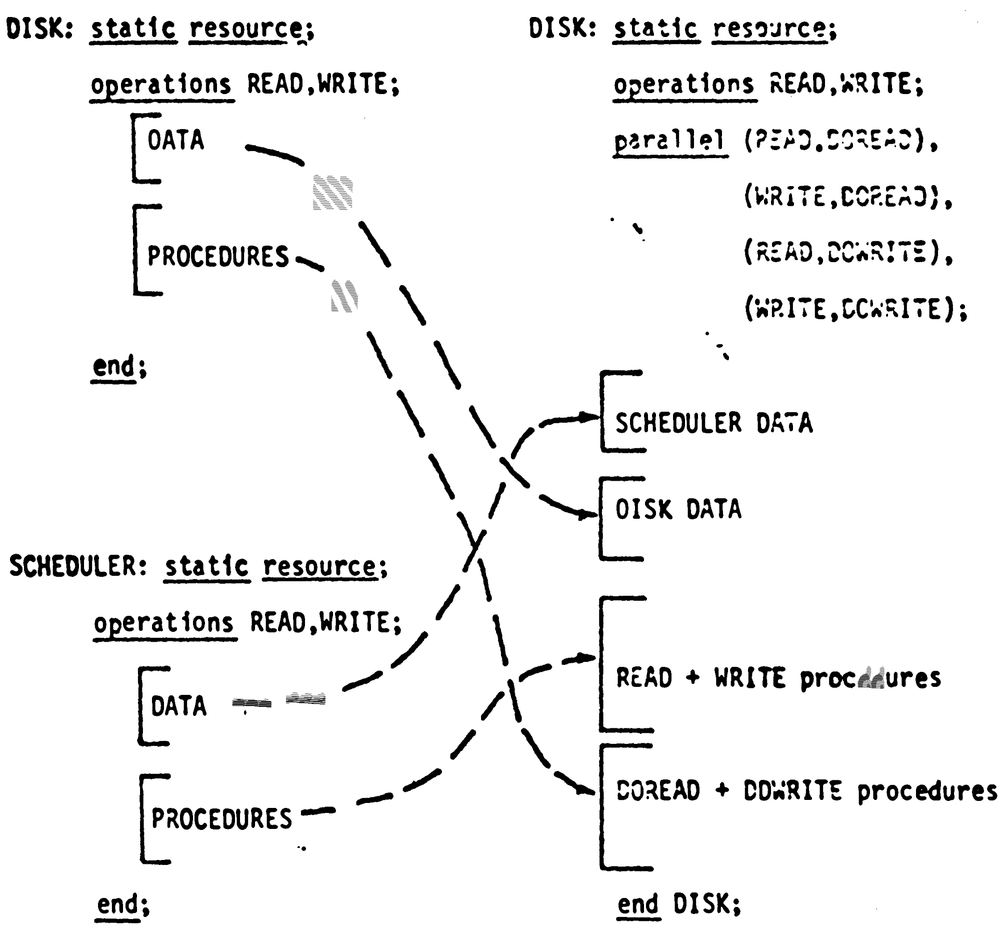
Note that these language features can handle many different disk scheduling designs, not just this particular one. A system could be organized so that a user process need not be delayed while his disk request is pending. He could send a message to a system process which completes the request and then sends a completion message back. Such an organization is displayed in Figure 7.3. In this case, the scheduler would be imbedded in a system process (SUPERVISOR). In order to simultaneously schedule and use the disk, a second system process (DRIVER) would actually initiate all disk operations. While this structure seems complex, the complexity is in the design, not in its realization with our features. The only effect caused by our features is that the DRIVER and USERs must share the same input line (MESS_CONTROLLER) to SUPERVISOR since it must be able to simultaneously wait for messages from either source. Notice, however, that the separate operations available to each sender prohibits a USER from disguising a request message as a completion message.

A third possible design for disk scheduling would be to merge the scheduler and real disk into one resource. Our initial design separated these tasks to illustrate the potential for modular design. We could, however, get an equivalent solution with only resource, as shown in Figure 7.4. We merge the contents of the scheduler and real disk into one place. The simultaneous execution (scheduling and use) is permitted via the <u>parallel</u> clause (instead of <u>release</u>). Notice that the interference-free conditions are satisfied because in each pair specified, one procedure was from the scheduler and the other from the real disk (from the first solution we know these groups did not share any kinds of variable accesses).



The numbers on the arcs show the order in which the operations would be used to process one request.

Figure 7.3: Alternate disk scheduling design.



Scheduling with two resources. Scheduling with one resource.

Figure 7.4: Disk scheduling with one resource.

This design flexibility is, we feel, an excellent practical example of the expressiveness which we have as our primary goal.

7.2. Buffer Allocation

The purpose of buffer allocation is to be able to give blocks of memory to users on a temporary basis. When a user acquires a block, he is free to use that space as he chooses. One example where such a scheme would be useful is a spooling system. A spooler process acquires a buffer and fills it with data from a card reader. He can then transfer the buffer to a supervisor for further processing. The dynamic allocation of buffers saves the spooler from declaring static space which he may not always use and which would force extensive copying when information is transferred to a supervisor. The major constraint on the solution is that when a process (or resource) acquires buffer space, it must have exclusive access to that buffer (i.e., we do not want two spoolers putting data in the same buffer at the same time).

From the buffer allocator's point of view, we need one additional protection guarantee. The allocator must be the only system object that can create buffer instances. Without this assurance, an allocator must make provisions for the situation where more buffers are returned than were initially created. This guarantee is conceptually difficult because users must be able to access instances of buffers. Hence, they need enough information about a buffer type to use an instance, without being able to create more of them.

In Figure 7.5 we implement a buffer as an instance of a <u>protected</u> array type called BUFFER. The static resource, BUFFER_ALLOCATOR, is responsible for creating and managing all buffer instances. During

```
SYSTEM: program;
    type BUFFER = protected array 1..1024 of char;
    restrict BUFFER to BUFFER_ALLOCATOR,SYSTEM;
    type BUF_POINTER = BUFFER pointer;

    BUFFER_ALLOCATOR: static resource;
          operations REQUEST,RELEASE;

          var FREE_BUFFERS: array 1..MAX_BUFS of BUF_POINTER;
              BUFFER_COUNT: 0..MAX_BUFS

          REQUEST: procedure (BUF: BUF_POINTER);
              var I: 1..MAX_BUFS;
              if active (BUF) then return;          /* pointer must be */
                                                    /* initially empty */
              await BUFFER_COUNT > 0;
              BUFFER_COUNT := BUFFER_COUNT - 1;
              I := 1;
              while ¬ active (FREE_BUFFERS(I)) do
                    I := I + 1;
              BUF ← FREE_BUFFERS(I)
              end REQUEST;

          RELEASE: procedure (BUF: BUF_POINTER);
              var I: 1..MAX_BUFS;
              if ¬ active (BUF) then return;        /* pointer must    */
                                                    /* contain a valid */
                                                    /* reference       */
              BUFFER_COUNT := BUFFER_COUNT + 1;
              I := 1;
              while active (FREE_BUFFERS(I)) do
                    I := I + 1;
              FREE_BUFFERS(I) ← BUF
              end RELEASE;
```

Figure 7.5: Solution to buffer allocation problem (continued on next page).

```
PRE_OP: procedure;
    var I: 1..MAX_BUFS;
    for I := 1 to MAX_BUFS do
        FREE_BUFFERS(I) := BUFFER. create;
    BUFFER_COUNT := MAX_BUFS
    end PRE_OP

  end BUFFER_ALLOCATOR;
  .
  .
  .
end SYSTEM
```

Figure 7.5 (cont.): Solution to buffer allocation problem.

initialization, the allocator creates the buffers and stores a reference to each in its array of pointers. When a request is made, ALLOCATOR searches for a pointer holding a valid reference and then transfers that reference to the caller. When a buffer is returned (via RELEASE), ALLOCATOR transfers the reference back to an empty pointer. The only protection enforced by the resource itself is that the pointer passed as a parameter has the proper status (e.g., on a call to RELEASE, the parameter actually contains a valid BUFFER reference).

The constraint that each user must have exclusive access to a buffer is enforced by the language. Since the buffers are implemented as <u>protected</u> objects, the language only allows one pointer variable to reference each buffer. Moreover, a pointer variable cannot be shared by two processes thus insuring exclusive access.

The constraint that users may access but not create buffer instances is enforced in another way. Our solution to this facet of the problem requires a precise understanding of our mechanism for creating objects. Notice that in any <u>create</u> command the type of the new object must be specified. For example, in Figure 7.5 the buffers are created by:

```
FREE_BUFFERS(I) := BUFFER. create;
```

The use of the word "BUFFER" is redundant in this statement since the type of the object must have been specified when the array of pointer variables was declared earlier. However, this redundancy implies that a process (or resource) must have access to the type name in order to create an instance of the object. This point is subtle, but very useful. In our buffer allocation solution, the single <u>restrict</u> command blinds the BUFFER type definition to all USER blocks; hence they cannot create buffer instances.

On the other hand, they do have access to the type BUF_POINTER which allows them to define pointer variables for buffers and use objects passed to them.[1]

Before leaving the buffer allocation problem we should discuss one weakness of our solution. Since the allocator gives up all knowledge of a buffer when it gives one to a user, return of that buffer can only be initiated by the user. What happens if the user does not choose to return the buffer? In particular, what if he dies while possessing a buffer? In our system we have two alternatives--(1) force a user to return the buffer before he dies, or (2) let the buffer die with the user, but replace it with another. The first alternative can only be achieved by a tricky use of static resources which uses the automatic POST_OP procedure as the key to returning the buffer. Unfortunately, this approach puts procedure call and synchronization overhead on every access to a buffer by a user. Hence, it defeats the purpose of the protected variable system. The second solution is better, but still not very pleasing. When a process dies, we can check to see if it had any buffers (by keeping track of which processes we gave them to), and if so create new ones to take their place. Neither of these solutions is appealing, but we want to resist changes to cope with every potential weakness.

7.3. A File System

The third problem we examine is the design of a simple file system. Here the major task is to turn disk space into abstract objects called FILES, and give USERS the ability to manage their own instances of these

[1] We used a similar approach in Chapter 5 to get use-without-create control over statically-defined objects.

files. We also want to introduce the "ownership" concept. If a user creates a file, we want to give him sole control over which other users may access it and how they may access it. Further, we want a user to be able to effectively destroy any file he has created.

We are concerned with two protection constraints. First, we do not want two files to share the same disk space. Second, we want each logical file to limit its disk accesses to the area specified for that file.

The design of the file system is displayed in Figure 7.6. USERS are given the right to create instances of the dynamic resource FILE. During initialization, each new file instance calls the DISK_ALLOCATOR to acquire an area of disk space. When the user calls the FILE, the request is translated into a disk request which FILE forwards to V_DISK (defined in Section 7.1, Figure 7.2). While a file is open (i.e., ready for reading and writing), the file resource uses buffer space to keep active portions of the file in core. These buffers are acquired from the BUFFER_ALLOCATOR described in the previous section. In addition to using a file, a user may want to give away some access rights to other users. The FILE_DISTRIBUTOR resource provides a communication channel for this purpose.

We now elaborate on some of the details involved with implementing the file system. The V_DISK and BUFFER_ALLOCATOR resources have already been defined in previous sections, so we will not redefine them here. The FILE_DISTRIBUTOR resource is basically a message passing facility similar to others we have done, except that in this case the message is of type FILE capability. Since it illustrates no new concepts, we will not define it here either. The two remaining resources, DISK_ALLOCATOR and FILE, are

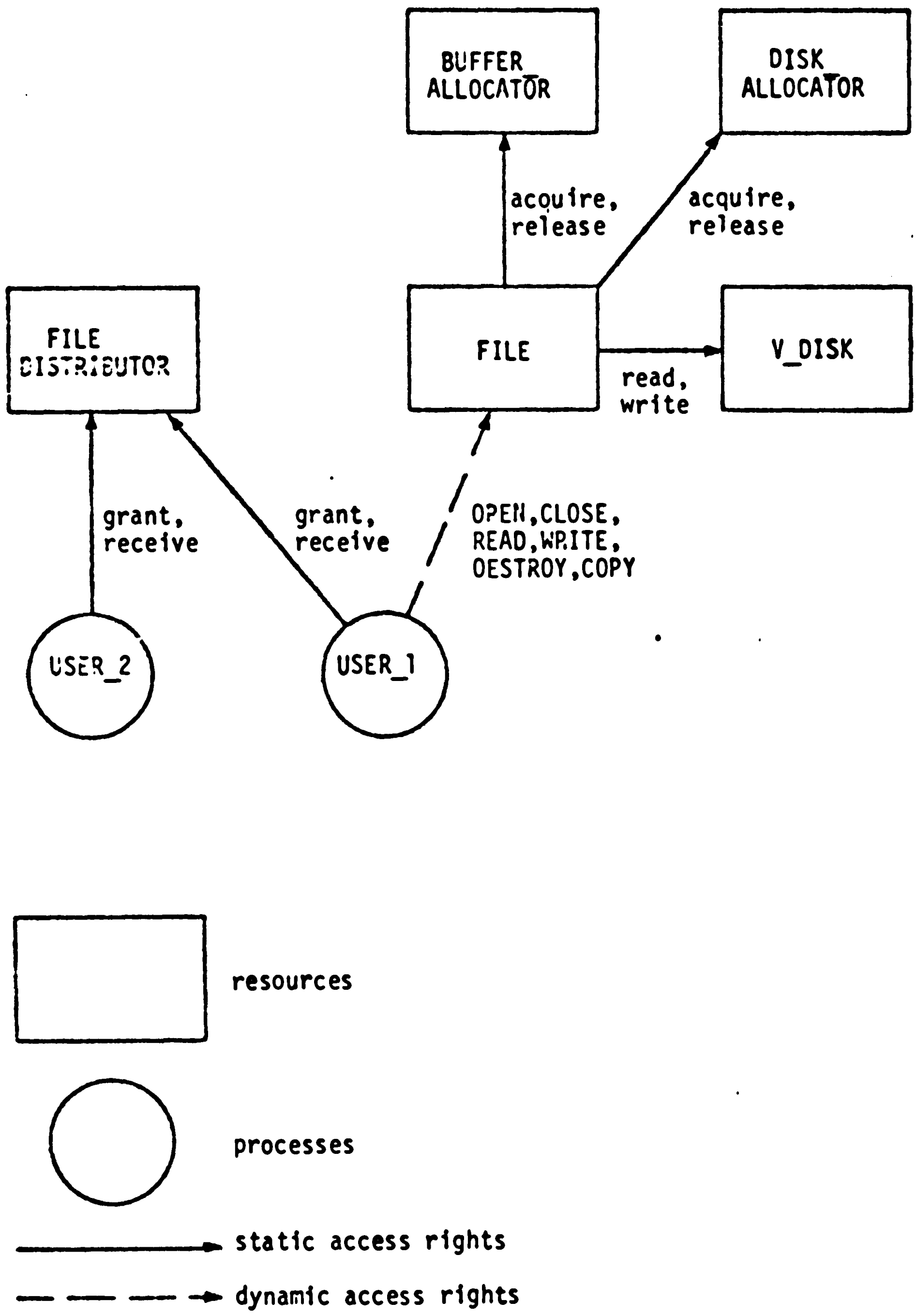


Figure 7.6: Diagram of a simple file system.

the key components of the system. The code for the former is outlined in Figure 7.7, and the latter in Figure 7.8.

The DISK_ALLOCATOR has complete responsibility for managing the use of space on the disk. It implements a memory management algorithm (which we leave unspecified) that permits FILES to request and release disk space. On a call to ACQUIRE (from FILE), the allocator finds a block of free space and its starting address. When RELEASE is called, the block o space identified by the parameters is recognized as being free.

This portion of the solution shows that safe use of the disk for accessing files is accomplished by an agreement between the allocator and file blocks. The allocator must insure that two files do not overlap in their address spaces, and each file must limit its disk accesses to the range specified by the allocator. If either of these conditions is broken, two files could overlap on the disk. We feel comfortable with this design because both of the blocks involved in the agreement are part of the operating system, and their actions are encapsulated in resources. In particular, a user may create a file instance, but the operating system has pre-defined the FILE type which every USER must use. By our scope rules we can insure that this path to the disk is the only one open to each USER. Hence, this approach has the necessary protection.

The FILE resource given in Figure 7.8 is the interface between the disk and the USER. The READ and WRITE operations translate USER requests into the appropriate disk operations. Each FILE maintains one buffer area for holding a physical record. If the file request happens to be for the current block in the buffer, no I/O is performed. Otherwise a different block is brought into the buffer. A file holds the buffer space only

```
DISK_ALLOCATOR: static resource;
                operations ACQUIRE,RELEASE;

                Data for maintaining status of all disk space.

                ACQUIRE: procedure (SIZE: integer, START: DISK_ADDR);

                         Find space having proper length--if none, wait.
                         Update status variables.
                         Return starting address of space in START.
                         end ACQUIRE;

                RELEASE: procedure (SIZE: integer, START: DISK_ADDR);

                         Mark disk space as free.
                         end RELEASE;

                PRE_OP: procedure;
                         Initialize all disk space to free.
                         end PRE_OP

                end DISK_ALLOCATOR
```

Figure 7.7: Skeleton of the disk space manager.

```
type FILE = dynamic resource (SIZE: integer);
        operations OPEN,CLOSE,READ,WRITE,DESTROY;

        var START_FILE: DISK_ADDR;      /* Start of file on disk        */
            BUF: BUF_POINTER;           /* pointer to in-core buffer    */
            PHYSICAL_REC: DISK_ADDR;    /* addr. of record in core      */
            ACTIVE_FILE: Boolean;       /* true--if file not destroyed  */
            MODIFIED: Boolean;          /* true--if buffer contents     */
                                        /*       have been modified     */
        OPEN: procedure;
            /* Get an in-core buffer, but only if the file has not      */
            /* been destroyed and not already opened.                   */
                if ACTIVE_FILE & ¬ active (BUF)
                    then begin
                        BUFFER_ALLOCATOR.REQUEST(BUF);
                        PHYSICAL_REC := EMPTY
                        end
            end OPEN;

        CLOSE: procedure;
            /* If the file is active, transfer buffer contents to       */
            /* disk (if necessary), and return buffer.                  */
                if ¬ ACTIVE_FILE then return;
                if (PHYSICAL_REC ≠ EMPTY) & MODIFIED
                    then begin
                        Compute proper location on disk
                        V_DISK.WRITE(..., BUF)
                        end;
                BUFFER_ALLOCATOR.RELEASE(BUF)
            end CLOSE;
```

Figure 7.8: File resource for file system solution (continued on next page).

```
READ: procedure (OFFSET: integer, RESULT: char);
    var TARGET_BLOCK: DISK_ADDR;
        TARGET_OFFSET: integer;
        :
        :
        if (OFFSET > SIZE) |       /* location out of range   */
             ¬ active (BUFFER) |  /* file not open           */
             ¬ ACTIVE_FILE        /* file has been destroyed */
                then return;

        Use file offset to compute disk location being
        referenced.  Put physical record id in TARGET_BLOCK
        and offset within record in TARGET_OFFSET.

        if TARGET_BLOCK ≠ PHYSICAL_REC
             then begin               /* get proper record   */
                if MODIFIED
                     then begin     /* first store old rec */
                          V_DISK.WRITE(___,BUF)
                          end;
                V_DISK.READ(___,BUF);
                PHYSICAL_REC := TARGET_BLOCK;
                MODIFIED := false
                end;

        RESULT := BUF↑ (TARGET_OFFSET)

        end READ;

        WRITE is nearly identical to READ--in last line
        reverse direction of assignment, and also set MODIFIED
        to true
```

Figure 7.8 (cont.): File resource for file system solution (continued on next page).

```
DESTROY: procedure;

    Execute same operations as in CLOSE.

    ACTIVE_FILE := false;
    OISK_ALLOCATOR.RELEASE(SIZE,START_FILE)
    end OESTROY;

PRE_OP: procedure;
    ACTIVE_FILE := true;
    DISK_ALLOCATOR.ACQUIRE(SIZE,START_FILE);
    PHYSICAL_REC := EMPTY;
    MODIFIED := false
    end PRE_OP;

POST_OP: procedure;
    If the file has not been destroyed yet, do so.
    end PDST_OP

end FILE;
```

Figure 7.8 (cont.): File resource for file system solution.

while the file is actually open. The OPEN and CLOSE operations permit USERS to control the current state of the file. The last FILE operation is DESTROY; its effect is to make the FILE completely inactive. All disk and buffer space is returned, and any future calls to a destroyed file will simply return without taking any action.

This form of file destruction is the price we pay for choosing to avoid the ownership principle in designing access mechanisms. With our language, a FILE instance can only die when no more references to it exist. If we want to let a file creator have the right to destroy the file, no matter who has current access to it, the best we can do is to let the creator make that file inoperative. The overhead of this approach is that each file operation must check to see if the file is active. The advantage is that if a USER tries to access a destroyed file he does not get a run-time protection error. The call simply fails to do anything. With suitable extensions to FILE, the USER could be told very politely that the file is gone. This error control allows a message to be returned to the user instead of some hardware protection fault, but the user pays the price of checking after each operation to determine if it was done properly.

A logical extension to this implementation is to permit each file to use multiple buffers instead of just one. In effect, a file would become a mini-virtual memory manager. Each file remembers which disk records are in buffers, and moves the blocks between memory and disk based on user requests. In this case it would be reasonable to have all calls to V_DISK <u>release</u> control of FILE, so that some requests for access to the file would not be held up by delays from disk usage. (The <u>release</u> option is not useful in this example since the one buffer forces FCFS scheduling on the file.) The only potential problem is forgetting to recognize when a buffer is being used to transfer data. When a buffer is passed to V_DISK, it is no longer accessible to the file until it is returned. If the disk call is released, another user may try to access information on the outgoing buffer. If the file fails to recognize this situation, it will reference an empty pointer which is an error. However, if the file is coded properly the multiple buffer extension can be an effective design approach.

To complete the file system design, in Figure 7.9 we sketch the possible actions of one USER process. This user creates two files, storing the full rights for those files in capabilities FILE_1 and FILE_2. He then chooses to share access to his files with two processes--one being his offspring, SON, and the other is some externally-defined process. SON gets all-but destroy rights for the first file and read-only rights for the second file. SON references the files through capabilities F1 and F2. Given this structure, those are the only capabilities he can access. The default scope rules prohibit him from using the names FILE_1 and FILE_2 which could give him more access rights. Thus, SON is effectively limited to those rights passed to him (in this case, during initialization).

The only other file rights go to a process identified by TARGET. This process acquires access to the user's second file through the FILE_DISTRIBUTOR resource. This resource is necessary because the two processes involved have no other common link for transmitting FILE capabilities. (In the case of the son, the user could pass capabilities at initialization, or he could define his own resource for passing rights to

```
USER: process;
      type FILE_CAP = FILE capability;
      var FILE_1,FILE_2,STRONG_RIGHTS,
          WEAK_RIGHTS,FRIEND_RIGHTS: FILE_CAP;
      :
      FILE_1 := FILE. create (1024);
      FILE_2 := FILE. create (2048);
      STRONG_RIGHTS := FILE_1{OPEN,CLOSE,READ,WRITE};
      WEAK_RIGHTS := FILE_2{READ};
      FRIENDS_RIGHTS := FILE_2{OPEN,CLOSE.READ,WRITE,COPY};

      FILE_DISTRIBUTOR.GRANT(FRIENDS_RIGHTS,TARGET);
      :
      begin
          type SUB_TASK = process (F1: FILE_CAP, F2: FILE_CAP);
                              :
                              F1.WRITE(OFFSET,VALUE);
                              F2.READ(OFF_2,RESULT);
                              :
                              end;
          var SON: SUB_TASK(STRONG_RIGHTS,WEAK_RIGHTS);
              SON. setstate (   );
              SON. activate;
              :
              FILE_1.OPEN;
              FILE_2.OPEN;
              :
          end
      end USER
```

Figure 7.9: User of the file system.

his son.) Once the USER gives the file rights to another process, he has no assurance that those rights will ever be surrendered. However, by keeping the destroy to himself, he has the final word over actual file usage.

Having to destroy a file to prevent access by one particular user is the major weakness of this solution. We could extend the definition of FILE to force all users to supply identification for each file access. However, this option requires run-time checking through programmed code, which we were trying to avoid. The problem comes down to the basic question of whether or not to permit unilateral revocation of rights. By not permitting it in the language (as we have done), it is difficult and expensive to introduce it in a program.

7.4. Interactive Terminal Processing

The last operating systems problem we consider is the control of remote terminals for interactive processing. The task here is to provide communication paths between people working at terminals and their internal processes. For this example, we assume that the terminal will initiate contact by transmitting a password. The operating system must use this password to determine which process should be linked to the terminal. After initial contact, both the process and the terminal may transmit messages to each other at any time. In particular, the terminal need not wait for the process to request input. The communication path must remain open until the terminal transmits an end-of-job signal. Any data transmitted from a terminal after an EOJ is assumed to be a password for starting a new communication link.

Three specific aspects of this problem are of interest to us. The first is to implement a protection scheme which insures that only the proper internal process will be able to communicate with a terminal. With a password system we must trust the person at a terminal to correctly identify the process with which he wishes to talk. But we can insure that no other process can interfere with the communication. The second part of the problem is to handle I/O initiated by a terminal, without requiring a process to make a request. Some provision must be made for receiving this information when a process is not waiting for it. The last part of the problem is to deal with the extremely slow transfer rate from terminals (10 char/sec maximum is common).

Because of the complexity of this problem, we will develop a solution in stages. The first level will consider only the protection aspect of the problem. Successive refinements will incorporate the other two aspects. This approach should help clarify the function of the various components in our final design.

The simplest solution to the protection problem is to use capabilities to represent access rights for each terminal, as pictured in Figure 7.10. We use dynamic resources to drive the external devices rather than static resources as previously discussed. A terminal allocator resource creates the terminal instances, giving each one a different real device to control permanently. A user process then calls the allocator to get a capability for a terminal. Once he acquires a capability, the process can open communications himself. As long as the allocator only gives out one capability for each terminal (without giving the "copy" right) we can be certain that at most one user is accessing each terminal. Unfortunately,

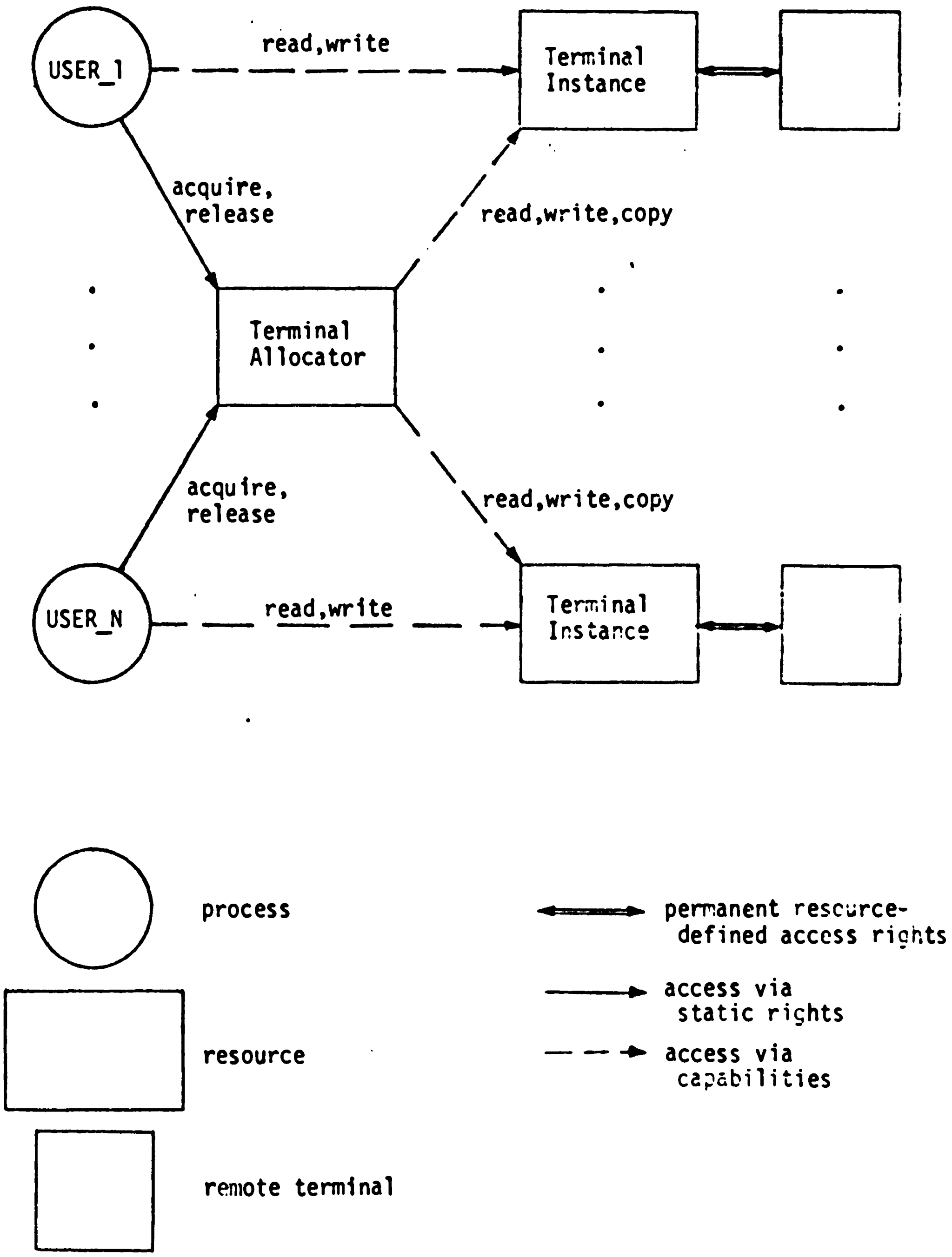


Figure 7.10: First design for handling interactive processing.

this design cannot handle terminal-initiated conversations. Hence, we must make some modifications.

To permit link initiation by a terminal we associate a dedicated system process with each terminal, as sketched in Figure 7.11. Each terminal driver process is responsible for receiving all data transmitted from a terminal. When a terminal initiates a communication, the password is sent to the allocator which returns a buffer resource for the terminal to use. The driver puts all future communications in the buffer (until an EOJ is sent). In this design users still invoke the allocator to acquire a communication link, but now they get a buffer resource. This resource can return information read by the driver and also transmit WRITE requests directly to the terminal instance. When the terminal sends an EOJ, the information is sent to the allocator. The allocator then tells the buffer to disconnect itself from the terminal. This tactic effectively invalidates a user's access rights without actually destroying them.

The major drawback of this design is the overhead of having a process associated with each device, particularly when the device is so slow. The transfer rate from terminals is slow enough for one process to handle many of them. The maintenance necessary for defining and controlling individual device drivers generates a significant amount of overhead. The problem with using only one driver process, however, is that the driver cannot simultaneously wait for input from several device resources. It can only access one resource at a time.

Our solution to this dilemma is to modify the approach we take for linking resources to devices. Instead of having a one-to-one pairing, in this type of situation it is reasonable to have one resource control many

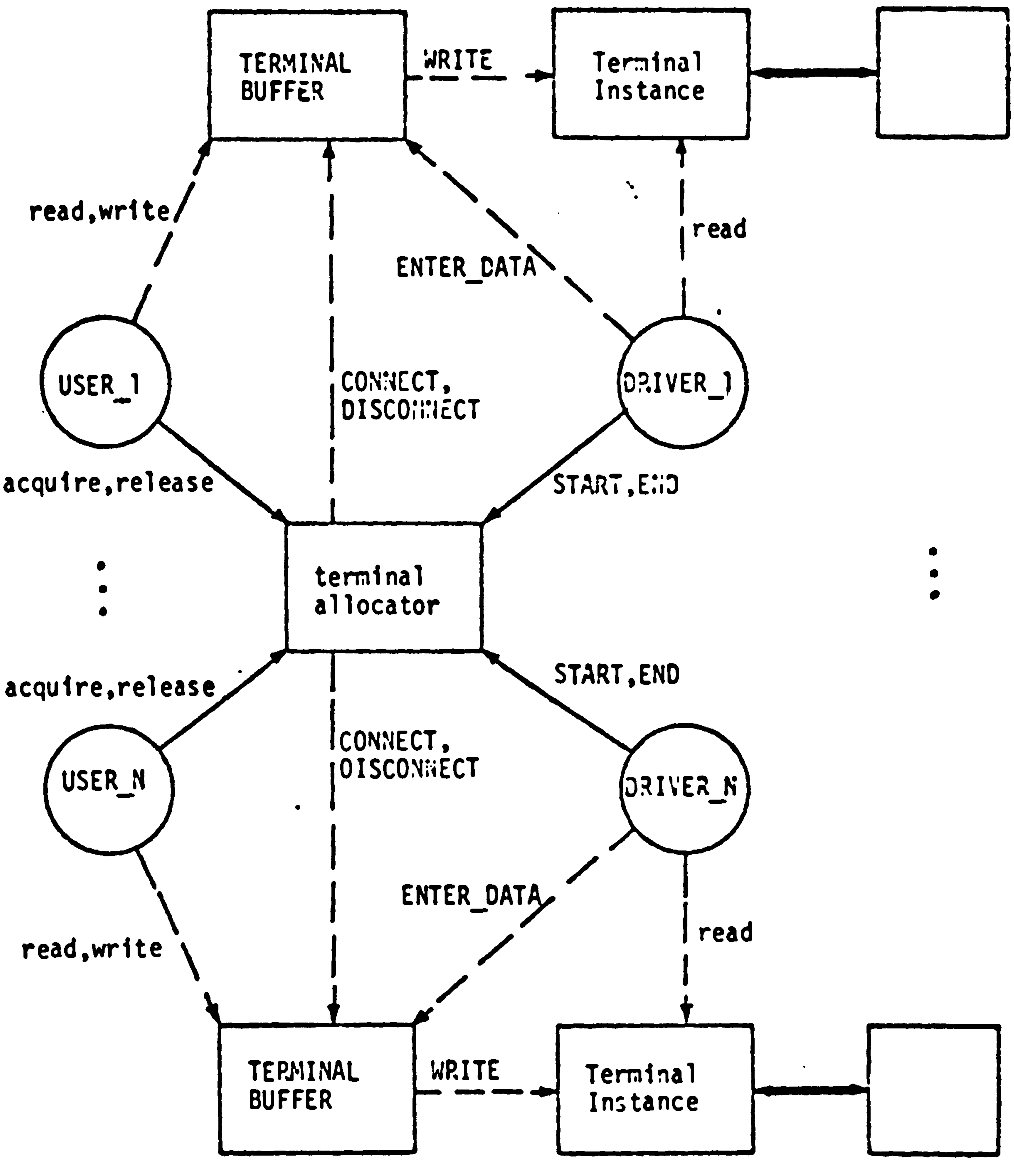


Figure 7.11: Second design for interactive terminal processing.

devices. One resource called TERMINAL_CONTROLLER could handle I/O to all devices. A plausible structure for this resource is shown in Figure 7.12. READ returns the next line transmitted from <u>any</u> terminal device along with the identity of that device. WRITE sends an output line to the device specified as part of the call. In this way we shift the problem of simultaneously waiting for several device signals inside the resource where hardware-dependent implementations can deal with it. Admittedly, we are somewhat sidestepping the problem by subsuming it into a more complex resource. However, we feel that it is better to handle it there than to let it affect the entire system design.

Once we put the responsibility for fielding information from several terminals into one resource, the system design becomes simpler, as shown in Figure 7.13. All the device drivers (processes) and terminal instances are replaced by one driver and one terminal resource. The one driver is now responsible for sending the input information to the proper terminal buffer. The interface with the users remains exactly the same as in the previous design. As a result, the DRIVER and CONTROLLER objects are slightly more complex. Savings accrue from having fewer processes to manage (less time and space needed in the process scheduler defined by the language), and fewer resource instances to control (requiring less space also).

The actual implementation of the final design is specified in Appendix B. Because the actual code is long and the details do not provide any new insights, it is omitted here.

The major significance of this example is the flexibility for design provided by the language. Each of the schemes we outlined can be

```
TERMINAL_CONTROLLER: static resource;
          operations READ,WRITE;

              var BUF_IN*: array 1..NUM_TERMINALS of LINE;
                  FILLED*: array 1..NUM_TERMINALS of Boolean;
                  :
          READ: procedure (INFO: LINE, DEVICE: DEVICE_IO);
              Wait until some element of FILLED equal true.**
              Get a line from BUF_IN, which has corresponding
              element of FILLED equal true.  Reset FILLED
              entry to false, and return line in INFO.
              end READ;

          WRITE: procedure (LINE: OUTPUT_LINE, DEVICE: DEVICE_ID);
              Transmit LINE to appropriate DEVICE.
              end WRITE

          end TERMINAL_CONTROLLER
```

Figure 7.12: Resource structure for controlling several terminals.

---

*Language receives line from device and puts it in appropriate entry of BUF_IN. It also sets corresponding entry of FILLED to true. These actions require special syntax (which we have not used), and complete agreement between the language and system designers, so they are very hardware-dependent.

**Special await statement needed here, to work in conjunction with the interface discussed above.

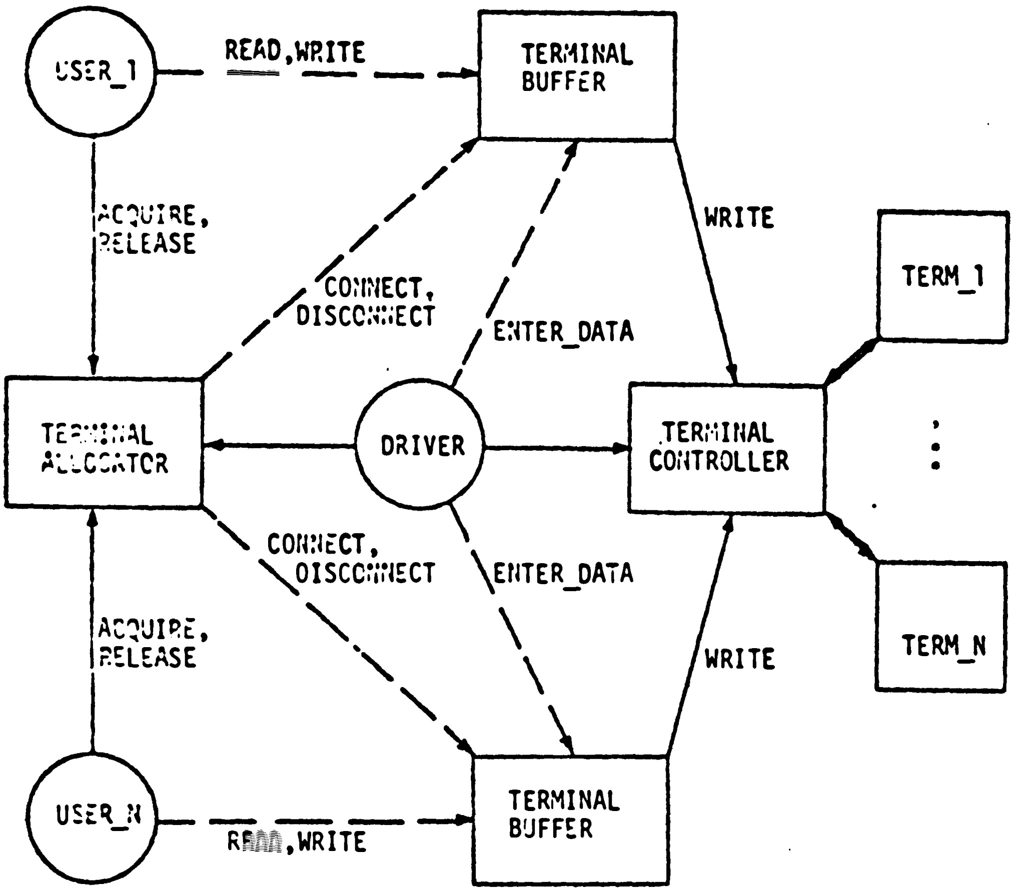


Figure 7.13: Final design for interactive terminal processing.

implemented with our features. We selected a difficult interaction polic (communication initiated from either direction) to illustrate its power. However, the other designs are reasonable for different system organizations.

7.5. Evaluation of Expressiveness

With these four detailed examples we can attempt to evaluate the expressiveness of our new language facilities. In Chapter 2 we developed criteria for evaluating both interaction and access control mechanisms. We now apply them to our facilities in order to give a reasonably objective analysis of their benefits.

7.5.1. Interaction Analysis

The resource mechanism is firmly based on the monitor concept devel oped by Hoare. In fact, if the special extensions are removed (POST_OP, parallel, and release) a resource is actually a monitor.[2] Therefore, mo of our analysis of monitors in Section 2.2 applies equally here. If a mon tor can be used to solve a problem well, a resource can do at least as well. So here we concentrate on areas where monitors proved to be inade qute--namely representation of temporary objects, multi-level interactic with simultaneous execution, and access overlap situations.

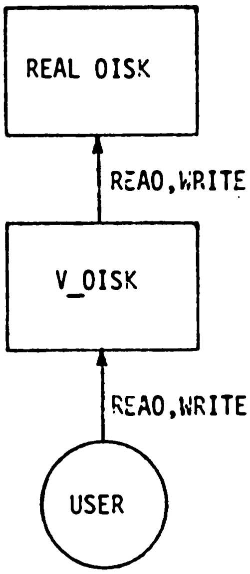
Two examples were used to demonstrate a resource's ability to repr sent temporary interactions. In Chapter 4 (Figure 4.4) we gave a soluti for the virtual printer problem. The key to that problem was to get th resource to transmit all accumulated output lines to the printer when tl

---

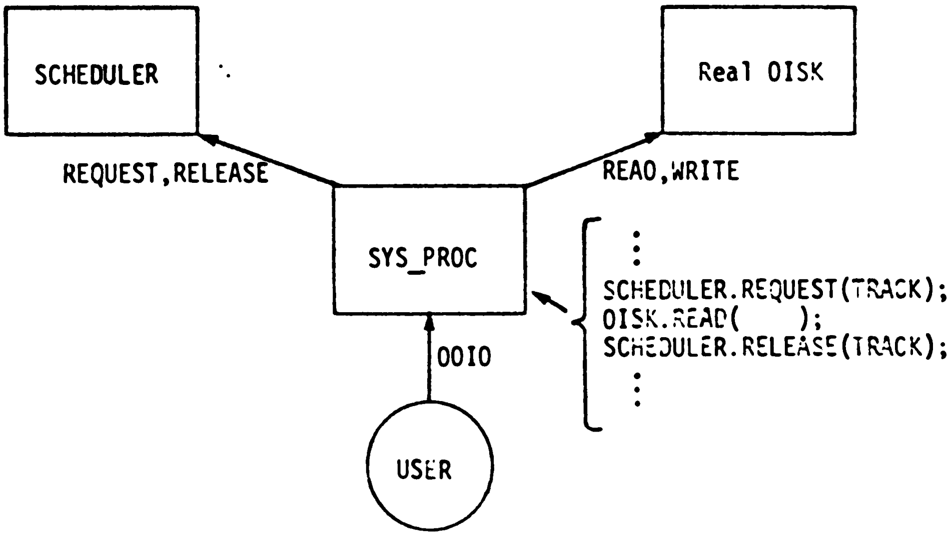
[2] The differences between await and WAIT/SIGNAL are unimportant at this level of consideration.

user had completed execution. The other example was the file system problem discussed earlier in this chapter. There, each file was assumed to have a limited life-time. A critical part of each solution was the automatic POST_OP procedure. In the virtual printer resource it was responsible for transmitting the data, and in the file resource it made sure that all buffer space associated with a dying file was returned. In equivalent monitor solutions, these two actions are the most difficult to accomplish. Since monitors have no feature comparable to POST_OP, a system process must trigger the necessary actions (assuming we cannot trust the user to do it himself). Our approach solves the problem in a straightforward manner without any of this overhead.

The second area where monitors are weak is in representing multi-level interactions involving simultaneous execution. We have two examples showing a resource's superiority: the disk scheduling problem and the file system (using multiple buffers for holding blocks in core), both solved in this chapter. The key to the improvement is the simpler design structure, demonstrated in Figure 7.14. Part A shows the simple approach possible with resources. Monitors require that the scheduler and disk not be nested so they can execute simultaneously. In order to insure proper use, an extra system procedure is inserted. This design is outlined in Part B. This organization of monitors is necessary any time simultaneous execution is required (including the alternate of the file system). Resources are able to avoid this weakness because of the <u>release</u> clause. It permits interaction systems to be designed in levels, without fear of having to later reorganize and redefine (notice that the scheduler in the



Part A: Structure with resources



Part B: Structure with monitors

Figure 7.14: Comparison of resources and monitors for disk scheduling.

monitor solution has a different operational structure from the resource one). We feel that this is a very significant improvement.

The final area of interaction analysis is access overlap. Our disk scheduling solution which combined the components into one resource is one example where access overlap is permitted. In this case two processes could execute simultaneously inside DISK, but we know that they will not attempt to use any of the same variables--so the overlap is only with respect to the resource. In Chapter 4 (Figure 4.9) we solved a version of the readers/writers problem which involved more overlap; processes were permitted to simultaneously read the same variables. Monitors cannot permit either of these overlaps because of their strong rules for enforcing data integrity (e.g., only one process at a time may execute in a resource). Our parallel option permits some safe access overlaps (like the two we mentioned) without losing data integrity.

Unfortunately, parallel cannot permit some safe forms of access overlap. For example, consider a resource for implementing a linked-list with operations SEARCH, INSERT, and DELETE. If coded properly, any number of SEARCHES and one INSERT could execute safely together. However, such a system cannot pass our interference-free constraint for parallel since the INSERT could modify a link while a SEARCH is using it. Parallel is limited to allowing access overlaps which do not depend on proper coding of an algorithm for their safety. In the linked-list case, a simple reordering of statements in INSERT would allow simultaneous reading and writing on a variable, when it was not possible before. In order to accurately distinguish these cases, some form of program verification is necessary (a feature not associated with parallel).

We summarize this analysis of expressiveness with the same chart that was used in Chapter 2; this time adding our resource feature. The results are shown in Figure 7.15. We feel that the improvements indicated are strong evidence of the superior expressive power of resources.

7.5.2. Access Controls

Our system for providing access controls can be cleanly divided into two cases--static and dynamic. The default scope rules and the restrict pseudo-statement enforce all static rights, and capabilities and pointers provide enforcement of dynamic rights. Both of these systems have significant improvements in expressiveness over existing mechanisms.

The fundamental static access control requirement is to permit a programmer to specify which blocks can access various objects. Both access authorization and prevention should be expressed as simply as possible. The disk scheduling example (Figure 7.1) shows how restrict can accomplish them both easily. With one statement the virtual disk is able to invoke the real one, and all users (and all other blocks) are unable to call it. Moreover, implementation of this mechanism does not require any run-time overhead.

Another important form of static access control is selective entry--different blocks may need to be able to use different operations of a resource. We have shown three examples of this situation--(1) the message passing system in Figure 5.6, (2) the second version of disk scheduling (Figure 7.3), and (3) the final design for terminal processing (Figure 7.13). The second and third ones are particularly significant because the solutions strongly depend on proper enforcement of selective entry rights. In the disk scheduling, the SUPERVISOR must receive messages from two

Partition groups	example	message passing	monitors	modules	resources
Lifetime					
permanent	message passing	Good	Good	Good	Good
	disk resource	Poor	Good	Good	Good
temporary	virtual printer	Poor	Poor	Poor	Good
Interaction structure					
one-level	shared data file	Poor	Good	Good	Good
multi-level mutual exclusion	message system with buffers	Poor	Good	Poor	Good
multi-level simultaneous execution	disk scheduling	Poor	Poor	Poor	Good
Internal overlap					
no overlap	message passing	Good	Good	Good	Good
overlap reads	readers/writers	*	*	*	Good
general access overlap	list manipulation	*	*	*	*

* impossible

Figure 7.15: Expressiveness of advanced interaction mechanisms.

different sources (a system DRIVER process, and potentially untrustworthy users). The individual control over access to resource operations prohibits a user from masquerading as a DRIVER. Similarly in the terminal system, various blocks have different rights to the TERMINAL_ALLOCATOR. Users may invoke ACQUIRE and RELEASE, but only the systems process may use OPEN and CLOSE which set up and break actual communication links with external devices.

The final static access problem is to permit a user to access an object without being able to create other instances of its type. The only example we have covering this situation is the static remote terminal system described in Figure 5.5. There, a type definition was used to define the structure of an interactive terminal. Instances were then defined to represent actual hardware. Users needed to be able to access the device, but only through the instances defined by the system. A <u>restrict</u> clause applied to the type definition solved the problem completely.

All of our solutions for static control center on the use of <u>restrict</u>, which demonstrates more flexibility than the systems used by Concurrent Pascal and Modula. The parameter passing technique (of Concurrent Pascal) cannot handle selective rights or use without create situations at all. The USE-list mechanism in MODULA, although reasonably close to <u>restrict</u>, has an important conceptual difference. It implies that a block can specify which objects it will use, even though they are defined in an outer block's scope. Our protection problems actually need the opposite policy. We want to be able to dictate to a block which objects he can use. As a result, the USE-list mechanism is weak for enforcing policy while the restrict clause gives us exactly the control we want.

The last part of our expressiveness analysis is on dynamic access controls--capabilities and pointers. Again here, we concentrate on the applications which are difficult to handle with existing tools (mainly the manager concept [57]). Our strength is in our ability to permit a wider range of policies while imposing fewer restrictions on a system design.

Selective entry rights is one policy application which can be enforced in our system. The file system example (Figures 7.6-7.9) shows how a user can limit the access rights of other users to his files. One process may have full rights for a file (OPEN, CLOSE, READ, WRITE, COPY, and DESTROY), while another may have READ-only access. The interactive terminal processing example of this chapter (Figure 7.13) shows a different use of the same concept. The TERMINAL_BUFFER resource provides an interface between the USER and an actual device. In this problem it was important to be able to disconnect a USER without his permission. Our solution gives the system allocator the only rights for the disconnect operation. Without this selective entry control, a user could reconnect himself and defeat the system.

Another policy we permit is the use of rights without the ability to create. Going back to the interactive terminal processing example, the correctness of the system is based on prohibiting a user from creating his own terminal buffers. Were this protection absent, not only would overlapped usage of a terminal occur, but also a user could give his buffer to the allocator for redistribution. By keeping a copy of the access rights, he could later tap into some other process's line. Another example is the buffer allocation problem in Figure 7.5. We do not want the allocator to worry about getting back more buffers than it had originally. In both cases, the use without create policy is enforced by the static feature, restrict, which is only reasonable since the policy is to never allow others to create instances. So although the policy is being applied to dynamic features, the static access control is the appropriate tool.

Finally, our facilities permit a wide range of policies with respect to which language objects may "own" dynamic resources and what they can do with them. In the file system, each process defined and controlled its own file instances, while in the buffer allocator (and most other examples), a resource was responsible for creating and managing access to the dynamic objects. Either organization can be easily represented. Moreover, in both cases if a process had rights to an object, it could give those rights to others directly. Free movement of rights (e.g., a file owner giving access rights to his son) is permitted. Obviously this movement can be limited through the copy right for resources, but that is a program decision--not a language one.

The one major weakness in our system for access control is rights revocation. The file system solution showed how we could approximate it, but too much run-time checking by program code is necessary to make it work. Language extensions we have considered so far have not been able to eliminate this problem satisfactorily. They greatly complicate the system (e.g., linking a capability's current contents back to the source) and require extensive run-time support in order to function properly (e.g., chaining a capability back to the initial one before accessing the object). We simply have no good solution here yet.

The major difference between our approach and managers is in flexibility. Managers are a high level feature for dynamic access control, and as a result they impose more policy than our capability system. For example, managers do not allow selective entry control over monitors. While nothing in their definition actually precludes adding this feature, it has not as yet been done. Another example is that managers are the only units which can create and control granting of rights to processes. If a particular problem fits this policy, managers work fine. But some of our examples do not fit it, such as users creating their own files, and file sharing via copying or giving away rights. In these cases the system design must be reorganized to compensate for the policies imposed by managers. No such action was necessary with capabilities and pointers. Their simplicity makes it easier to represent different access policies.

We summarize the results of this analysis with a chart (Figure 7.16). While we have not been able to meet all of our goals, we feel that the improvements are significant.

			PL/I	Concurrent Pascal	Modula	Restrict	Managers	Capabilities
Language-enforced data integrity			No	Yes	No	Yes	Yes	Yes
Static Access Rights	no access	block real disk	Poor	Good	Fair	Good		
	full access	file system	Good	Poor	Good	Good		
	selective-entry access	receive-only messages	Poor	Poor	Fair	Good		
	use without create	remote terminals	Poor	No	Good	Good		
Dynamic Access Rights	full vs. no access	virtual disk					Good	Good
	selective entry	read-only file					Poor	Good
	use without create	remote terminals					Good	Good
	general rights sharing	file system					Poor	Good
	rights revocation	ownership situations					No	No
	monitors as managers	buffer pool					Good	Good
	processes as managers	garbage collector process					Poor	Good

Figure 7.16: Summary of access control mechanisms.

CHAPTER 8

ANALYSIS OF LANGUAGE AND PROGRAM PROPERTIES

So far our work has involved the development of expressive language features for process interaction and access control. In this chapter we analyze some interesting and relevant properties of the language and of programs written in the language. The first property we discuss is language-enforced data integrity, which has been one of our top priorities. We use informal arguments to show that the facilities we have defined do indeed enforce data integrity. The second property we examine is deadlock [34]. A significant amount of research [51,52,61] has been done on this topic and we apply it to our language. Although the results are not particularly good for our facilities, we are able to gain insight into the relation between deadlock and programming languages. Finally, we examine our ability to insure protection. We formulate the safety problem [30] in our language environment and show that it is solvable in this case. However, we also show that safety does not cover some important practical protection systems.

8.1. Data Integrity

Language-enforced data integrity is one of the absolute requirements that we placed on our design of programming facilities. In this section we show that this property does indeed hold true for the language. Since many of the restrictions concerning scope rules, parameter passing, and resource definitions are necessitated by it, we also discuss the types of violations that could occur if they were not present. So our purpose here is two-fold: (1) to show that our rules are sufficient for enforcing data integrity; and (2) to show that in some sense they are also necessary, given our demands for expressiveness.

The major difficulty in this analysis is to present a convincing "proof" of data integrity. We have not attempted to build up any formalism which could assist us, and all of our definitions and explanations about the meaning of the facilities are intended for potential programmers, not for program provers. Hence, we can claim only an informal, intuitive proof. A formal approach to data integrity is beyond the scope of this thesis, and probably beyond the scope of current techniques of language analysis.

We begin the analysis of data integrity by clarifying the term "simultaneous access." Each process executes in one program environment at a time. That environment may be changed by invoking procedures and resource operations or returning from them. Associated with each environment is the set of data variables that a process may currently use. Simultaneous access is possible if some variable is in more than one current environment. For data integrity, we must show that if variables are classified by read-only (R) or write (W) in each current environment, then all variables that appear in more than one environment are of type R.

8.1.1. Sufficiency of Facilities.

To prove that our language enforces data integrity, assume that we have some arbitrary, valid program written in the language. We can partition the explicitly declared[1] data variables of that program into three sets: (1) resource variables, (2) process variables, and (3) procedure variables. Each one is classified by the type of the surrounding block nearest to its declaration. An example of this partitioning is shown in Figure 8.1. Notice that resource variables include only those variables declared at the head of the resource, and not ones declared in the operation bodies. Also, all formal parameters (in this case W, X, Y, and Z) are associated with block types in which they are declared. Finally, we are only concerned with variables which represent data, so R1, P1, and PR1 are not involved in the partition but pointers are.[2]

Using this partition, we will show that a resource variable may be accessed by many processes, but simultaneous access to it can only occur if all such accesses are for reading. We will also show that each instance of a process and procedure variable can only be accessed by the process that is responsible for bringing it into existence. Hence for those variables all forms of simultaneous access are impossible.

The first case we examine is resource variables. Because of the interaction complexity introduced by parallel and release, we carry out this analysis in steps. In the first step we consider resources which use neither of these options and which also do not call any external procedures (or resources). In this case, all accesses to resource variables are within the resource block and only one process at a time may execute

---

[1]We leave out instances of protected variables now, since they are accessed indirectly. They are considered as a special case later.

[2]Pointer variables must be included since they provide indirect access to otherwise unprotected data variables. Here we are concerned with access to the pointer as opposed to the protected object.

```
MAIN: ┌ process;
      │
      │    var A: integer;
      │
      │       R1: ┌ static resource (W: integer);
      │           │      operations IN,OUT;
      │           │      var B: real;
      │           │
      │           │      IN: ┌ procedure (X: real);
      │           │          │      var C: Boolean;
      │           │          │   :
      │           │          │ end;
      │           │          └
      │           │       :
      │           │ end R1;
      │           └
      │       P1: ┌ procedure (Y: integer);
      │           │      var O: BUFFER pointer;
      │           │
      │           │ end P1;
      │           └
      │
      │      PR1: ┌ process (Z: real);
      │           │      var E: integer;
      │           │
      │           │      end PR1;
      │           └
      │         :
      └
```

Resource variables - W,B (for R1)

Process variables A (for MAIN); Z,E (for PR1)

Procedure variables - X,C (for IN); Y,O (for P1)

Figure 8.1: Sample partitioning of program variables.

there; hence, no simultaneous access is possible. Adding parallel permits simultaneous execution in the resource, but it only allows reading access overlap. We then show that calls out which are released, add no new accesses to resource variables so data integrity still holds. Finally, we consider calls out which are not released. Here new dynamic access paths are created, but we show they can be treated as if they were static accesses within the reosurce. So the conclusion is that simultaneous access to a resource variable can only occur when all processes are reading it. We now fill in the details supporting these arguments.

In the most restricted resource case, two points must be verified: (1) all accesses to resource variables are within the resource block, and (2) only one process at a time is executing within that block. By the definition of the language, a variable can only be accessed using a static right (direct use of the variable's name) or a dynamic right (indirect use through aliasing caused by parameter passing). From the scope rules we know that all static accesses to resource variables must be inside the resource. At this stage no dynamic accesses are possible since calls within a resource cannot pass them, and we are assuming no calls out. These facts imply that all accesses to resource variables are within the resource's block. The second part, mutually excluded use of a resource, comes almost directly from the resource definition. If several processes simultaneously invoke a resource's operations, only one is allowed to proceed; when he finishes another is selected. The only exception arises from the await statement. If a process is delayed via await, another process may execute a resource operation. However, since the delayed process cannot restart until it is safe to do so (in this case--when no one is using the resource), no execution overlap is possible. So we have shown that in the simple case, resource variables cannot be accessed simultaneously.

The addition of the parallel feature only changes a small portion of the previous analysis. We still know that all accesses to resource variables are static, and hence within the resource. The difference is that now several processes may simultaneously execute resource procedures. However, we have limits on the simultaneity. The parallel clause enumerates every type of overlap that the resource is to permit. The interference-free rules insure that each overlap can, at most, involve simultaneous reading of some resource variable. Hence, with the parallel clause all access overlap is guaranteed to be safe.

The next step in our analysis is to include calls out of a resource that involve the release mechanism. Only one minor change in the previous arguments is necessary to handle it. In the basic resource case we were able to conclude that no dynamic accesses could be made to a resource variable because internal calls could not pass them and external calls were prohibited. Now, the result still holds, but for a different reason. One constraint on release clauses is that in the affected calls, resource variables cannot be passed. So, although we can make calls out, we cannot create dynamic accesses to resource variable and hence the results are still valid.

To complete the analysis, we now consider calls out of the resource where release is not involved. In this case, resource variables can be passed so we must make some changes in our approach. The critical factor here is that, as far as the resource scheduling system is concerned, if a process calls out of resource procedure A to some procedure B (without release) the process is still considered to be executing in A. So, passing a resource variable out and then modifying it through dynamic access rights is the same as statically modifying it inside A. Since our interference-free algorithm treats all passing of resource variables as write-accesses (even though the procedure may only read it), we have effectively transformed the dynamic accesses into static ones which our previous analysis already handled.

We summarize the arguments we have used in Figure 8.2. The conclusion is that resource variables may be accessed simultaneously, but only if all accesses are for reading.

We must now establish data integrity for procedure and process variables. We have stated that each invocation of either type (procedure call or process create) gets a new copy of variables declared in the block. Our goal is to show that a process can only access those procedure and process variables which it created through its own execution. This statement then implies that process and procedure variables cannot have any form of simultaneous access. We prove our claim by giving an induction argument on the number of execution environments (procedure and resource calls) that a process enters.

The base step in the induction is that when a process is initially created, it can only access those variables that are in its reach. We know from the language that all accesses must be either static or dynamic. The scope rules prevent a process from having any static accesses to variables outside the process' block and also to any variables declared in any inner blocks (like processes or resources). So static accesses are language rule ⇓ only accesses are static and dynamic

scope rule ⇓ all static accesses are inside a resource

resource rule ⇓ no internal dynamic accesses

release rule ⇓ external dynamic accesses can be transformed into static ones

⇓

all accesses to resource variables can be treated as accesses within the resource

defn. of resource timing, await, parallel, and interference-free

⇓

multiple activity in resource is limited to read-only access overlap

⇓

resource variables may be simultaneously accessed, but only for reading

Figure 8.2: Summary of data integrity arguments for resource variables.

limited to variables it defined. Dynamic accesses can only be made to objects passed in as parameters (i.e., during process creation). But our parameter passing rules force all such objects to be passed by value--so the process cannot really access any objects dynamically. These arguments prove our basis for induction.

The induction step assumes that a process has called through some set of n environments where it cannot access any variables it did not declare. We prove that if the process changes to one more environment, the same condition still holds. Three types of environment changes are possible: (1) return to a previous one, (2) call of a resource, and (3) call of a procedure. In the first case, we know by induction that the previous environment was safe (i.e., the process could only access its own variables), and if no new dynamic accesses are set up, it still is safe. But on a return, only references to data variables that were passed in can be sent back. (Capabilities and pointers are passed by a pseudo value-result, so that does not create new access rights for the variable types with which we are concerned.) Hence, returning to a previous environment does not give a process any new access rights.

The second case is a process calling a resource. The new static rights acquired by the switch are to instances of the variables defined by the resource operation and to the resource variables accessible to the operation. However, we are only concerned with procedure and process variables (i.e., not resource ones) that it did <u>not</u> create. Hence the new static accesses cannot violate our condition. The only new dynamic access rights are to objects passed as parameters. But in that case, the objects must have been accessible in the previous environment and by induction we know that they all satisfied the access condition. Hence, resource calls cannot give a process access to data variables it did not create.

The final case is for a call to a procedure, and again we need to worry about the two types of access. The arguments about new dynamic accesses are exactly the same as in the resource case, so we do not repeat them here; the result is that no new dynamic accesses arise from parameter passing. The difficulty in this case is that some procedures can access variables declared outside their reach. (The accesses within the reach are all variable instances created by the call, so they do not affect our proof.) These external accesses are to variables not created by the call, so potentially they could belong to another process. However, the language imposes a sufficient restriction to insure that this is not true: a procedure may access variables outside its scope, only if it is invoked solely by the process in which the procedure is declared (i.e., the procedure must be defined within the range of the process that calls it). Figure 8.3 illustrates several applications of this rule. Since P1 and P3 access externally-defined variables, they may only be invoked by the processes which defined them, MAIN and SUB respectively. P2 does not access any global variables <u>and</u> it does not call P1, so it may be invoked by several processes (like SUB and SUB2). Going back to our proof, the scope rules limit how far out a procedure may go to access a variable--namely the nearest surrounding process block. But by the procedure call rule, this process block must be the one responsible for <u>all</u> calls on the procedure in question. Hence, the procedure is accessing variables which the process created at an earlier stage. So in summary, a procedure call can only introduce accesses to variables which the calling process

```
MAIN:  process;
          var A: integer;

       P1:    procedure;
                  A := A + 1;

       P2:    procedure;

       SUB:   process;
                  var B: integer;

              P3:    procedure;
                         B := B - 1;

              SUB2:  process;

                         Call P2;

                  Call P3;

           Call P1;
```

Figure 8.3: Examples of the rule for procedure access of externally-defined variables.

creates. This step completes our induction which shows that procedure and process variables are local to the process which creates them.

Protected variables are the only data type in the language not yet covered by our analysis. They are a special case because all accesses are accomplished by indirection through pointers. From the definition of operations on pointers (assignment and parameter passing) we know that at most one pointer may hold a reference to an instance of a protected variable. So simultaneous access via two different pointers is impossible. To complete our proof of data integrity we only need to show that two processes cannot use the same pointer at the same time (if one is modifying the protected variable). But notice that each pointer must fall into one of our three classes of variables: procedure, process, and resource. Our previous arguments show that if a pointer falls in either of the first two classes, only one process can ever use it. In order to properly handle the case of a pointer as a resource variable, all uses of it (whether dereferenced or not) must be treated as write accesses by the resource (i.e., for computing interference-free in parallel's). With this assurance, protected variables can never be simultaneously accessed and data integrity is complete.

8.1.2. Necessity of Rules

The previous subsection showed that the language rules are strong enough to enforce data integrity. We could also ask if the rules are stronger than need be. In many cases, the necessity of a rule can be observed from the sufficiency proof. For example, the interference-free constraint on parallel permits any access overlap that does not involve writing. If the test were any weaker, data integrity would not hold.

Unfortunately, the necessity of some other rules is not as clear. Since our major concern is the design of language facilities, we feel that the reasons behind our rules are as important as the rules themselves. In this section we examine three key rules whose necessity may not be completely obvious. They affect actions like:

1. exporting resource variables,
2. internal passing of resource variables, and
3. passing of resource variables on release calls.

The first rule is actually an implicit one--a resource cannot export one of its variables outside its scope. Such an option is available in Modula through the define-list feature which allows variables, types, and procedures to be visible outside of a module. Our comparable feature, the operations list, only allows procedure names to be exported. The reason for prohibiting the export of variables (even if it is a read-only export, as in Modula) is actually very simple. If one is exported, we never know when it is being used, so to insure data integrity we would have to assume it was always being used. Hence, the resource could never safely modify that variable. The important point is that exporting a read-only variable from a resource permits any number of readers and one writer to overlap access. This property makes the feature incompatible with data integrity.

The second rule is that procedure calls within a resource cannot pass resource variables as parameters. Notice first, that we do not lose any real programming power in the language by having it because all procedures have static rights to those variables so they do not need dynamic rights. The problem with internal parameter passing (of resource variables) is that it permits aliasing--the same variable can be accessed by several different names. Our algorithm for deciding if two procedures are interference-free (for parallel) requires knowledge of when each resource variable is being accessed. Similarly with release, we must detect any attempt to pass a resource variable out. If aliasing is permitted, particularly multi-level aliasing, both detection algorithms must analyze the calling structure in each resource. By prohibiting aliases, we have a much simpler detection scheme.

The final rule prohibits a resource from using release on a call with resource variables as arguments. A simple example (Figure 8.4) shows how data integrity could be violated if this rule were not present. Process ALPHA could execute INCR and modify VALUE while BETA is modifying it within the resource operation SUB. The logic behind this rule is that if some process has the ability to immediately use a resource variable (i.e., it need not encounter a delay like entering the resource), then it must acknowledge that ability by holding control of some procedure in the resource. In this way, we are able to accurately identify all potential simultaneous actions, and only permit those which do not violate data integrity.

8.2. Deadlock Analysis

We now turn our attention from language properties to properties of programs written in the language. The first one we examine is deadlock [34]. It is possible to write a program which has processes that become permanently delayed during execution. One simple example is a process that awaits a condition which never becomes true. Informally, we say th

```
MAIN:
    INCR:  procedure (A: integer);
               A := A + 1;
               end;

    RES:   static resource;
               operations ADD,SUB;
               release ADD to INCR;
               var VALUE: integer;

               ADD: procedure;
                        INCR(VALUE)*
                        end ADD;

               SUB: procedure;
                        VALUE := VALUE - 1
                        end SUB;
               .
               .
               .

    ALPHA: process;
               RES.ADD;

    BETA:  process;
               RES.SUB;
```

Figure 8.4: Example of data integrity violation--assuming starred statement is legal.

a program is deadlocked if any of its processes are permanently delayed.[3] In this section we discuss the problems of detecting and preventing deadlock in our language.

Previous research on this topic has led to some significant, but fairly general, results. From theoretical results, we know that for just about any useful multiprogramming language, it is impossible to construct one algorithm for detecting, avoiding, or preventing deadlock that works for all programs of that language. On the other hand, Holt uses a modelling approach, where in his somewhat restricted environment and with his definition of deadlock, those algorithms are possible to construct. Finally, Owicki and Gries [50,51,52] have studied deadlock from a program-proving viewpoint. They have defined sufficient conditions for preventing total system deadlock (all processes are unable to continue or have terminated).

Our purpose here is to apply this research to our programming environment. We begin by reviewing the problem framework and assumptions that led to each of the results just mentioned. We then discuss what they mean in terms of our language facilities. The net effect of this analysis is mostly negative with respect to our language: automatic deadlock detection and avoidance are too expensive and inaccurate, and prevention is very difficult to prove. However, we improve our understanding of deadlock by relating these results to each other in a single framework.

---

[3] The actual definition of deadlock, as made by Holt, is much more restrictive than this one. This difference will be clarified later. We use this broader definition to stress the general concept.

8.2.1. Undecidability Results

Some fairly old results from the theory of computations show that answering questions about deadlock is going to be a difficult task. Rather than actually deriving the results here, we will simply state them and reference previous work which proves them. We then briefly discuss what they mean in terms of our language environment.

The two important results relevant to our deadlock study can be informally stated as follows:

> Result 1: Given our language L, it is undecidable for an arbitrary program P, written in L, whether any processes in P have a possibility of being permanently blocked.
>
> Result 2: Given an arbitrary program P (written in L) which is executing with a current state S, it is undecidable whether any processes blocked in S will ever become unblocked after further execution.

The first result implies that it is impossible to write a deadlock prevention algorithm for our language which will work for every program (i.e., it will recognize only programs where deadlock could occur). The second one indicates the same problem with deadlock detection--we cannot write an algorithm which will detect in all cases the fact that some process is blocked and will never execute again. In each case the proof is based on a reduction to another unsolvable problem [35]. The details are not critical here, except to note that they do not depend heavily on any of our particular language features; similar results apply to almost every multiprogramming language.

These results are strong in a general sense, but they do not shut the door on all deadlock analysis. The key point is that we cannot find one algorithm that will work for all programs. However, it may be possible to prevent or detect deadlock for some useful subclass of parallel programs. We now look at two approaches to deadlock analysis that use this strategy.

8.2.2. Holt's Deadlock Analysis

Holt's [34] approach to deadlock is based on abstracting the problem from languages to a graph model. Within this model he defines deadlock (which is more restrictive than our informal definition) and develops efficient detection and prevention algorithms. An obvious question is-- what happened to the undecidability results we discussed? The answer becomes clear when we relate the model to our features. His definition does not include some situations which we would commonly call deadlock. His algorithms succeed because they only work on the more restrictive definition.

Holt's graph model represents the current state of a computing system with directed arcs and two types of nodes, process and resource.[4] Each process node represents a system component that can request and release units of some resource. Each resource node represents a type of object which can be allocated to a process (i.e., the node may hold several units). Arcs in the graph represent the current state of requests and allocations.

---

[4] Not to be confused with our language facilities.

In order to model computing system changes, Holt defines three types of graph transformations corresponding to the request, allocation, and release of resource units. He makes no provisions for changing the number of processes or types of resources during the modelling of a system--they must be known at the outset and are assumed to remain forever.

Deadlock in this model is based on a reduction algorithm that can be applied to any current state graph. The algorithm tries to find a sequence of valid transitions which will grant all outstanding requests. A state is deadlocked if and only if the reduction algorithm fails to find a valid sequence. In effect, the reduction algorithm is mimicking actions that the real system could take in the future. Herein lies the key difference between our informal definition of deadlock and Holt's restrictive one. Holt would say that a system is not deadlocked if a valid sequence of graph transformations exists, regardless of whether that sequence can actually occur in the real system. Hence, he could have permanently blocked processes, and yet his system will not be able to call it deadlocked. (For further details on this approach see [34]).

We can now relate Holt's approach back to our language system. Two basic weaknesses stand out:

1. Holt's assumption about a static environment avoids some realistic problems which would otherwise complicate the model; hence it is not often applicable; and
2. by ignoring all information about what a process can do (i.e., program code), the reduction algorithm may decide a process is not deadlocked even if in reality it has no chance of executing.

We demonstrate the first point by summarizing the problems associated with applying his model to our language. Most of the details are discussed in Appendix C. We illustrate the second point through a simple programming example. Some processes become permanently blocked, but the reduction algorithm does not detect it.

While many of our language features can be reasonably mapped into Holt's model, three of them are particularly difficult to handle: await, dynamic resources, and the parallel option. The problem with await is that it is most naturally represented as a consumable resource (but without the identification of a permanent producer). Unfortunately, all processes in our system only have temporary power to "produce units," while Holt's model assumes all producers have the right permanently. The second feature, dynamic resources, has a very obvious problem--all of Holt's resources must be permanent, whereas these are not. It is possible to write a program which, during execution, would be deadlocked except for the fact that a yet to be created resource provides a path for unblocking every process. Similarly, a program could become deadlocked because a dynamic resource died. These situations are difficult to recognize if all resources are assumed to be permanent. The parallel option creates problems because it permits a resource to be shared, whereas Holt's model requires exclusive allocation.

In Appendix C we discuss modifications of the model to eliminate these problems. The first two can be handled by adding a new "potential" access graph. However, the cost of the new reduction algorithm is so high as to be useless in practice (execution time is $O(E^P)$ where E is the number of access edges and P is the number of processes). Finally, we still have not found <u>any</u> reasonable way to extend Holt's model to handle <u>parallel</u>.

This analysis shows some of the problems with applying the model to a "real" language. We are not trying to claim that all our difficulties are the fault of his model--some of them probably fall on our language. However, we feel that the types of problems we have illustrate that the model is not applicable to practical situations.

The second major weakness with Holt's approach is that the reduction algorithm does not use any information about what each process might actually do in the future. Hence, in order to be sure that deadlock exists, it must assume that every active process will take any action it can in order to unblock each stalled process. The problem is that the future actions of a process could be significantly less than its full potential. If that happens, a system could have permanently blocked processes that are never detected as being deadlocked.

In Figure 8.5 we show an admittedly contrived example to demonstrate this point. The MAIN process defines a message system and two processes who will normally use it. ALPHA and BETA are coded so as to deadlock. The only unusual aspect is that MAIN sends one initial message. This one send forces the model to treat MAIN as a permanent producer. So, while ALPHA and BETA are permanently delayed, they are not recognized as being deadlocked because MAIN "could" send another message.

This type of problem is inherent in the graph modeling approach, and we feel that it is a severe limitation. In order to be absolutely certain that deadlock exists, the criteria for recognition must be very strict. Unfortunately many permanently blocked states fall short of the deadlock

```
MAIN:
   MES:   static resource;
             operations SEND,RECEIVE;
             :
             end;

   ALPHA: process;
             MES.RECEIVE(___);
             do forever;
                MES.RECEIVE(___);
                :
                MES.SEND(___);
                end
             end;

   BETA:  process;
             do forever;
                MES.RECEIVE(___);
                :
                MES.SEND(___);
                :
                end
             end;
   :
   MES.SEND(___);
   :
   do forever;
      /* no further messages sent */
      :
      end
   end MAIN
```

Figure 8.5: Example of permanent blocking that is not deadlock.

criteria--so the potential is high for having deadlock without recognizing it. From a practical standpoint, such inaccuracy is unacceptable.

8.2.3. Owicki's Deadlock Analysis

The most recent work on deadlock has been based on axiomatic techniques for proving properties of parallel programs [50,51,52,61]. This approach is exactly the opposite of the previous one in that analysis of the program code is essential. Actually, the deadlock work is only a special case of research on verification of parallel programs. Using the assertions developed as part of a program's proof, Owicki gives a sufficient condition that insures that deadlock cannot happen. In this section we briefly summarize the approach taken by Owicki and the relevant deadlock results. We then relate it to our language facilities discussing the meaning of the sufficiency condition in that environment.

The multiprogramming language used by Owicki is considerably simpler than ours. Multiprogramming is described through the cobegin-coend statement [7]. Each process is permitted to access global variables, but only within a modified version of the region statement [7]:

with r when B do S

where r is a shared variable, B a Boolean expression, and S a statement which uses r. When a process attempts to execute such a statement it is delayed until the condition B is true, and r is not being used by any other process (all uses of a shared variable must be inside a with-when statement). Conceptually, the mutual exclusions on S are similar to those enforced by resources in our language. Likewise, the Boolean test is closely related to our <u>await</u>; a process may be permanently delayed at one of these commands.

Within this language, Owicki examines the problem of preventing permanent blocking from ever happening in a particular program. She defines a program to be permanently blocked if and only if <u>all</u> processes which have not completed are blocked. This definition is considerably different from deadlock, where only a subset of the processes may be stopped and some processes may run forever. In an operating system which has permanent processes, like an input spooler, this difference in definition is important. If the spooler can run forever without being blocked, we cannot get permanent blocking (no matter what happens in the rest of the system).

Based on this definition and assertions about a program's correctness, Owicki develops several sets of sufficiency conditions for preventing permanent blocking. We examine the simplest one as an example.

<u>Theorem</u> (Owicki [51]): Suppose program S contains the statement:

$S' =$ <u>resource</u> r; <u>cobegin</u> S_1 // ... // S_N <u>coend</u>.

Let the <u>with</u>-<u>when</u> statements of process S_K be:

$S_K^j =$ <u>with</u> r <u>when</u> B_K^j <u>do</u> T_K^j, $1 \le j \le n_K$.

Let $pre(S_K^j)$ be the precondition (in the proof {P} S {Q}) of statement S_K^j and let $I(r)$ be the invariant for resource r.

$$01 = \bigwedge_K (post(S_K^j) \vee (\bigvee_j(\neg B_K^j \wedge pre(S_K^j)))) \text{ and}$$

$$02 = \bigvee_K \bigvee_j (\neg B_K^j \wedge pre(S_K^j))$$

Then if $01 \wedge 02 \wedge I(r) \rightarrow false$, S cannot be permanently blocked if P is true when execution begins.

In simple terms, what does this theorem mean? Boolean expression D1 describes an invariant if all processes in S are unable to execute. Each process must either be done (i.e., post(S_K) is true) or it must be stopped at one of its with-when operations (i.e., pre(S_K^j) $\wedge \neg B_K^j$ is true). D2 is true only if at least one process is blocked at a with-when. Conceptually, if D1 $\wedge$ D2 $\rightarrow$ false then permanent blocking is impossible (using Cwicki's definition). However, these two expressions (by themselves) can never contradict each other because in the axiomatic proof technique each B_K^j, pre(S_K^j), and post(S_K^j) can only use variables local to process K; nothing relates these variables to each other. Invariant I(r) is needed to bridge this gap between process variables; it describes a relationship between resource and process variables which holds when no process is using a resource. Hence, I(r) $\wedge$ D1 $\wedge$ D2 $\rightarrow$ false can be used to show an absence of permanent blocking.

The next relevant question is--can this theorem help us in practice? Unfortunately, we have several problems with trying to use it. First, we need to develop assertions about the execution of each program that we want to prove avoids permanent blocking. This step is almost certain to be very difficult and very expensive (we are not, however, trying to say that it should not be done). Next, we need to show that D1 $\wedge$ D2 $\wedge$ I(r) $\rightarrow$ false. Finally, since this condition only proves sufficiency, the program may indeed prevent permanent blocking but D1 $\wedge$ D2 $\wedge$ I(r) $\not\rightarrow$ false. So if the test fails after all of the work, we may not know why. Possibly by changing the proof and recomputing D1, D2, and I(r), absence of deadlock may be shown.

Another problem is caused by the fact that Owicki's theory was developed mainly for programs that terminate. For systems programs that contain permanent processes, the sufficiency condition is not strong enough to find a sub-set of potentially deadlocked processes (unless all permanent processes are in the sub-set).[5]

The worst problem for us, however, is that we do not have axioms for our language facilities--and it is not clear how to derive them. While Owicki's facilities have a resemblance to ours, they are much simpler. The parallel and release facilities appear to be the most troublesome, and currently we know of no techniques for handling them.

8.2.4. Deadlock Summary

Our initial goal in this section was to discuss the possibilities for preventing, or at least detecting, deadlock in programs using our facilities. In this respect our conclusions are negative. Modelling along the lines of Holt is far too expensive to ever be practical. Moreover, in many cases the model could not detect permanent blocking. Although we did not mention it earlier, the model can also be used to define sufficient conditions for preventing deadlock. But these conditions are so restrictive (like no use of awaits) that any useful program could not meet them. Deadlock prevention via correctness proofs also has problems. The difficulty and uncertainty of getting the desired results currently make this approach infeasible. So for now, we have no good solutions for our language.

---

[5]Some recent research by van Lamsweerde and Sintzoff [61] may provide a solution to this problem.

However, this analysis is valuable in a more general sense since it clarifies the problems and suggests possible future work on deadlock. The undecidability results are a bad omen in that completely general solutions are impossible. But they do not hamper techniques for handling special cases.[6] The modelling approach looks very good in simple situations. However, we have tried to show that extending the model to handle complex cases is not feasible. More important, completely ignoring program code leads to too many inaccuracies for practical use. It is this point that strongly favors the axiomatic approach in the long run. Intuitively, we probably never write a program that is supposed to deadlock; however, with objects like disk schedulers we may come close. Prevention of deadlock will depend on the correctness of the scheduling algorithm. While the initial results from the axiomatic approach are not positive, it looks like our best hope for more useful results in the future.

8.3. Protection Analysis

One of the important responsibilities of an operating system is to guard against the misuse and misappropriation of objects by unauthorized users. Some of our new language facilities permit a designer to enforce his own policy decisions. But how does he know when his system is actually safe? In this section we restate this question in more concrete terms, and then discuss how our language facilities can help answer the question. We begin by briefly reviewing a paper by Harrison, Ruzzo, and Ullman [30] which presents a formal model for protection systems and

---

[6] One good example is the rule for ordering all requests and releases of resources based on a fixed hierarchy. For reusable resource systems, this rule prevents deadlock.

introduces the safety problem. We map their formulation into our language environment and outline a simple algorithm for deciding safety which could be useful to a system designer.

8.3.1. A Model of Protection

The protection model developed by Harrison, Ruzzo, and Ullman has five basic parts: objects, subjects, generic rights, an access matrix, and a set of commands. The first four are related in the obvious way. The access matrix P has a row for each subject and a column for each object. P(S,O) is a subset of the generic rights, R, which subject S possesses for object O. The commands define the ways in which the access matrix may be changed. They have the following form:

$$
\begin{array}{l}
\underline{\text{command}}\ \alpha\ (X_1,\ldots,X_K) \\
\quad \underline{\text{if}}\ r_1\ \underline{\text{in}}\ P(X_{S_1},X_{O_1})\ \text{and} \\
\qquad \vdots \\
\qquad r_m\ \underline{\text{in}}\ P(X_{S_m},X_{O_m}) \\
\qquad \underline{\text{then}}\ op_1 \\
\qquad\quad \vdots \\
\qquad\quad op_n\quad \underline{\text{end}}
\end{array}
$$

Each command is passed some subjects and objects which it can use to test the current state of the access matrix. If the conditions are satisfied, the matrix is modified by the operations which may add or remove generic rights and create or destroy subjects and objects. In a specific protection system, both the generic rights and the commands must be finite and fixed throughout execution. Thus, the protection state of a computation is represented by the access matrix and the only valid transitions are defined by the commands.

In this model, the safety problem [30] can be loosely formulated in the following way. Given a particular protection system (i.e., generic rights and commands) and some current access matrix with subject S about to enter r in P(S',O), can r be subsequently entered somewhere new? If not the system is safe for that right. In simple terms, we want to know the effect of adding some new right over and above that which was possible before. Can future valid transitions give this right to a new subject? In order to avoid having a trivial "unsafe" answer,[7] we must ignore actions taken by trusted subjects. Hence, the granting of a right is safe if for all future transitions (excluding those taken by trusted subjects) the rights is only accessible to those subjects who have it in the current state.

It should be clear from this discussion that knowledge about each subject is limited. We must know their potential actions with respect to the protection state (i.e., commands), but we assume these actions can take place in any order (i.e., actual program code is ignored). In that sense, this model is similar in approach and limitations to Holt's deadlock model. If none of the transitions from a state show a leak of rights, we can be certain there is none. On the other hand, even if a transition indicates a potential leak, we cannot be sure that it will actually happen.

One result from the Harrison, Ruzzo, and Ullman paper is somewhat surprising. It is undecidable whether a given configuration of a given

[7]Since S could confer r to many other subjects directly.

protection system is safe for a given generic right. Simply stated, this means that one safety algorithm cannot work for all possible protection systems that can be cast in this model. As the authors point out, this does not preclude the decidability for a particular protection system.

8.3.2. Safety and Our Language Facilities

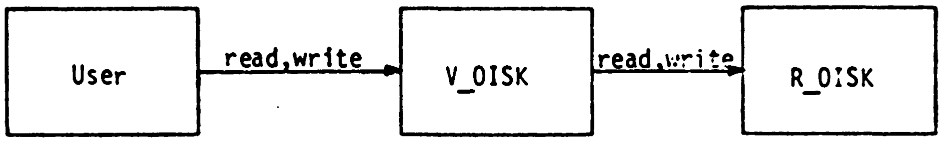
We now consider how the safety problem relates to our work. Our language can be viewed as defining a class of protection systems, each program being a specific instance. The language provides mechanisms for creating subjects and objects, and sets some constraints on the movement of access rights (e.g., scope rules and parameter passing rules). A program defines the generic rights (e.g., operations on resources) and command structure for actually using rights (e.g., use of restrict and capabilities). An obvious question is--can we decide safety for this class of protection systems? With the model of the problem which we shall now give, safety is decidable. However, we also show that some useful protection schemes are not covered by the safety property so we do not really have a complete solution.

The important objects of our system are fairly easy to identify. In a protection model, the concern is with items that can be shared among various subjects. The only shareable objects in our language are procedures (and resources), resource variables, and protected variables. All others are local to the process that declares them, so it is impossible to leak access rights for them.

The identification of subjects, on the other hand, is not quite so clear. Certainly each process in the system is a subject. But how do we treat a process when it calls out to an externally-defined procedure or resource? Is it the same subject with some new rights (e.g., access to resource variables) or is it somehow a new subject? Our problem can be illustrated with the disk scheduling system defined in Chapter 7. The solution was designed so that each user process calls a system-defined resource to service its request. While in the resource, the user has the ability to directly access the disk, but the code it is executing is a part of the operating system. If we treat a process as being the same subject regardless of the procedure it is executing, then users have access to the real disk. In particular, we cannot differentiate the acceptable case (our solution) from the dangerous one where a user bypasses the scheduler and gets direct access. For safety to have any useful meaning in our language, these two situations must be separable.

Our solution is to make each process, procedure, and resource block a separate subject. Then for each subject we need to know its direct access rights (i.e., which procedures it can call, and which variables it can access). Figure 8.6 demonstrates this approach with the disk scheduling solution. A user process only has call rights for the virtual disk, which in turn has rights for the real one. Clearly with this level of information, we can compute a process' absolute potential rights by taking an appropriate closure operation on the access matrix. But now we have the added knowledge that the user's only access path is through a safe system routine.

The next step in the analysis of safety is to enumerate all ways in which a subject can acquire direct access rights for an object. For three object types (resource variables, procedures, and static resources) access can only be granted through scope rules and hence those rights are perma-



Subjects \ Objects		User	V_DISK Read	V_DISK Write	R_DISK Read	R_DISK Write	V_DISK vars.	R_DISK vars.
User			Call	Call				
V_DISK	Read				Call		Use	
	Write					Call	Use	
R_DISK	Read							Use
	Write							Use

Basic access rights

Subjects \ Objects		User	V_DISK Read	V_DISK Write	R_DISK Read	R_DISK Write	V_DISK vars.	R_DISK vars.
User			Call	Call	Call	Call	Use	Use
V_DISK	Read				Call		Use	Use
	Write					Call	Use	Use
R_DISK	Read							Use
	Write							Use

Absolute access rights (closure)

Figure 8.6: Access rights system.

nent. Either a subject has access to them or it does not. Clearly, if these were the only objects in the system, safety would be easily decidable. No actions could change the protection state, so everything must be safe.

The remaining two object types, dynamic resources and protected variables, are the ones where access is granted dynamically. In order to answer the safety question for these objects, we need to know the maximum set of rights each subject could acquire. Fortunately, the capability scheme for representing them makes the analysis easy to accomplish. The operations on capabilities (assignment and parameter passing) define the only ways in which rights can move, and we can use this to compute a maximum flow of access rights for any program. Hence, we can determine all of the objects that each subject could potentially acquire.

Three observations about our language are relevant to computing the flow of access rights:

1. Each capability variable is accessible to either one process or one resource. Hence the maximum potential access of each subject can be determined by examining the flow of rights into each of its capabilities.
2. All methods for acquiring dynamic rights are explicit in the language (namely via create, assignment, and parameter passing).
3. When an object is created, full rights for it are given to one and only one capability.

These observations show that given sufficient information on a program's use of capability variables, we can compute a maximum potential flow of rights between each pair. Then to determine the potential use of a particular dynamic object we only need to examine which capabilities (and hence which subjects) can receive access rights from the capability that initially references the object. Similarly, if we want to know the effect of giving some right to a new subject, we only need to look at the flow out of the capability variable which received the right.

Our algorithm for computing potential access rights movement for a particular program is based on graph manipulations. Each node in the graph represents an instance of a capability in the program. The directed arcs represent possible access right transfers between nodes. In addition, they are labelled to indicate the type of rights that could move. The algorithm has the following five steps:

1. For each capability variable C which is not a formal parameter, add a node C to the graph. If C is a formal parameter add a node C_i for each different place in the program where C could be passed rights.
2. For each call that passes a capability (between P and Q_i) add two arcs $P \xrightarrow{\text{all}} Q_i$ and $Q_i \xrightarrow{\text{all}} P$.
3. For each process and resource initialization that passes a capability (from P to Q_i) add arc $P \xrightarrow{\text{all}} Q_i$.
4. For each capability assignment Q := P{rights list}; add arc $P \rightarrow Q$ with the rights list as the label.
5. Compute the transitive closure of the graph constructed in the previous steps (i.e., for each path from A to B, add an arc $A \rightarrow B$, where the label is the intersection of all sets of labels in the path).

The first four steps set up the basic information on rights movement.

Rights can flow from A to B only if a path exists in the proper direction. The final step consolidates this data so path searches are not necessary to decide flow between pairs of nodes.

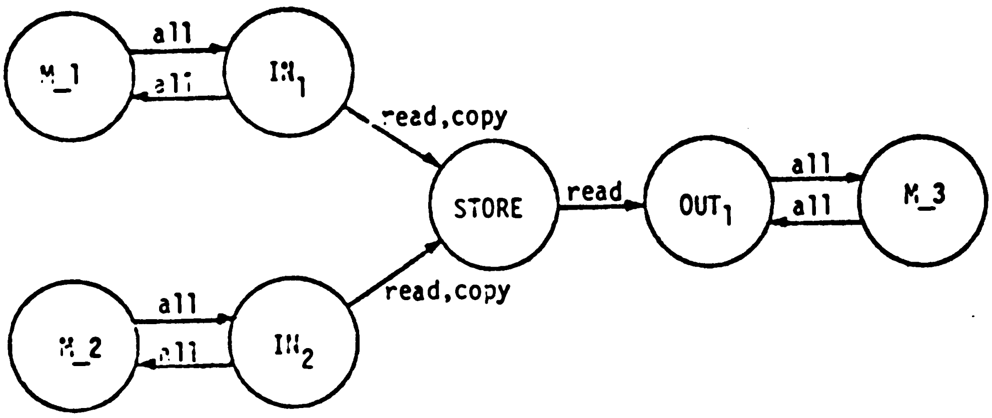
We illustrate this algorithm on a very simple system which passes messages. Figure 8.7 outlines code for a program that has three processes and one static resource. SENDER1 and SENDER2 transmit messages (instances of the dynamic resource MESSAGE) to RECEIVER through the static resource MESS. In Figure 8.8, part A shows the access graph before closure, and part B shows it after closure. The final graph verifies that access rights only flow from the two senders to the receiver. In particular, the senders cannot exchange rights. (Notice that we could not verify this if we had only one node for each formal parameter (i.e., IN); we must distinguish each call.)

8.3.3. Safety Summary

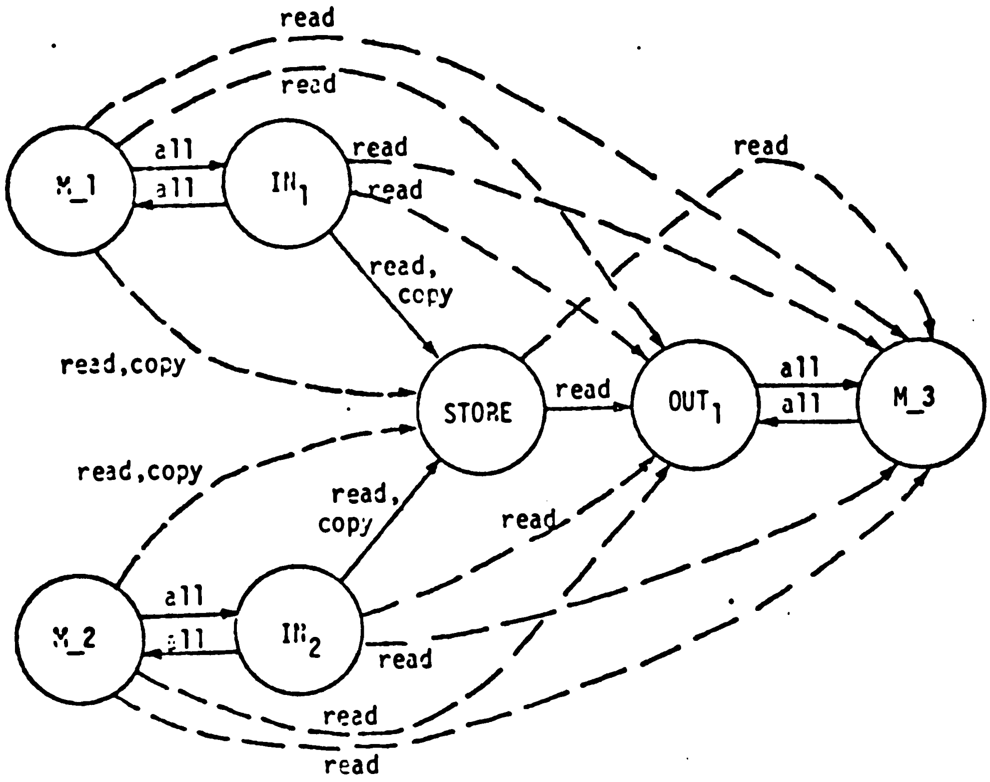
The graph algorithm just presented computes the maximum flow of dynamic rights to any capability. An identical algorithm can compute the same information for any pointer. Hence, any time a capability or pointer is referenced, we have a bound on which objects are actually be accessed. Furthermore, we can trivially compute the static access rights of each block in a program. Combining this information yields a bound on the direct access rights for every segment of a program. From a system designer's point of view, this data can be used to verify that his implementation of a protection policy is exactly what he thinks it is. Also, this data could be used to compute the total access rights of a process by taking one more closure operation, this time on the combined static and dynamic rights.

```
MAIN:  type MESSAGE =  dynamic resource;
                            operations READ,WRITE;
                            :
           var MESS:   static resource;
                            operations SEND,RECEIVE;
                            var STORE: MESSAGE capability;
                            SEND: procedure (IN: MESSAGE capability);
                                        :
                                        STORE := IN{READ,COPY};
                                        :
                                        end;
                            RECEIVE: procedure (OUT: MESSAGE capability);
                                        :
                                        OUT := STORE{READ};
                                        :
                                        end;
       restrict MESS.SEND to {SENDER1,SENDER2};
       restrict MESS.RECEIVE to {RECEIVER};
           SENDER1:    var M_1: MESSAGE capability;
                            :
                            M_1 := MESSAGE. create;
                            MESS.SEND(M_1);
                            :
           SENDER2:    var M_2: MESSAGE capability;
                            :
                            M_2 := MESSAGE. create;
                            MESS.SEND(M_2);
                            :
          RECEIVER:    var M_3: MESSAGE capability;
                            :
                            MESS.RECEIVE(M_3);
                            :
```

Figure 8.7: Outline of a two-sender message system.



A. Direct flows caused by assignments and parameter passing.



B. Closure of flows on graph in part A.

Figure 8.8: Capability flow graph for program in Figure 8.7.

The strength of this approach for our language facilities is that it is relatively inexpensive to compute the flow of rights. It can be done once, at compile-time, and then used for several types of analysis. Also the information on maximum flow could be very close to the actual flow, depending on the system design and required protection. For example, all of the flows in Figure 8.8 are almost certain to occur. Hence it could b a practical tool for a system designer to check his protection scheme.

The weaknesses of this approach goes back to the fundamental definition of safety, which ignores the meaning of program code. For an analysis such as we have developed, we must know a great deal about a subject' code--like which capabilities he uses to call the various resources. This detail almost demands that we be able to see the entire program, yet we d not use any information about the actual algorithms. Conceptually, our flow analysis assumes that a subject first acquires all rights and then gives away as much as possible. The program may actually do considerably less. Consider, for example, the message system we were analyzing--but with one more receiver process. The flow graph would show full transmission through resource MESS from both senders to both receivers, even if MESS were coded in such a way that each sender was actually tied to only one receiver. Our protection scheme cannot verify policies that are implemented by executable program code. The only answer we see for this problem is to develop techniques for formal program verification which include the ability to treat protection problems.

CHAPTER 9

CONCLUSIONS AND COMMENTS

This thesis has examined the problems of representing process interaction and access control in a high-level language. We now summarize this work and discuss directions for future research.

9.1. Summary

We explored these problems within the framework of writing an operating system which had user programs embedded as sub-blocks. One of our major concerns was to make this a viable approach--i.e., insure that a user could not use the language features to arbitrarily subvert its operating system. We identified three properties, language integrity, data integrity, and abstraction integrity, which we felt were necessary for adequate system protection. Our primary goal was to develop facilities for process interaction and access control which would permit an operating system designer to freely implement policies of his choosing. We termed this property "expressiveness," and gave a set of relatively objective criteria for defining it. We then showed that existing language facilities were inadequate for our purposes.

We proposed one new feature, the resource, for describing almost all forms of process interaction. It extends the monitor concept [33] in several useful directions. An optional POST_OP procedure allows a resource to automatically execute some code just prior to its destruction. This mechanism is complementary to the initialization available in both monitors and resources. The parallel and release clauses permit simultaneous use of a resource by several processes while preserving the basic soundness and understandability of monitors. We feel that these extensions, particularly the last two, are very useful for increasing the efficiency of and simplifying solutions to many process interaction problems.

Our other proposal for process interaction, protected variables, has a more limited scope of usefulness. Protected variables permit an efficient form of direct data sharing in the situation where only one process at a time has access rights to that data. We have found it useful for defining buffer pools shared among various processes.

Our access control mechanisms improve a system designer's ability to specify both static and dynamic access rights. The restrict pseudo-statement works in conjunction with basic scope rules to permit a program block's permanent access rights to be limited to those objects which it actually needs. We showed in Chapter 5 that this feature could be implemented without incurring any run-time overhead. The capability system provides the same type of protection, but for dynamic objects. Capabilities are associated with resource operations so run-time overhead, although very small, does exist. The examples in Chapter 7 showed the utility and expressive power of these mechanisms.

Finally, we analyzed some important properties that are intimately tied to the types of features under study. We informally argued that the language we defined enforces data integrity (i.e., if a process is modifying some program variable, then no other process may be simultaneously accessing it). This property is very useful because it eliminates a class of time-dependent errors that are difficult to find. The second property we examined was deadlock [34]. We were able to relate some previous research on deadlock to our facilities, but unfortunately we are still unable to determine how to prevent it or even detect it in some plausible situations. Models and proof-schemes that could help are still too expensive to be practical. The last property we discussed, safety [30], was related to our access control mechanisms. We showed that it is possible to compute a maximum set of access rights for each block in a program. However, the weakness was that the set may be larger than the accesses that are actually possible.

9.2. Future Research

Further research is required in this area in both practical and theoretical directions. On the practical side, an obvious next step is to implement the language features that we have already described. We need to determine the compile-time and run-time costs associated with resources (particularly <u>await</u>, <u>parallel</u>, and <u>release</u>), <u>restrict</u>, and capabilities. Without some implementation, we have no way of answering this question accurately. Also, an implementation would give us more freedom to try system designs and further evaluate our expressiveness criteria.

In a similar vein, some work still needs to be done to tune these facilities. As we indicated in Chapter 4, the <u>await</u> operation is conceptually good, but it is also very expensive. To make the language feasible, some compromise will probably be necessary. But how can we do it? Capabilities and pointers have a different problem. We refused to incorporate the ownership principle in the language, so true rights revocation is not possible. It could be useful to redesign that part and compare the relative merits of each. Redell's work [53] might provide a good starting point.

A third area of practical work is the design of facilities for parts of the problem framework which we have not yet examined in detail. For example, if users are treated as sub-blocks of an operating system then we must have some mechanism for adding and deleting blocks during execution.[1] This is not an easy feature to add if we want to maintain language integrity (i.e., users should not be able to use it to subvert the operating system). Another interesting but difficult addition is virtual memory management. In keeping with our separation of policy from mechanism, the operating system designer should have the freedom to choose page management policies. However, if memory management procedures can arbitrarily modify page mappings (e.g., indicate a page is present when it is not) then language integrity is lost. Some middle ground must be found.

Another useful area of work is applying this language to other types of parallel systems. We have restricted our view to the design of operating systems with the assumption that most of the interesting problems will arise there. Spooner [59] is currently using our language facilities to design a multi-user, data base management system. His initial results indicate no difficulties in this application. Other interesting applications include communication systems, computer networks, and real-time control systems.

On the theoretical side, our work in Chapter 8 has only scratched the surface of what needs to be done. We still need to develop axiomatic

[1] Obviously we could recompile the system for each user, but the expense would be unbearable.

or denotational semantics [23,47,60] for our language features. In [50], Owicki mentions the importance of applying her techniques to more practical languages. These facilities certainly are an excellent candidate for such a study because they were designed for expressiveness (and not simplicity of proof rules). The parallel and release clauses in resources as well as capabilities will probably be the most difficult to handle. We cannot ignore them, however, because they are critical to our language's expressive power. With either proof rules or denotational semantics we have a means of tackling the most important program property--correctness.

Our knowledge about other properties, such as deadlock, safety and information flow, could improve as a by-product of this work. So far, undecidability results have made it difficult for us to use data about a program's code to get accurate results about these properties. The approximation techniques of modelling can overestimate potential to the point of being useless. However, if we have a proof of correctness for a program, estimation may not be necessary. For example, we may be able to extend a correctness proof to show that deadlock cannot occur. The point is that all of these system properties are related to semantics. With the right semantic tools progress here is almost certain.

As a final comment, we feel that the major significance of this thesis is that it bridges the gap between practical and theoretical problems of parallel systems. Our programming facilities are sufficiently practical to be of genuine value in designing a real operating system. Likewise, they are sufficiently structured to be subject to rigorous analysis. The result is that each problem area can directly benefit from knowledge gained about the others.

APPENDIX A

SUMMARY OF THE LANGUAGE FEATURES

This appendix summarizes the important language features presented in the thesis. Five areas are covered:

1. processes,
2. resources,
3. static access controls,
4. capabilities and dynamic resources, and
5. protected variables and pointers.

In each area we give the proper syntax, the meaning of the features, and the rules of usage.

I. Processes.

A. Syntax:

```
               ( PROCESS_NAME: process (parameters);
               |                 declarations;
  declaration <                  statements
               (                 end PROCESS_NAME

               ( PROCESS_NAME. setstate (priority, slice, limit)
               | PROCESS_NAME. readstate (priority, slice, limit,
   operations <                           used, status)
               | PROCESS_NAME. activate
               ( PROCESS_NAME. suspend
```

B. Meaning:

A process block delimits code which can execute in parallel with other parts of the program. If it is defined in a type statement, each process instance receives its own set of variables (i.e., those it declares). The four process management operations are listed above. Setstate gives the process an execution priority, a time slice value, and a time limit. Readstate returns the current values for priority, time slice, time limit, time used, and statue (0 → active, 1 → suspended, 2 → terminated). A process begins execution only after it has received an initial state and has been activated. Processes are selected for execution in order of highest priority; ties are scheduled in round-robin fashion, each getting an amount of time equal to its time slice. When a process reaches its time limit it is suspended.

C. Rules of usage:

1. Statements in a process block may only use type names, static resources, and pure procedures declared outside the range of the block.
2. A process can only use the management operations on a process that it defines (i.e., not itself).
3. All parameters (data variables, pointers, and capabilities) must be passed into a process by value.
4. If a process creates other processes, it cannot terminate until all of its progeny have already done so.

II. Resources.

A. Syntax: R-NAME: $\left\{\frac{\text{static}}{\text{dynamic}}\right\}$ resource (parameters);

operations $OP_1, \ldots, OP_K$;

parallel* $(PR_J, \ldots, PR_L), \ldots, (PR_M, \ldots, PR_N)$;

release* PR_S to PR_T, ...,

PR_V to PR_W;

resource data variables;

OP_1: procedure ( ); ... end OP_1;

OP_K: procedure ( ); ... end OP_K;

local resource procedures;

PRE_OP*: procedure; ... end PRE_OP;

POST_OP*: procedure; ... end POST_OP

end R_NAME (* optional

B. Meaning:

A resource defines a form of process interaction. All procedures named in the operations list may be invoked outside the range of the resource. The default rule is that only one process at a time may execu within a resource. Parallel identifies groups of procedures (all within the resource) which may execute simultaneously (hence overriding the default). Release identifies calls from within the resource to procedur or operations outside, which give up control of the resource for the dur tion of the call. Within a resource, the primitive "await b" (b is any Boolean expression) is used to synchronize processes. A process can on' pass this statement when b is true and when the process can get control the resource (if b is false when it is encountered, the process loses resource control). The PRE_OP procedure executes automatically when a resource instance is defined. The POST_OP procedure executes automatically when the resource is about to die.

C. Rules of usage:

1. Statements in a resource may access the resource's variables and any static resources, types, and pure procedures defined outside.
2. A resource cannot define any processes or use any management operations.
3. Resources may be defined in types (dynamic resources must be defined this way), and each instance has independent variables and access synchronization.
4. PRE_OP and POST_OP cannot be named in the operations list, parallel, or release.
5. Calls within a resource cannot pass resource variables.
6. Calls identified in release cannot pass resource variables.
7. In parallel, all procedures within a group must be mutually interference-free with respect to resource variables (i.e., if one procedure writes to a variable--no other procedure can read or write it). All variables passed as parameters are treated as writes, and an array is treated as one variable (i.e., subscripts are not evaluated).
8. Static resources are accessed directly by an operation call, while dynamic resources are accessed indirectly using capabilities (see section IV).

III. Static access controls.

A. Syntax: restrict object_list to block list

individual static resource operations may be controlled with:

R_NAME{op list} as an object

B. Meaning:

All objects in the object list are to be accessible within the blocks specified. If a particular object is named in some restrict, and no restrict gives a particular block access to it, then that block cannot use that object.

C. Rules of usage:

1. Restrict may be applied to any data variable, capability, pointer, procedure, static resource, type, or process name.
2. It cannot be used to give a block access to an object which it could not otherwise use.
3. Blocks in block list must either be the current block, or blocks defined at the level of the restrict. If the current block is not named, then the reach of that block cannot use the objects.
4. In order for a block to use restrict, it must have access to all objects in the list (it is legal for a block to prevent itself from further use of an object).

IV. Capabilities and dynamic resources.

A. Syntax: declaration { C_NAME: R_NAME capability;

statements

- C_NAME := R_NAME. <u>create</u> (params)
- C_NAME.R_OPERATION (params)
- C_NAME_1 := C_NAME_2 (rights list)
- C_NAME := <u>null</u>

Boolean expressions

- C_NAME_1 = C_NAME_2
- (rights list) <u>in</u> C_NAME

B. Meaning:

A capability holds a reference and access rights for an instance of a dynamic resource. It is empty when initially defined. The <u>create</u> operation creates a new instance of the dynamic resource type, putting the reference and full access rights (all operations and COPY) in the capability. A dynamic resource operation is invoked using the dot notation, specifying the desired operation. Assignment copies the reference and specified rights into the left-side capability. The <u>null</u> operation empties a capability; the resource dies when no references for it exist. The equality test for capabilities is true if both reference the same object, regardless of rights. The <u>in</u> test is true if the capability contains a reference and all rights named in the list.

C. Rules for usage:

1. A capability is only accessible within the process or resource block that defines it.
2. A capability may be passed as a parameter (by value) to a process or resource at creation, and (by value-result) to a procedure or resource operation. Value passing assigns contents of actual parameter to the formal and nullifies the actual. Result passing assigns

the formal to actual, and nullifies the formal.

3. On assignment, the right side capability must have the COPY right and all rights in the list. Also, the left side capability must be empty.

4. For an operation call, the capability must have that right.

V. Protected variables and pointers.

A. Syntax:

declarations:

<u>type</u> V_NAME = <u>protected</u> <object type>

P_NAME: V_NAME <u>pointer</u>;

statements:

P_NAME := V_NAME. <u>create</u>

PNAME_1 ← PNAME_2

PNAME ← <u>destroy</u>

expressions:

PNAME ↑

<u>active</u> (PNAME)

B. Meaning:

A pointer holds a reference for an instance of a protected variable (the variable may be of any data type, record, pointer, or capability). The pointer is empty when initially declared. The create operation creates a new instance of the protected type, and puts the reference in the pointer. The left arrow operation transfers the reference from the right side pointer to the left one (i.e., the right side loses the reference). Destroy empties the pointer and destroys the object. The object referenced in a pointer is accessed using the up-arrow notation. Active is a Boolean operation which is true only if the pointer contains a reference.

C. Rules of usage:

1. A pointer is only accessible within the process or resource block that defines it.

2. A pointer may be passed as a parameter (by value) to a process or resource creation, and (by value-result) to a procedure or resource operation. Value and result passing are each equivalent to a transfer operation (in the appropriate direction) between the formal and actual parameters.

3. In the transfer operation, the left side pointer must be empty.

APPENDIX B

DETAILS FOR THE SOLUTION OF THE INTERACTIVE TERMINAL PROCESSING PROBLEM IN SECTION 7.4

In this appendix we present the details of the solution for our terminal processing problem. The organization is based on the descriptive layout shown in Figure 7.13. Figure B.1 displays the structure of the solution and outlines the static protection necessary for safe execution. The remaining figures, B.2-B.5, detail the code bodies of the various program blocks.

```
SYSTEM: program;
        TERMINAL_CONTROLLER: static resource;
              operations READ,WRITE;
              (body described in Figure 7.12)
              end;
        restrict TERMINAL_CONTROLLER(READ) to DRIVER;
        restrict TERMINAL_CONTROLLER(WRITE) to TERMINAL_BUFFER;
        type TERMINAL_BUFFER = dynamic resource;
              operations READ,WRITE,ENTER_DATA,
                         CONNECT,DISCONNECT;
              (body described in Figure B2)
              end;
        restrict TERMINAL_BUFFER to TERMINAL_ALLOCATOR,SYSTEM;
        type TERM_BUF = TERMINAL_BUFFER capability;
        TERMINAL_ALLOCATOR: static resource;
              operations ACQUIRE,RELEASE,START,END;
              (body described in Figure B.3)
              end;
        restrict TERMINAL_ALLOCATOR(ACQUIRE,RELEASE) to USER_1,...,USER_N;
        restrict TERMINAL_ALLOCATOR(START,END) to DRIVER;
        USER_i: process;
              (body described in Figure B.5)
              end;
        code to set-state and activate processes;
        end SYSTEM
```

Figure B.1: Skeleton program structure for interactive terminal processing system.

```
type TERMINAL_BUFFER = dynamic resource;

      operations READ,WRITE,ENTER_DATA,CONNECT,DISCONNECT;

      release WRITE to TERMINAL_CONTROLLER.WRITE;

      /* For simplicity, each TERMINAL_BUFFER will hold the last line   */
      /* input from a device.  A user may READ only when a line is      */
      /* present.  User WRITEs are passed on to the TERMINAL_CONTROLLER  */

      var MY_DEVICE: DEVICE_ID;
          NEXT_LINE: LINE;
          LINE_READY: Boolean;

      WRITE: procedure (INFO: OUTPUT_LINE);
          if MY_DEVICE = EMPTY then return;
          TERMINAL_CONTROLLER.WRITE(INFO,MY_DEVICE)
          end WRITE;

      READ: procedure (INFO: LINE);
          await LINE_READY | (MY_DEVICE = EMPTY);
          if MY_DEVICE = EMPTY then return;
          INFO := NEXT_LINE;
          LINE_READY := false
          end READ;

      ENTER_DATA: procedure (INFO: LINE);
          NEXT_LINE := INFO;
          LINE_READY := true
          end ENTER_DATA;
```

Figure B.2: Terminal Buffer resource (continued on next page).

```
CONNECT: procedure (DEVICE: DEVICE_IO);
    MY_DEVICE := DEVICE;
    LINE_READY := false
    end CONNECT;

DISCONNECT: procedure;
    MY_DEVICE := EMPTY
    end DISCONNECT;

PRE_OP: procedure;
    LINE_READY := false;
    MY_DEVICE := EMPTY
    end PRE_OP

end TERMINAL_BUFFER
```

Figure B.2 (cont.): Terminal Buffer resource.

```
TERMINAL_ALLOCATOR: static resource;

    operations OPEN,CLOSE,ACQUIRE,RELEASE;

    /* TERMINAL_ALLOCATOR is responsible for making and breaking       */
    /* communication links between user processes and remote           */
    /* terminals.  A terminal may request a link, which DRIVER will    */
    /* turn into a call on OPEN.  Similarly, a link is broken          */
    /* through a call on CLOSE.  User processes invoke ACQUIRE and     */
    /* RELEASE in order to complete and break (respectively) their     */
    /* part of the link.  A communication link is an instance of       */
    /* TERMINAL_BUFFER; links are established by handing out copies    */
    /* of the appropriate rights.                                      */

    type ACCESS_INFO = record
                       USER: PROCESS_IO;
                       ACCESS_COOE: array 1..N of char;
                       OEVICE: integer
                       end;

    var SECURITY: array 1..USER_COUNT of ACCESS_INFO;
        OEVICE_BUFFERS: array 1..TERM_COUNT of TERM_BUF;

    /* SECURITY associates an access code with each user               */
    /* OEVICE_BUFFERS contains master capabilities for all TERMINAL_   */
    /* BUFFER instances.                                               */

    OPEN: procedure (PASS: LINE, NEW_OEVICE: TERM_BUF,
                                 OEVICE: OEVICE_IO);
        Find password in the line PASS.
        Look up user record (J) in SECURITY associated with it.
        If another device is using that key, reject this request.
```

Figure B.3: Terminal allocator resource (continued on next page).

```
    Otherwise, find a currently free terminal buffer.
    Store index of appropriate capability in I.

    NEW_DEVICE := DEVICE_BUFFERS(I){ENTER_DATA};
    DEVICE_BUFFERS(I). CONNECT(DEVICE);
    SECURITY(J). DEV := I
    end OPEN;

CLOSE: procedure (OLD_DEVICE: TERM_BUF);

    Find information in SECURITY associated with this password,
    put entry in INDEX.

    OLD_DEVICE := null;
    DEVICE_BUFFERS(SECURITY(INDEX). DEV). DISCONNECT
    end CLOSE;

ACQUIRE: procedure (USER: PROCESS_ID, NEW_DEV: TERM_BUF);

    Find INDEX of user in SECURITY.
    with SECURITY(INDEX) do
         await DEV > 0;
         NEW_DEV := DEVICE_BUFFERS(DEV){READ,WRITE}
    end ACQUIRE;

RELEASE: procedure (OLD_DEV: TERM_BUF);

    Find INDEX in SECURITY for user with this capability.
    Show this capability as ready for reallocation (unless
    it has not yet been closed).
```

Figure B.3 (cont.): Terminal allocator resource (continued on next page).

```
        OLD_OEV := null;
        SECURITY(INDEX). OEV := D
        end RELEASE;

    PRE_OP: procedure;
        Initialize the SECURITY matrix;
        for I := 1 to TERM_COUNT do
            OEVICE_BUFFERS(I) := TERMINAL_BUFFER. create
        end PRE_OP

    end TERMINAL_ALLOCATOR
```

Figure B.3 (cont.): Terminal allocator resource.

```
DRIVER: process;
    var TERMS: array 1..TERM_COUNT of TERM_BUF;
        DEV: 1..TERM_COUNT;
        INPUT: LINE;

        /* TERMS holds a capability for a TERMINAL_BUFFER for each    */
        /* terminal in use--initially all are empty.                  */

        /* The DRIVER continuously processes input lines received     */
        /* from the controller.  If it is the first line from the     */
        /* device, OPEN a communication link.  If device is in use,   */
        /* ENTER_DATA in appropriate TERMINAL_BUFFER.  If line is an  */
        /* EOJ, CLOSE link.                                           */

          do forever;
             TERMINAL_CONTROLLER.READ(INPUT,DEV);
             if {ENTER_DATA} in TERMS(DEV))
                then
                   if INPUT = EOJ
                      then TERMINAL_ALLOCATOR.
                              CLOSE(TERMS(DEV));
                      else TERMS_DEV. ENTER_DATA(INPUT);
                else
                   TERMINAL_ALLOCATOR.
                      OPEN(INPUT,TERMS(DEV),DEV);
             end

        end DRIVER
```

Figure B.4: Driver process for handling all terminal input.

```
USER_i: process;
        var PERMIT: TERM_BUF;        /* capability for getting to device */
            INFO: LINE;
            OATA: OUTPUT_LINE;
        :
        TERMINAL_ALLOCATOR. ACQUIRE(PERMIT,MY_IO);
        :
        PERMIT. REAO(INFO);
        :
        PERMIT. WRITE(OATA);
        :
        TERMINAL_ALLOCATOR. RELEASE(PERMIT);
        :

        end USER_i
```

Figure B.5: Sample user block in interactive terminal processing.

APPENDIX C

EXTENSIONS TO HOLT'S DEADLOCK MODEL

In this appendix we discuss some of the extensions to Holt's deadlock model that are needed to accurately represent programs using our language features. We assume here that the reader is familiar with the basic model described in [34].

Several features of our language make it difficult to directly apply Holt's model. They include await, parallel, and dynamic resources. We first consider modelling the language without these features, and then discuss the changes required to add them.[1] Our basic resource facility can be modelled by a reusable resource node with one unit. If a process is allocated that unit, then he is currently executing within the resource. (Notice we only need one unit if the parallel feature is omitted.) Nested resource calls are handled in one of two ways, depending on whether or not the release clause is employed. If it is not used, the process simply holds the first resource, while a request for the second is made. If the nested call is named in release, the process returns the first unit before requesting the second. Similarly when that call returns, the unit representing the second resource is given back before the first unit is requested again.

---

[1]We also omit consideration of process controls such as SUSPEND and ACTIVATE. Further, we assume all processes will get sufficient time to execute. These features are not hard to handle, but they add uninteresting special cases.

The await feature can be added to the model by using consumable resources. Each different Boolean expression used in an await is treated as a consumable resource that never has any units available. If a process is delayed at an await, it releases the resource's reusable unit (i.e., the process is permitting another one to gain entry) and then requests the appropriate consumable resource node. When the process is restarted, the model grants that request, and also gives back the unit representing the resource (in the program).

The only problem here is representing the producer arcs for these consumable resources. Intuitively, any process that can call a resource and affect a condition is a producer. We can compute this information and include it in the model. Unfortunately, this approach does not reflect the true nature of our language. A process can only affect an await condition if it is inside the appropriate resource (i.e., this right to be a producer is dynamic, not static). Hence, the reusable resource representing control of a resource is logically connected to the consumable resources representing the awaits inside, but the model does not recognize it.

As a result, a program could be deadlocked, but the reduction would not recognize it. For example, consider a message passing resource which uses a buffer resource for acquiring storage space. If a process attempts to send a message and no buffer space is available for storing it, deadlock is possible. Figure C.1 illustrates one implementation where deadlock can happen.[2] If a process is delayed waiting for a buffer, it

---

[2]Other designs, particularly ones that use release on calls to the buffer resource can avoid deadlock.

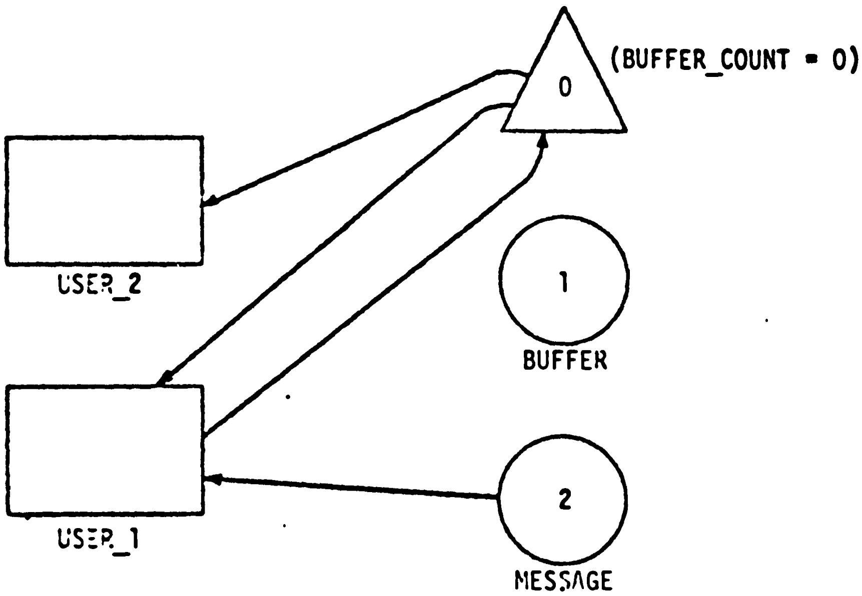
```
BUFFER: static resource;
        operations ACQUIRE,RELEASE;
        :
        ACQUIRE: procedure (   );
                await BUFFER_COUNT > 0;
                :
                end ACQUIRE;

        RELEASE: procedure (   );
                BUFFER_COUNT := BUFFER_COUNT + 1;
                :
                end RELEASE;
        :
        end BUFFER;

MESSAGE: static resource;
        operations SEND,RECEIVE;

        /* no parallel or release clauses */
        :
        SEND: procedure (   );
              BUFFER.ACQUIRE(   );
              :
              end SEND;

        RECEIVE: procedure (   );
                :
                BUFFER.RELEASE(   );
                end RECEIVE;
        :
        end MESSAGE
```

Figure C.1: Message system structure with potential deadlock.

still retains control of the message resource so no other process can get in to return one. However, if we diagram this situation using Holt's graph model (Figure C.2), no deadlock is apparent. The problem is that possession of MESSAGE by one user invalidates other users as producers of empty buffers, yet the model cannot recognize this. Unless all users are blocked on some other condition, Holt does not call this deadlock when in fact we are certain the delayed process will not execute again.

The addition of dynamic resources makes it much more difficult to decide when a process is deadlocked. The key to unravelling some set of blocked processes could be a resource that does not yet exist. Consider a file system design similar to our example in Chapter 7 where each user creates an instance of a dynamic resource to represent his file. Now suppose each file does some statistical reporting to a system process STAT, through a resource LOG. If no files exist at some time during execution, it may appear that STAT is deadlocked waiting for more data (see Figure C.3). However, we know it is not deadlocked unless all users that can create files are blocked or terminated.

It is possible to extend Holt's model to handle these situations. The approach is based on adding a second graph which represents the potential actions of each process. For example, it would include information like: process A can create an instance of type B, or process C can call an instance of type B. The potential access graph for the system in Figure C.3 is shown in Figure C.4. Notice that we can incorporate information about <u>awaits</u> more accurately here (and remove producer arcs from the current state graph). The new reduction algorithm combines the information from both graphs. Current state shows all unblocked processes,



USER_1 is waiting for a buffer, and simultaneously holding the message resource. USER_2 is a producer of buffers because sometimes he can call MESSAGE.RECEIVE which returns them. This situation is deadlocked, however, if all paths to BUFFER are through message.

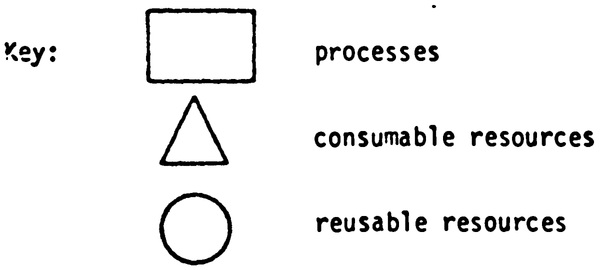
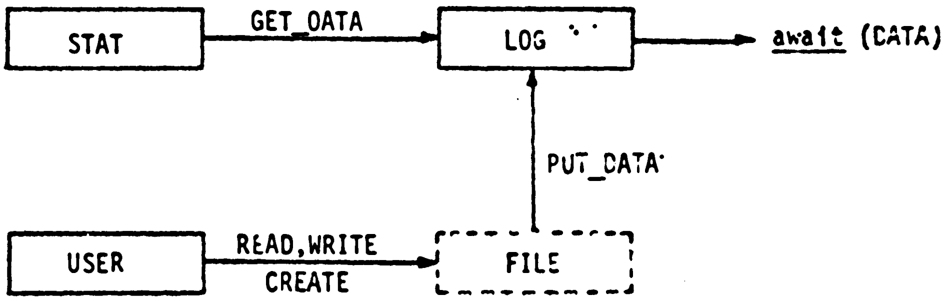
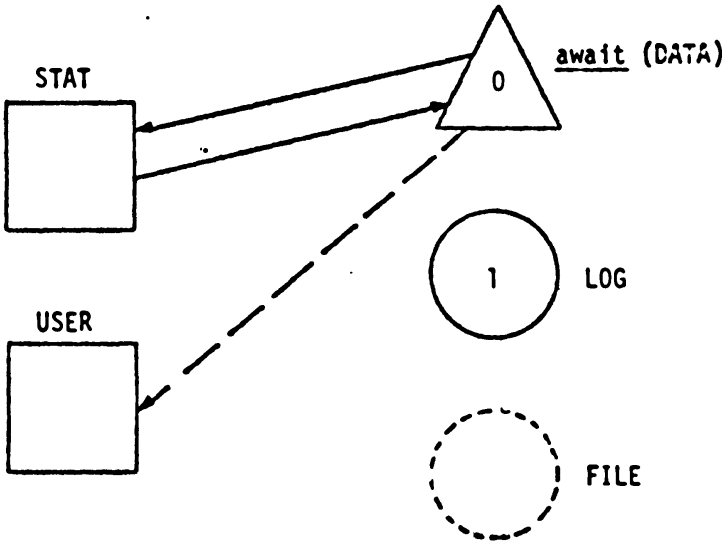


Figure C.2: Graph representation of a deadlock situation.

Program structure:



State graph:



(Dotted objects do not currently exist.)

Figure C.3: Deadlock detection problem with dynamic resources.

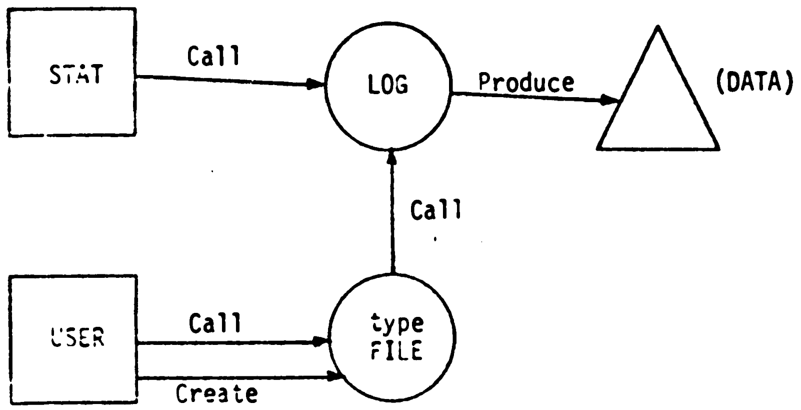


Figure C.4: Potential Access Graph for system in Figure C.3.

and the potential access graph specifies what each of them could possibly do. By cycling until no new processes are freed, deadlock is determined more accurately.

We have not completely described this extension to Holt's model for two reasons: (1) the details are messy, and (2) the reduction algorithm is too expensive to ever be useful. The high cost is in the use of the new graph. To find out what one process may do requires a search of the potential access graph--which takes time on the order of the number of edges. But in general, we may have to search the graph many times for the same process. This repetition could happen as many times as the number of processes in the system (intuitively, each process could establish one new resource in a critical chain). So although the extensions could detect deadlock more accurately, they are too expensive to be useful.

The remaining language feature, <u>parallel</u>, is even more difficult to handle. In the section on data integrity, we commented that with <u>parallel</u> the relevant concept is control of an operation--not control of the entire resource. However, our initial modelling scheme was based on the latter. The only solution we see is to treat each procedure (operation and internal procedure) of a resource (using <u>parallel</u>) as a separately acquirable reusable resource. But the reduction algorithm would then treat each one as an independent object, when in fact their allocations are interdependent. So we either get more inaccuracy (the reduction algorithm sees a free path when it does not exist) or we expand the model again with decision procedures for computing the dependencies (adding even more expense).

In conclusion, we can extend Holt's model to recognize deadlock situations for programs written using our facilities. The key extension is a potential access graph which works in conjunction with the current state graph to eliminate from consideration some reduction sequences that cannot possibly occur. Unfortunately, the cost of this new approach is too high to ever make it useful in a practical situation.

BIBLIOGRAPHY

1. Andrews, G. R., COPS--a protection mechanism for computer systems. Ph.D. Thesis, Department of Computer Science, University of Washington, July 1974.

2. Andrews, G. R., and McGraw, J. R., Language features for process interaction. Proc. ACM conf. on Language Design for Reliable Software, Raleigh, N.C., March 1977.

3. Atwood, J. W. (editor), Project SUE status report. Technical report CSRG-11 University of Toronto, April 1972.

4. Brinch Hansen, P. (editor), RC4000 software multiprogramming system. Regnecentralen, Copenhagen, Denmark, April 1969.

5. Brinch Hansen, P., A comparison of two synchronization concepts. Acta Informatica 1, 3 (1972), 190-199.

6. Brinch Hansen, P., Structured multiprogramming. Comm ACM 15, 7 (July 1972), 574-578.

7. Brinch Hansen, P., Operating Systems Principles. Prentice Hall, Englewood Cliffs, New Jersey, July 1973.

8. Brinch Hansen, P., Concurrent programming concepts. ACM Computing Surveys 5, 4 (Dec. 1973), 223-245.

9. Brinch Hansen, P., The programming language Concurrent Pascal. IEEE Transactions on Software Engineering Vol. SE-1, No. 2, June 1975, 199-207.

10. Brinch Hansen, P., The SOLO operating system: a Concurrent Pascal program. Software Practices and Experience 6 (1976), 141-149.

11. Brinch Hansen, P., Distributed actions--a concurrent programming concept. University of Southern California, March 1977 (submitted to Comm. ACM).

12. Campbell, R. H., and Habermann, N., The specification of process synchronization by path expressions. Lecture Notes in Computer Science 16, Springer-Verlag, 1974.

13. Clark, B. L., and Ham, F. J. B., The Project SUE language reference manual. Technical Report, CSRG-42, University of Toronto, Sept. 1974.

14. Corbato, F. J., and Vyssotsky, V. A., An introduction and overview of the MULTICS system. Proc. AFIPS Fall Joint Computer Conf., Nov. 1965, 185-196.

15. Courtois, P. J., Heymans, F., and Parnas, D. L., Concurrent control with readers and writers. <u>Comm ACM 14</u>, 10 (Oct. 1971), 667-668.

16. Dahl, O. J., Myhrhaug, B., and Nygaard, K., Simula 67--common base language. Norsk Regnesentral, Oslo, Norway, May 1968.

17. Dahl, O. J., Hierarchical program structures. <u>Structured Programming</u>, Academic Press, New York, 1972.

18. Denning, D. E., A lattice model of secure information flow. <u>Comm ACM 19</u>, 5 (May 1976), 236-243.

19. Dennis, J. B., and Van Horn, E. C., Programming semantics for multiprogrammed computations. <u>Comm ACM 9</u>, 3 (March 1966), 143-155.

20. Dijkstra, E. W., Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, 1965.

21. Dijkstra, E. W., The structure of THE operating system. <u>Comm ACM 11</u>, 5 (May 1968), 341-346.

22. Dijkstra, E. W., On the fly garbage collector: an exercise in cooperation. <u>Lecture Notes in Computer Science 46</u>, Springer-Verlag, 1975.

23. Donahue, J., Complementary definitions of programming language semantics. <u>Lecture Notes in Computer Science 42</u>, Springer-Verlag, 1976.

24. Fabry, R. S., The case for capability based computers. Proc. Fourth Symposium on Operating System Principles, Yorktown Hts., New York, 1973.

25. Feldman, J., A programming methodology for distributed computing. TR 9, Dept. of Computer Science, University of Rochester, 1976.

26. Graham, G. S., and Denning, P. J., Protection--principles and practice. AFIPS Conf. Proceedings (SJCC 1972), 417-429.

27. Gries, D., An exercise in proving parallel programs. <u>Lecture Notes in Computer Science 46</u>, Springer-Verlag, 1975.

28. Habermann, A. N., Synchronization of communicating processes. <u>Comm. ACM 15</u>, 3 (March 1972), 171-176.

29. [illegible], K. J., A fault-tolerant design for real-time control. <u>Computer Design 12</u>, 12 (Dec. 1973), 75-82.

30. Harrison, M. A., Ruzzo, W. L., and Ullman, J. D., Protection in operating systems. Comm ACM 19, 8 (Aug. 1976), 461-471.

31. Hoare, C. A. R., Towards a theory of parallel programs. International Seminar on Operating Systems Techniques, Belfast, Northern Ireland, Aug. 1971.

32. Hoare, C. A. R., and Wirth, N., An axiomatic definition of the programming language Pascal. Acta Informatica 2, 1973, 335-355.

33. Hoare, C. A. R., Monitors: an operating system structuring concept. Comm ACM 17, 10 (Oct. 1974), 549-557.

34. Holt, R. C., Some deadlock properties of computer systems. Computing Surveys 4, 3 (Sept. 1972), 179-196.

35. Hopcroft, J. E., and Ullman, J. D., Formal Languages and Their Relation to Automata. Addison-Wesley Publishing Co., 1969.

36. Howard, J. H., Proving monitors. Comm ACM 19, 5 (May 1976), 273-279.

37. IBM, PL/I (F) Language Reference Manual. Third edition, IBM Systems Reference Library, 1969.

38. Jensen, K., and Wirth, N. Pascal User Manual and Report. Springer-Verlag, Second edition, 1975.

39. Jones, A. K., Protection in programmed systems. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1973.

40. Jones, A. K., and Liskov, B., An access control facility for programming languages. Technical report, Dept. of Computer Science, Carnegie-Mellon University, May 1976.

41. Lampson, B. W., Dynamic protection structures, AFIPS conf. proceedings 35 (1969 FJCC), 27-38.

42. Lampson, B. W., and Sturgis, H. E., Reflections on operating systems design. Comm ACM 19, 5 (May 1976), 251-265.

43. Lampson, B. W., et al., Report on the programming language Euclid. Technical Report, Xerox Corporation, Aug. 1976.

44. Liskov, B., et al., Abstraction mechanisms in CLU. Comm ACM 20 (to appear).

45. London, T. B., The semantics of information flow. Ph.D. Thesis, Dept. of Computer Science, Cornell University, Dec. 1976.

46. McKeeman, W., Horning, J. J., and Wortman, D. B., A Compiler Generator. Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

47. Milne, R. E., and Strachey, C., A Theory of Programming Language Semantics. Chapman and Hall, London; Wiley, New York, 1977.

48. Organick, E. I., The MULTICS System: An Examination of Its Structure. MIT Press, Cambridge, Mass., 1972.

49. Organick, E. I., Computer Systems Organization: the B5700/6700 Series. Academic Press, New York, 1973.

50. Owicki, S. S., Axiomatic proof techniques for parallel programs. Ph.D. Thesis, Dept. of Computer Science, Cornell University, 1975.

51. Owicki, S. S., and Gries, D., Verifying properties of parallel programs: an axiomatic approach. Comm. ACM 19, 5 (May 1976), 279-285.

52. Owicki, S. S., and Gries, D., An axiomatic technique for parallel programs I. Acta Informatica 6, 1976, 319-340.

53. Redell, D. D., and Fabry, R. S., Selective revocation of capabilities. Colloques IRIA Protection in Operating Systems, Roquencourt, France, 1974.

54. Shaw, A. C., The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

55. Shaw, A. C., et al., A multiprogramming nucleus with dynamic resource facilities. Software Practices and Experience 5, 1975, 245-267.

56. Shaw, M., and Wulf, W. A., Global variables considered harmful. Sigplan Notices 8, 2 (Feb. 1973), 28-34.

57. Silberschatz, A., Kieburtz, R. B., and Bernstein, A., Extending Concurrent Pascal to allow dynamic resource management. Technical report #53, Dept. of Computer Science, State University of New York at Stony Brook, June 1976.

58. Spier, M. J., A model implementation for protective domains. International Journal of Computer and Information Sciences, Plenum Press, New York, June 1973.

59. Spooner, D. L., Master's thesis (in progress), Cornell University.

60. Tennent, R. D., The denotational semantics of programming languages. Comm ACM 19, 8 (Aug. 1976), 437-453.

61. van Lamsweerde, A., and Sintzoff, M., Formal derivation of strongly correct parallel programs. Technical report, MBLE Research Laboratory, Brussels, Belgium, 1977.

62. Wirth, N., The programming language Pascal. Acta Informatica 1, 1971, 35-63.

63. Wirth, N., Modula: a language for modular multiprogramming. Software Practices and Experience 7, 1 (Jan. 1977).

64. Wirth, N., The use of Modula and Design and implementation of Modula. Technical report #19, Institut für Informatik, ETH, June 1976.

65. Wulf, W. A., et al., Hydra: the kernal of a multiprocessor operating system. Comm ACM 17, 6 (June 1974), 337-345.

66. Wulf, W. A., Russell, O. B., and Habermann, N.; Bliss: a language for systems programming. Comm ACM 14, 12 (Dec. 1971), 780-790.

67. Wulf, W. A., London, R. L., and Shaw, M., Abstraction and verification in Alphard: Introduction to language and methodology. Technical reports, Carnegie-Mellon University, 1976.