Synchronization Mechanisms for Modular Programming Languages

by

Toby Bloom

January, 1979

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS74-21892.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

02139

This empty page was substituted for a blank page in the original document.

Synchronization Mechanisms for Modular Programming Languages

by

Toby Bloom

Submitted to the Department of Electrical Engineering and Computer Science

on January 26, 1979, in partial fulfillment of the requirements

for the degree of Master of Science

ABSTRACT

Any programming language that supports concurrency needs a synchronization construct with which to express access control for shared resources. This thesis examines synchronization constructs from the standpoint of language design for reliable software. The criteria a synchronization mechanism must satisfy to support construction of reliable, easily maintainable concurrent software are defined. Some of these criteria, such as expressive power, can be defined only with respect to the set of problems the mechanism is expected to handle. A definition of the range of problems considered to be synchronization problems is therefore needed. Such a definition is provided by describing the possible types of constraints that may be imposed on access to shared resources. We then use this taxonomy of synchronization constraints to develop techniques for evaluating how well synchronization constructs meet the criteria discussed. These techniques are then applied to three existing synchronization mechanisms: monitors, path expressions, and serializers. Evaluations are presented, and the three mechanisms compared.

Thesis Supervisor: Barbara H. Liskov

Title: Associate Professor of Electrical Engineering and Computer Science

Keywords: synchronization, concurrency, modularity, data abstractions, programming methodology

ACKNOWLEDGMENTS

I would like to express my appreciation to Professor Barbara Liskov, my thesis supervisor, for her guidance in my research, as well as for her patience and support while I was writing this thesis.

Craig Schaffert provided many useful insights and helped in clarifying many of the ideas in the thesis. Russ Atkinson and Mark Laventhal provided a great deal of encouragement and support, as well as technical help; without them I would have given up long ago.

I would especially like to thank Tim Anderson for reading the many drafts of this thesis, and for helping to put it into readable form. Bob Scheifler also helped edit the final draft.

Finally, I would like to thank my roommates, Margery Colten, Vicki Bier, and Dale Hattis, for giving me the encouragement I needed, and for putting up with me while I was writing this thesis.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant MCS74-21892.

CONTENTS

FIGURES

1. Introduction

1.1 Background and Motivation

In recent years there has been great interest in development of high-level language constructs to support parallel programming. Numerous synchronization constructs have been proposed since Dijkstra introduced the semaphore[12]. These include conditional critical regions[5], monitors[18,7], path expressions[8], and serializers[3].

In addition, we have come to realize the importance of the role programming languages play in the development of reliable, high quality software. Languages that support good program structure significantly enhance programmer effectiveness in producing reliable software. One methodology for improving software quality is the use of modular programming techniques and abstraction mechanisms. Languages such as CLU[25] and Alphard[35] support this methodology.

The need for reliable, easily maintainable software is even greater when concurrency is involved. Parallel programs are more complex and harder to understand than sequential ones because processes interact more, and time-dependent errors, which are not susceptible to traditional debugging techniques, are much more likely. It is therefore imperative that the language constructs used to implement parallelism support good program design.

While a synchronization mechanism that supports modular programming and the use of data abstractions would certainly contribute to the reliability and quality of concurrent software, no clear description of the requirements that such a mechanism must satisfy has been established. Attempts to evaluate existing synchronization mechanisms usually depend on the rather ad hoc technique of attempting to implement numerous synchronization schemes using the mechanism. Unfortunately, one can never tell, when using this method, whether the analysis is complete. If the analysis reveals a weakness in the mechanism, the construct is modified or extended to handle the one case found. The result has been the development of

numerous constructs, each designed to correct one flaw in a previous version, with no standard criteria for deciding when a mechanism is satisfactory.

The aim of this thesis is to state as explicitly as possible the criteria a synchronization mechanism must meet if it is to support construction of reliable, well-structured concurrent software. We will develop techniques to evaluate how well mechanisms satisfy these requirements. The criteria and evaluation techniques presented can then be used, not only to evaluate existing mechanisms, but as a basis for defining new mechanisms.

1.2 Research Goals and Outline of the Thesis

Our intention is to develop a methodology for evaluating the effectiveness of synchronization mechanisms in supporting the development of quality concurrent software. The first step in this process is to identify the function synchronization mechanisms serve in programming languages, that is, we must identify the class of problems to which these mechanisms will be applied. We accomplish this in Chapter 2 by developing a taxonomy of the synchronization constraints.

The first criterion we establish is that a mechanism be able to express straightforward solutions to any problem that can be defined in terms of the constraints described. A mechanism is said to have sufficient expressive power if it satisfies this property. Any construct designed to support reliability must satisfy certain other basic criteria also. These include ease of use, modifiability, modularity, and correctness. None of these has a precise definition, and we must decide how each applies to synchronization. In the remainder of Chapter 2, we define these criteria with respect to synchronization and develop techniques for assessing how well each is supported by a given construct.

In Chapters 3, 4 and 5, we examine three synchronization mechanisms: monitors, path expressions and serializers. The use of each mechanism is illustrated by a set of examples chosen to represent each class in our taxonomy of synchronization constraints. These examples

are then used in applying the evaluation techniques developed in Chapter 2. This analysis indicates whether a given mechanism satisfies our requirements and can be incorporated into a language designed to support software reliability without undermining the goals of that language. Furthermore, it provides information as to which problems can be easily implemented using a given mechanism.

1.3 Related Work

Most of the research directly related to this thesis has been mentioned in the previous sections. It falls into two basic categories: the development of synchronization constructs for high-level languages, and evaluations of these mechanisms.

The monitor construct was developed independently by Hoare[18] and Brinch Hansen[7] as an extension of Dijkstra's secretary concept[13].

The path expression mechanism was first developed by Habermann and Campbell[8], and has since been extended and modified several times [15, 14]. The mechanism is intended to provide a means of specifying synchronization non-procedurally, as a set of relationships among the operations used to access the shared resource. It thus appears to be a higher level construct than monitors.

Serializers are the most recent of the mechanisms discussed. They are based on the monitor mechanism and were developed by Atkinson and Hewitt[3] to eliminate certain characteristics of monitors that were thought to be detrimental to good program structure.

Several other, less extensive, proposals have been made to change specific features of monitors. Among these are the automatic signalling mechanism suggested by Kessler[23], and the manager construct of [21], which are aimed specifically at improving the signalling mechanism in monitors (see Chapter 3). These are not discussed in the thesis; serializers are a more extensive revision of the monitor construct and cover the changes made by these proposals.

Few papers exist on techniques for evaluation of synchronization mechanisms according to the criteria mentioned earlier. Andler[1] presents a comparison of semaphores, conditional critical regions, monitors and path expressions. The comparison is based on solutions to the bounded buffer problem, and focuses on correctness issues. While we are concerned with correctness, our interest is primarily in how well a mechanism supports construction of correct programs, rather than with proof techniques for the mechanism.

In [20], Howard has compared several versions of monitors. Howard is primarily interested in equivalence of internal specifications of the various versions, and does not address issues of expressibility or ease of use.

The work on the "nested monitor call" problem by Lister[28], and the responses to his initial presentation of the problem [29, 31, 22] are also relevant to our research. Further discussion of this work appears in later chapters.

A brief comparison of monitors with serializers appears in [3]. Some discussion of the differences between monitors and path expressions also appears in [15].

2. Criteria and Evaluation Techniques

In this chapter, we present the criteria to be used in the evaluation of synchronization mechanisms. Techniques for measuring how well these criteria are supported by various synchronization constructs are also presented.

Our principal concerns in this evaluation focus on programming methodology and the ways in which the addition of synchronization to a language influence software quality and reliability. We are therefore interested in such properties of synchronization mechanisms as expressive power, ease of use, modularity, modifiability, and correctness. These terms are sufficiently vague to make evaluation according to these criteria extremely difficult.

We will attempt to clarify the definitions of these properties with respect to synchronization. This chapter is divided into several sections. The first deals with modularity; it applies the concept of abstraction mechanisms to the problem of modularizing the implementation of shared resources. The following section is devoted to defining a method for classifying synchronization problems and describing the range of problems that synchronization mechanisms will be expected to satisfy. This classification of problems will be needed in later sections to describe techniques for evaluating expressive power, ease of use, and modifiability, since these properties are meaningful only with respect to a given set of problems. The final section discusses correctness, and the properties of a synchronization mechanism that influence how easily a program can be written correctly and how easily it may be proved correct. We will not actually discuss proof techniques.

Thus, this chapter is devoted to establishing the definitions and techniques necessary for evaluating how well synchronization mechanisms support production of reliable, high quality software. It is a first attempt at establishing some standard criteria for evaluating

properties long held to be very important for programming language constructs, but which have only intuitive, imprecise definitions.

2.1 Modularity

By modularizing programs, we limit the complexity the programmer must deal with at any given time, thus making it easier to write correct programs. The increase in complexity of software due to the presence of concurrency makes modularization essential for maintaining correctness. In this section we describe the ways in which software used to access or control access to shared resources should be modularized. This modularization is based primarily on the use of abstraction mechanisms[28].

There are two distinct modularity requirements for concurrent programs accessing shared resources. The first follows from the principle that the definition of an abstraction should be separated from its use. We consider a shared resource to be a data abstraction. The definition of the synchronization for a shared resource should be part of the definition of that resource, rather than being associated with each resource access. Thus our first modularity requirement is that users[1] see a shared resource abstraction that can be assumed to be properly synchronized. No synchronization code need be included in programs accessing the resource.

Our other modularity requirement has to do with the shared resource definition. Within the module that implements the shared resource, we have the definition of the structure and operations on the resource, as well as the definition of the synchronization scheme for the resource. These two parts actually serve different functions and should be separable into different subsidiary abstractions of the shared resource.

---

1. "User" of a resource refers to systems or applications software that accesses the resource.

We thus have a model of shared resources that consists of two levels of abstraction. At the higher level we have a protected resource abstraction with the operations that users may perform in accessing the resource. At the lower level, we have the resource abstraction, with the access operations that may be performed after a synchronizer ensures that access is safe, and a "synchronization abstraction", which contains state information necessary for synchronization, but not conceptually meaningful as a part of the resource, as well as synchronization operations.

In examining synchronization constructs, we will be attempting to determine whether they automatically provide this modularization, and if not, whether they allow the resource implementor to easily modularize the design in this manner.

2.2 Categorizing Synchronization Problems

As stated earlier, expressive power, ease of use, and modifiability can only be evaluated relative to a specific set of problems. A synchronization mechanism need only be powerful enough to easily express solutions to those problems we consider to be valid synchronization problems. We therefore need a way to describe the range of problems in which we are interested. In this section, we identify a set of properties of synchronization schemes by which we can classify these problems. We will later use the idea that, since synchronization schemes have various combinations of these properties, testing whether the mechanism can express schemes with each property, and whether it allows us to easily combine properties, will indicate the power and usability of the mechanism.

2.2.1 Categorization of Constraints

Synchronization mechanisms serve two main functions with respect to shared resources. One is excluding certain processes from the resource, under given circumstances; the other is scheduling access to the resource according to given priorities. Synchronization schemes are thus composed of a set of constraints, each having the form:

if condition **then** process A **is excluded from** the resource

or:

if condition **then** process A **has priority over** process B

We will refer to constraints of the first type as exclusion or concurrency constraints and the second as priority constraints. Within these two main classes, constraints differ in the kinds of information referred to in the conditional clause. The information that should be available to the synchronizer, and thus the information that can appear in constraints, falls into several categories:

1. the procedure(s) requested :[2]

   The resource is a data abstraction, so access to it is always obtained through operations of the resource type. In some synchronization schemes, the constraints depend on the operation requested. In stating, for instance, that readers of a data base have priority over writers, we are giving a constraint in terms of the types of procedures requested. In contrast, a strict first_come_first_serve ordering uses no information about the procedures requested.

2. the time at which requests were made:

---

2. We will often refer to this information as the "type" of the request.

Though it is rarely necessary to know exact times of requests, the time of a request relative to other events is often important. The most frequent use of time information is the determination of the order of requests. In addition, it is sometimes necessary to determine the synchronization state(see below) at the time of a request.

3. arguments passed with requests:

In many cases, the arguments passed with a request for resource access are needed to determine the order in which processes should be admitted to the resource.

4. the "synchronization state" of the resource:

Synchronization state includes all local data and state information needed only for synchronization purposes. Included in this category is information about the processes currently accessing the resource, and the procedures those processes are executing.

5. the local state of the resource :

Local state includes information that would be present regardless of whether the resource were being accessed concurrently or sequentially. It is information meaningful to the actual unsynchronized resource abstraction. Though local state information is used in many synchronization schemes, its use often causes problems because it interferes with modularity requirements. (The local state information belongs in the resource module, and thus a synchronizer will not have automatic access to it. Several options for handling this problem are discussed in later chapters.)

6. history information:

History information is concerned with whether or not a given event has occurred, such as whether a specific procedure has been executed. This information type differs from synchronization state in that it refers to resource operations that have already completed, as opposed to those still in progress. It is often interchangeable with local state

information, since past events in which we are interested will most likely have left some noticeable change in the state of the resource. It is convenient to treat it as a separate category because it may be easier for the synchronizer to keep track of the history of operations executed than to obtain the required state information from the resource.

We have thus identified two major types of constraints, and several classes of information that distinguish different kinds of constraints within the two major categories. To be sufficiently powerful, a synchronization mechanism must provide a means of expressing exclusion and priority; it must also enable the resource implementor to express those constraints in terms of any of the information types described.

In the next section, examples that use various combinations of constraint types will be given. The way in which a mechanism makes use of different types of information, and how easily it can get access to this information are very important in determining how easily well-structured, reliable solutions can be developed.

2.2.2 Examples

The following are standard examples of synchronization problems. This set was chosen to cover all of the information types presented. Only informal descriptions of the problems are given. Formal specifications seem unnecessary for our purposes, but to avoid any ambiguity, the appendix contains formal specifications using notation from [24].

The bounded buffer problem

The bounded buffer problem assumes there is a fixed size buffer, of length n, into which producer processes are placing data, and from which consumer processes are retrieving it. The constraints specified are that only one process may access the buffer at a time, that the producer may store in the buffer only if it is not full, and that a

consumer may retrieve information from the buffer only if it is not empty. Thus, the constraints make use of information on synchronization state, resource state, and the procedure requested.

Readers_Writers Problems

There are several readers_writers problems[10] that illustrate the use of different types of information. The readers_writers problems assume there is a shared data base having read and write operations. All of the versions used here have the same set of exclusion constraints: reads may occur in parallel, but a write operation excludes both readers and other writers. The priority constraints are different in each version. The similarity of the various forms of the problem makes this set of problems especially useful in evaluating modifiability.

Writers_exclude_others

This version of the problem uses the exclusion constraints mentioned above but imposes no priority constraints. This synchronization scheme illustrates an important type of problem that synchronization mechanisms should be able to handle. The user may not care about the order in which operations are executed in certain cases. There may be external constraints that guarantee that eventually every request will be served, and the order is unimportant. Many mechanisms force the programmer to define an ordering when the specification has none. The inability to leave specifications nondeterministic is a weakness in the expressive power of the mechanism.

Readers_priority (or writers_priority)

In this version a priority constraint is added. If both a read request and a write request are pending, then the read (or in writers_priority, the write) is always given priority. The exclusion constraints remain the same. The priority is now based on the

operation requested. Notice that this scheme allows starvation.[3]

First_come_first_serve (fcfs)

In this version of the readers_writers problem, the type of operation requested is not used at all in the specification of priority constraints. Instead, priority is based entirely on order of request.

Fair_readers_priority

The fair_readers_priority scheme gives readers some priority over writers but limits that priority enough to be sure writers will eventually be served. One way of fulfilling this requirement is by use of the following constraints. If there are writers waiting when a read is requested, then the read must wait until one write completes. All reads waiting at the termination of that write may proceed. These constraints imply that only a finite number of readers have priority over a given writer. The writer that has been waiting longest will have priority over any readers not yet in the resource. The priority constraints for this scheme use a combination of request time and operation type.

The One_slot buffer

The one_slot buffer[8] problem assumes there is a message buffer with room for exactly one message. Users may insert and remove messages. The synchronizer must guarantee that a message is inserted before any process executes a remove, and that no message may be inserted before the previous one has been removed. Thus, an insert may occur only if the previous operation was a remove or a create, and remove may

---

3. Starvation means that a process waiting to access a resource may wait forever and never be granted access. In the readers_priority scheme, since readers have higher priority than writers, if reads are requested often enough a writer may wait forever.

occur only when the previous operation was an insert. Operations occurring out of order must wait until these constraints are satisfied. This example therefore illustrates the use of history information. The synchronizer must keep track of the operations already executed to determine whether a process may enter the resource.[4]

The disk scheduler

The disk scheduler [18] is a scheme to control access to a disk by using an "elevator" algorithm. The disk head moves in one direction until there are no more requests for tracks in that direction; then the direction is reversed. The access request contains the track number as an argument. The algorithm works as follows. If the head is currently moving up (toward higher-numbered tracks) then requests for tracks at the current track or lower must wait for the return pass. Requests that arrive for higher-numbered tracks will be serviced when the head reaches that track on the current sweep. Thus, it is the parameter of the request, and the state of the resource(i.e. current head position and direction) that determine the priority. The exclusion constraint allows only one process at a time to use the disk.

The alarmclock

The alarmclock is a system facility that allows processes to block themselves and request to be restarted after a specified period of time. Thus, granting the "resource request" means restarting the process. The order in which requests are served is based on the argument telling the alarmclock when to grant the request. The alarmclock example itself may not be a realistic use of synchronization. However, it is felt that it illustrates a class of problems that a synchronizer should be able to handle.

---

4. This problem can be restated using local state information if there is some way to determine whether the buffer contains an unread message.

Both the alarmclock and the disk scheduler represent examples of synchronization problems using arguments passed with requests as the basis for determining priorities. The primary difference between them is that the disk scheduler has a fixed number of possible parameter values on which to base the ordering, while the alarmclock may take any integer value as an argument. It therefore may require more mechanism to handle the type of problem illustrated by the alarmclock. Either the disk scheduler or the alarmclock can be used to represent the class of problems using arguments as a basis for priority.

All of the examples given deal with single resources and single accesses in each call to the synchronized resource. We have assumed throughout this thesis that the correct level of synchronization is at the point of access to the resource. One may, in addition, want synchronization at a level encompassing several resource accesses in the course of executing a synchronized operation. Some of the problems resulting from this extension are discussed in the section on correctness of hierarchically structured resources.

We have presented a method for categorizing synchronization problems according to their function and the types of information needed to express their solutions. This categorization will be used in the following section to develop methods for evaluating the expressive power of synchronization mechanisms. Evaluation techniques for ease of use and modifiability also make use of this problem classification.

2.3 Expressive Power

In evaluating the expressive power of a synchronization mechanism, we will be attempting to decide whether the mechanism provides straightforward methods for expressing priority and exclusion constraints, and whether one has the ability to express those constraints in terms of any of the information types described earlier.

One test of expressive power is to use the mechanism to implement solutions to the examples given in the previous section. If there is no direct way to use a certain kind of information, it should become obvious when an attempt is made to implement a solution requiring it. While testing one example from each class of information may be insufficient to guarantee that a mechanism is actually powerful enough, it does provide us with some indication of a mechanism's power, and will at least point out any large gaps in power.

A more general way to measure expressive power is simply to examine each mechanism and attempt to determine what features it has that will enable it to deal with each type of constraint. For example, we will see that monitor queues are a construct for handling request time information, while serializer crowds retain synchronization state information. Some data manipulation technique must be available for each type of information. The ability to identify the particular way in which to handle each information type will also make a mechanism easier to use because the structure of a solution will be indicated by the kinds of information referred to in the specification.

One technique that is often used for comparing the computational power of language constructs, and that has recently been used to compare several versions of monitors [20], is translation between solutions using different mechanisms. In comparing computational power, this technique is useful because if one mechanism can be implemented in terms of another, then the implementing mechanism must be at least as powerful as the one implemented. If the translation is possible in both directions, the two mechanisms must be equally powerful. This technique has been used to show that monitors, serializers and path expressions are all as powerful as semaphores. Since semaphores are considered to be sufficiently powerful as a synchronization construct, all three higher level constructs must have sufficient computational power.

It has been suggested that this translation technique can be used in comparing *expressive* power as well. If there is a *straightforward, simple* translation from one mechanism to another, then the one translated to must have at least the expressive power of the other. We have chosen not to employ this technique because the results of such a translation are unclear. It is too difficult to judge how simple and straightforward a translation algorithm is, or whether the translations in each direction are equivalent in complexity. If the translation in one direction varies slightly in complexity from the one in the other direction, the mechanisms probably vary slightly in power. Though the methods presented earlier for analyzing expressive power seem less algorithmic than the translation technique, we feel that by defining the set of properties we expect a mechanism to express, and then testing for the ability to do so, we have in fact used a more objective approach than translation.

2.4 Ease of Use

In analyzing expressive power, we determine whether a synchronization mechanism allows the straightforward implementation of the synchronization constraints described earlier. Whether or not a mechanism is easy to use depends not only on the ability to easily construct solutions to individual constraints, but on the ability to easily construct implementations of complex synchronization schemes made up of many such constraints.

Given that our requirements for expressive power are satisfied, complex synchronization schemes will be easy to implement only if they can be decomposed into individual constraints that can then be realized independently. If the implementation of any one constraint is dependent upon the other constraints present, solutions quickly become very difficult to construct as the number of constraints increases. Since the implementor must be aware of the entire set of constraints, and make sure that each constraint is consistent with every

other constraint present, the complexity of *constructing* the solution (not the complexity of the solution itself) increases with the number of combinations of constraints present. It is therefore far more difficult to construct a solution than if it were possible to implement each constraint separately, regardless of which other constraints were present.

One way to test whether a mechanism allows independent implementation of constraints is to examine solutions to two similar synchronization problems. If the solutions share some constraints, but differ in others, then the common constraints should be similarly implemented in both solutions. Differences in the way a given constraint is implemented in two different synchronization schemes, or solutions in which the implementations of each individual constraint are not even identifiable as separate parts of the solution, indicate that our independence criterion for constraints is being violated.

Among the examples presented earlier in this chapter, there are several readers_writers problems having a common exclusion constraint. The problems differ in the priority constraints used. These examples provide a good basis for examining independence of constraints. If the implementation of the exclusion constraint cannot be isolated in each mechanism, or if the implementation in each mechanism differs, it is an indication that the mechanism is hard to use. Conversely, if the implementation of this constraint is the same or very similar in each solution, we have a fairly strong indication that each constraint is independent of other constraints in the synchronization scheme.

Assuming a mechanism satisfies this constraint independence property, if it is easy to express solutions to each individual constraint, it will be easy to express solutions to more complex synchronization problems. Our evaluation of expressive power should indicate how easily individual constraints can be expressed. Mechanisms that are easiest to use will be those for which there is a particular structure or method for handling each information class and

constraint type.

2.5 Modifiability

We define modifiability to mean that a small change in a synchronization specification will result in a similarly minor change in its implementation. Like ease of use, modifiability is primarily dependent on the constraint independence property discussed in the previous section. If each constraint is implemented independently, a modification to one constraint should affect only the part of the solution implementing that constraint. If we have shown in our evaluation of expressive power that each type of constraint is easily implementable, then a small change in the specification should be easy to implement.

We can also evaluate modifiability by looking at modifications that might typically be made to some synchronization schemes, and judging whether the extent of the change required in the implementation was consistent with the size of the change in specifications. We would expect that a modification to one constraint that did not affect the type of the constraint or the kinds of information used, would be simple to implement. The structure of the modified solution should be similar to that of the original.

Modifications involving many constraints, or those involving changes in the types of constraints or kinds of information used, are more extensive, and can be expected to require more significant changes to the implementation. However, if it is extremely difficult to change an implementation when a realistic change in specifications has been made, the mechanism may not be consistent with our goals. Such a weakness in modifiability is usually indicative of a weakness in understandability, expressive power, or ease of use as well.

We would like to analyze and compare the ease with which modifications may be made both within a constraint class and between constraint classes. To do so, we will examine several

versions of the readers_writers problem: readers_priority, writers_priority, and fair_readers_priority. The readers_priority and writers_priority examples can be used to evaluate modifiability for the case in which the constraint types are not changed, since both use priority constraints based on procedure requested. The fair_readers_priority problem combines priority based on procedure type with that based on order of request. We would thus expect a change from readers_priority to writers_priority to be easier than a change from readers_priority to fair_readers_priority.

Thus, we can measure the "size" of a modification in terms of the number and types of constraints changed, and use this metric in evaluating how well synchronization mechanisms support modifiability. In this thesis, we will use transformations between various versions of the readers_writers problem to test modifiability.

2.6 Correctness

In the area of correctness, we are concerned primarily with the ability to write correct programs, rather than with techniques for verifying those programs. In the sections on correctness in the following chapters, we will concentrate on two main topics. One is whether there are specific features of each mechanism that will either aid or impede the production of correct programs. Highly structured mechanisms that perform a great deal of syntactic checking will find errors sooner, leaving less to be debugged at runtime. This is especially important when concurrency exists, because parallel programs are prone to time-dependent errors that may not become evident when using traditional debugging techniques. (These mechanisms also ease the verification task by enforcing certain criteria at compile time and removing the burden from the verifier.) We will also attempt to determine whether there are specific syntactic constructs within a mechanism that are particularly hard to use correctly (or are easy to misuse).

The other correctness criterion with which we are concerned is whether the use of a mechanism will often lead to deadlock. When data abstractions are used as a design tool in implementing synchronized resources, the resources may have a hierarchical structure in which the data abstraction representing the resource actually depends on one or more independently implemented, lower level abstractions. If these lower level abstractions are themselves synchronized, we must be careful that the interactions among the various synchronizers do not lead to deadlock. In a hierarchically structured resource, deadlocks can occur in the following situation: suppose an operation of the higher level abstraction calls an operation at a lower level and the synchronizer at the lower level causes the process to wait on some condition. If that condition can only be satisfied through execution of a higher level operation that is excluded until the current operation completes, a deadlock results. This situation, as it applies to monitors, has gained much attention [28] recently. We will find that the problem applies to other mechanisms as well. Part of our examination of correctness issues will be an attempt to decide how often deadlocks due to hierarchical structuring occur in using a given mechanism, and whether such deadlocks can be avoided. Because hierarchical structuring is fundamental to well-modularized programs, it is important that synchronization mechanisms support this structuring in a safe manner.

2.7 Summary

The criteria upon which we plan to base our evaluation of synchronization mechanisms have been presented. These include modularity, expressive power, ease of use, modifiability and correctness. We have provided reasonably precise definitions for these (usually only vaguely defined) terms with respect to synchronization, and developed methods for evaluating how well synchronization mechanisms conform to these criteria. Because relatively

precise meanings have been provided for each criterion, we have been able to provide testing procedures that allow for uniform and fairly objective analyses of each mechanism.

3. Monitors

The monitor construct was developed independently by Hoare[18] and Brinch Hansen[7] as an extension of the secretary concept of Dijkstra[13]. The version used here is the one defined by Hoare.

A monitor consists of a set of operations needed to schedule access to a shared resource, and any local data needed by those operations. Its structure is derived from that of the Simula class construct[11] and is similar to the cluster in CLU[25] and the form in ALPHARD[35]. The construct is presented here using syntax from the programming language CLU,[1] rather than the Simula syntax used in [18], but we have not modified the semantics of the mechanism in any way. The form of a monitor definition is:

```
monitorname = monitor is opl, ..., opn;
rep = record[..local data..]

opl = proc( )
        ...
        .
        .
opn = proc( )
        ...
end monitorname
```

The procedures defined within a monitor module are mutually exclusive. Only one process at a time may execute an operation on a given monitor object. All monitor operations that may be called by users are listed in the *is_list*. The rep (the internal structure of the monitor) is a record that contains all local data needed by the monitor in making synchronization decisions. It may also contain the resource object, in which case users will view

---

1. All examples in this thesis are written in CLU-like syntax so as to provide a uniform language for comparing solutions.

the monitor as a protected resource.

Synchronization is accomplished via two special operations, *wait* and *signal*, which are called from within monitor operations. The invocation wait(queue) causes the calling process to be suspended and placed at the end of the named queue. Control of the monitor is relinquished by the waiting process, so another process waiting to execute a monitor procedure may continue. When a waiting process is restarted, it continues execution at the statement following the invocation of wait.

The invocation signal(queue) restarts the first process on the named queue. This process immediately regains control of the monitor and continues execution. The signalling process is suspended on an *urgent queue*. Processes on the urgent queue have highest priority for regaining control of the monitor when another process relinquishes it. One other operation on queues is provided for use in monitor procedures; the operation *queue* takes one argument, which is a queue, and returns *true* if there is a process waiting on that queue, and *false* otherwise.

When a process executes a wait, it is normally placed at the end of the specified queue. In some cases, it is desirable to specify the order in which processes are to be placed on the queue. The monitor mechanism therefore provides priority queues. The wait operation on priority queues takes a second argument specifying the priority associated with the waiting process.

Monitors may be used in one of two ways; the shared resource may be made a component of the monitor, or the resource and monitor objects can be created independently. If the resource is part of the monitor object, it will be created when the monitor is created; the resource will therefore be accessible only through monitor operations. Since monitor operations are mutually exclusive, mutual exclusion on the resource is automatic.

To allow concurrent access, the resource must be separated from the monitor object. Since the resource will now be accessed outside of the monitor operations, appropriate monitor operations must be invoked before and after resource accesses to ensure proper synchronization. This structure leaves open the possibility of accessing the resource without first using the monitor. Later in this chapter, we will discuss methods of structuring shared resources so as to prevent unsynchronized access, while allowing concurrency.

An example of the use of monitors to solve the bounded buffer problem is given in Figure 1. In this example, the monitor contains the resource (the buffer), two queues, nonfull and nonempty, and the maximum buffer size. We use the name *condition* instead of *queue* in the examples to conform to the notation in [18]. Since the buffer is inside the monitor mutual exclusion is guaranteed.

The monitor operations work in the following way. In the append operation, a test is made to see if the buffer is full. If it is, the append cannot proceed, so the executing process is placed on the *nonfull* queue, and the monitor is released. When there is space in the buffer, the process continues at the statement following the wait. After the data is appended to the buffer, the *nonempty* queue is signalled. Since a message was just inserted, the buffer can no longer be empty, so a process waiting to do a remove may proceed.

The remove operation keeps processes waiting on the *nonempty* queue until data is available in the buffer. When a remove operation completes, a buffer slot becomes available, so the *nonfull* condition queue is signalled. This will cause a process waiting to perform an append to continue.

Figure 1. Bounded Buffer using Monitors

```
bounded_buffer = monitor is create, append, remove;

am= array[message];
rep = record[ slots:am, max:int, nonempty, nonfull: condition]

create = proc(n:int) returns (cvt);
        return (rep${slots:am$new(),
                     max:n,
                     nonempty,nonfull: condition$create()});
        end create;

append = proc(buffer:cvt, x:message) ;
        if am$size(buffer.slots) = max
          then condition$wait(buffer.nonfull);
          end;
        am$addh(buffer.slots,x);
        condition$signal(buffer.nonempty);
        end append;

remove = proc(buffer:cvt) returns (message);
        if am$size(buffer.slots) = 0
          then condition$wait(buffer.nonempty);
          end;
        x:message := am$reml(slots);
        condition$signal(buffer.nonfull);
        return (x);
        end remove;

end bounded_buffer;
```

3.1 Expressive Power

In the last chapter, a set of examples representative of the classes of common synchronization problems was presented. In this section, the monitor solutions to these examples will be described and these solutions will be used to evaluate the expressive power of the mechanism.

The bounded buffer solution has already been presented. This example makes use of resource state information to describe exclusion constraints. The solution given demonstrates

that the use of such information poses no problems for the monitor construct. This type of information is obtainable either by invocation of resource operations that return state information or by keeping the needed information in the monitor object. In Figure 1, the current buffer size is obtained by invoking the size operation, but the maximum size is stored in the monitor.

The next examples to be discussed are the readers_writers problems. These solutions use monitors that are associated with, but do not contain, the resource. Such a structure allows concurrent access to the resource.

Readers_priority

The readers_priority monitor is shown in Figure 2. (The solution is taken from [18], but translated into CLU.) It contains four operations, one to be used before and one after each resource access. To properly synchronize the resource, users must invoke the appropriate monitor operations preceding and following each access.

The solution is relatively simple. The local variable *busy* is used to keep track of whether there is a writer in the resource. *Readercount* is the number of readers in the resource. The startread operation prevents readers from proceeding if a writer is in the resource, while writers must wait in startwrite if any process is currently in the resource, or, because readers have priority, if there are reads waiting. (Since readers only wait if there is a writer in the resource, there is no need for a separate test to determine whether readers are waiting if we are testing *busy*) Endread will signal the writers queue when the last read exits the resource. Endwrite will check whether there are readers waiting and, if so, signal the readers queue; otherwise it will signal the writers queue.

This solution's structure, and its use of request type and synchronization state information are fairly straightforward. The needed information about synchronization state is

Figure 2. Readers_Priority Monitor

```
readers_priority = monitor is create,
                            startread,
                            endread,
                            startwrite,
                            endwrite;

rep = record[readercount: int,
             busy:boolean,
             readers, writers:condition];

create = proc() returns (cvt);
         return(rep${readercount: 0,
                busy:false,
                readers,writers:condition$create()});
         end create;

startread = proc(m:cvt);
         if m.busy then condition$wait(m.readers) end;
         m.readercount:= m.readercount + 1;
         condition$signal(m.readers);
         end startread;

endread = proc(m:cvt);
         m.readercount:= m.readercount - 1;
         if m.readercount:=0
           then condition$signal(m.writers)
           end;
         end endread;

startwrite = proc(m:cvt);
         if m.readercount > 0 | m.busy
            then condition$wait(m.writers)
            end;
         m.busy:=true;
         end startwrite;

endwrite = proc(m:cvt);
         m.busy:=false;
         if condition$queue(m.readers)
            then condition$signal(m.readers)
            else condition$signal(m.writers)
            end;
         end endwrite;

end readers_priority;
```

kept in the local variables *busy* and *readercount*. Request type information is kept by queuing processes requesting different operations on different queues.

While this solution indicates that monitors can adequately handle these information types, it also illustrates some weaknesses in the construct. The monitor mechanism provides no way to associate the monitor with the resource it is to synchronize. If this monitor is used with no additional structure, correct synchronization depends on users of the resource properly invoking monitor operations before and after each access; there is no protection against unsynchronized access. Modularity is impaired because monitor invocations must appear in user procedures, and correctness is undermined because no guarantee of proper synchronization exists.

A method for using monitors that conforms to the model of protected resources discussed in Chapter 2 is needed. Users must only have access to the protected resource, and the synchronization for the resource should be localized within it. This can be accomplished by constructing a *protected_database* abstraction that encapsulates both the monitor and the resource. Users will then have access only to protected_database objects; invocations of monitor and resource operations will be allowed only within protected_database operations. A protected readers_writers database is shown in Figure 3.

It is thus possible to construct synchronized resources with the resource and monitor separated, while maintaining protection from unsynchronized access. This method for doing so is discussed further in the section on modularity.

First_come_first_serve

Another version of the readers_writers problem is the first_come_first_serve scheme. Its solution is given in Figure 4. Because priority in this example is based on time of request rather than type of request, the queuing scheme is different from that of the previous example.

Figure 3. Readers_Priority Protected_Resource Module

```
protected_data_base = cluster is create,read,write;

rep = record[m: readers_priority,d: data_base]

create = proc()returns(cvt);
        return (rep${m: readers_priority$create(),
                d: data_base$create()});
        end create;

read = proc(pdb: cvt) returns(data);
        readers_priority$startread(pdb.m);
        x:data :=data_base$read(pdb.d);
        readers_priority$endread(pdb.m);
        return (x);
        end read;

write = proc(pdb: cvt, x:data);
        readers_priority$startwrite(pdb.m);
        data_base$write((pdb.d, x);
        readers_priority$endwrite(pdb.m);
        end write;

end protected_data_base;
```

Readers and writers are placed on a single queue, thereby ordering them by time of request. However, the exclusion constraints for readers are different from those for writers, so information about type of request is also needed. Because the monitor construct provides no means of identifying the process at the head of a queue or determining the conditions for which it is waiting, the first process on the queue must be dequeued before the exclusion constraints can be checked. In the first_come_first_serve case, it happens that the exclusion constraints for readers are always met when a process is dequeued from the users queue. However, there can be readers in the resource when a signal on the users queue occurs, so the constraints for writers may not be satisfied. If a writer is dequeued when the resource is not empty, the writer will have to wait on a second queue until the constraints are satisfied. The signalling scheme

Figure 4. First_Come_First_Serve Monitor

```
first_come_first_serve = monitor is create, startread, endread, startwrite, endwrite;

rep = record[ busy: boolean,
              readercount: integer,
              users, writer: condition]

create = proc() returns (cvt);
         return(rep${busy:false, readercount:0, users, writer: condition$create()};
         end create;

startread = proc(m: cvt)
         if m.busy | condition$queue(m.writer) | condition$queue(m.users)
           then condition$wait(m.users);
           end;
         m.readercount:=m.readercount + 1;
         condition$signal(m.users); %start all readers
         end startread;

endread = proc(m:cvt);
         m.readercount := m.readercount - 1;
         if m.readercount = 0
           then if condition$queue(m.writer)
                   then condition$signal(m.writer)
                   else condition$signal(m.users)
                   end;
           end;
         %anyone on the writers queue has been waiting longer than those on users queue
         end endread;

startwrite = proc(m:cvt);
         if m.readercount > 0 | m.busy
           then condition$wait(m.users);
           end;
         if m.readercount > 0
           then condition$wait(m.writer);
           end;
         m.busy := true;
         end startwrite;

endwrite = proc(m:cvt);
         m.busy:=false;
         condition$signal(m.users);
         end endwrite;

end first_come_first_serve;
```

ensures that a writer waiting on the writers queue will be served before any other process is dequeued from the users queue. Since no processes are being allowed into the resource, it will eventually empty and the writer will be signalled. This signalling order preserves the first_come_first_serve specification.

It is thus possible to express request time information using monitors. This solution is more complex than the readers_priority solution, but since it contains an additional type of information we would expect some additional complexity. The two queues in the solution maintain different types of information. The users queue keeps track of relative times of request, while the writers queue maintains request type information. Thus, we can identify the part of the solution associated with each constraint. Though it is more complicated than the readers_priority solution, this example still appears reasonably straightforward and easy to understand.

Writers_exclude_others

Though conceptually simpler than the other problems, the writers_exclude_others example creates special difficulties for monitors. The specification of this problem contains only exclusion constraints; the order in which waiting processes are granted access to the resource is unspecified. The difficulty in implementing this specification arises from the way in which priority constraints are handled. The monitor construct requires that control of the monitor be explicitly passed to waiting processes via the signal mechanism. In cases where more than one queue contains processes ready to continue, the signalling procedure must select one of the queues; the priorities of those queues must therefore be explicit in the code of the monitor procedures. There is no way for the programmer to leave the priorities unspecified. In such cases, the design process is made more difficult, and the likelihood of error increased. We would prefer the mechanism to grant access requests in some fair order at times when the order

is not determined.

As a basis for comparison with other mechanisms, we present a solution that satisfies the writers_exclude_others constraint, and, in the cases in which order is not determined by the specification, grants access in order of request. (The ambiguity arises in exactly one case here: when a write terminates, and both readers and writers are waiting.) This solution is basically the first_come_first_serve solution, with the change that if there are already readers in the resource, any new readers will be allowed to continue, even if there are writers waiting. (This solution therefore allows writers to starve.) The solution appears in Figure 5.

One_Slot Buffer

The one_slot buffer problem is a simple example of the use of history information. The resource is a message buffer that can contain only a single message. The insert and remove operations on the buffer must alternate to ensure that no message is lost. Monitors have no specific method for sequencing operations, so the history information is kept as local data. The easiest way to solve this problem in monitors is to treat it as local state information, rather than history information, making it a special case of the bounded_buffer problem. The only local data needed is a boolean indicating whether there is an unread message in the buffer. We could alternatively keep a local variable indicating the last operation performed. Either solution is simple; however, because the implementor must manage the information explicitly, it will be difficult to implement solutions using complex history information. This is not a serious drawback because such schemes do not appear to be common. However, as will be seen in the next chapter, path expressions provide a direct method of expressing such constraints, and are thus better suited for these kinds of problems.

Figure 5. Writers_Exclude_Others Monitor

```
writers_exclude_others = monitor is create, startread, endread, startwrite,endwrite;

rep = record[busy: boolean,
             readercount: int,
             users, writers: condition];

startread = proc(m:cvt);
        if m.busy
          then condition$wait(m.users);
          end;
        m.readercount := m.readercount+1;
        condition$signal(m.users);          %start all waiting readers
        end startread;

endread = proc(m:cvt);
        m.readercount := m.readercount-1;
        if m.readercount = 0
          then if condition$queue(m.writers)
                 then condition$signal(m.writers)
                 else condition$signal(m.users)  %there might be a writer on the users queue
                 end;
          end;
        end endread;

startwrite = proc(m:cvt);
        if m.readercount > 0 | m.busy then condition$wait(m.users) end;
%if there are readers waiting behind a writer at this point,
%they will not be restarted.  To do so requires signalling users again before
%the wait in the next statement.
        if m.readercount > 0 then condition$wait(m.writers) end;
        m.busy := true;
        end startwrite;

endwrite = proc(m:cvt);
        m.busy := false;
        if condition$queue(m.writers)
          then condition$signal(m.writers)  %processes on writers queue have waited longest
          else condition$signal(m.users)
          end;
        end endwrite;

end writers_exclude_others;
```

Alarmclock

The one category of synchronization schemes not yet discussed involves constraints based on arguments passed to the synchronization operations. The alarmclock problem illustrates the use of priority queues to handle such constraints. The alarmclock is a system facility that allows processes to put themselves to sleep until a specified time. The monitor solution to the alarmclock problem is given in Figure 6. One shortcoming of this solution is that the first process in the queue is awakened every time unit; if it is not yet the time at which it was to be restarted, it is requeued. Thus the implementation is awkward. The awkwardness

---

Figure 6. Alarmclock Monitor

```
alarmclock = monitor is create, wakeme, tick;

pq=priority_queue;

rep= record[wakeup: pq, now: int];

create = proc() returns(cvt);
        return (rep${wakeup: pq$create(), now: 0});
        end create;

wakeme = proc(ac: cvt, time: int)
        alarmsetting: int := time+ac.now;
        while ac.now < alarmsetting do
           pq$wait(ac.wakeup, alarmsetting)
           end;
        %the while statement is necessary because the first process on the
        %queue is awakened every tick.
        pq$signal(ac.wakeup);
        %in case the next process has same wakeup time.
        end wakeme;

tick = procedure(ac:cvt);
        ac.now := ac.now + 1;
        pq$signal(ac.wakeup);
        end tick;

end alarmclock;
```

exists because monitors cannot examine the first entry on the queue without dequeuing it first. As noted by Howard[19], adding an operation on priority queues to return the priority of the first element will eliminate this problem.

Disk Scheduler

Although the disk scheduler illustrates the use of information types already presented, we present it here for comparison with other mechanisms. The solution uses two priority queues, upsweep and downsweep, which hold the processes to be served on the next sweep of the disk head up or down the disk. The track number requested serves as the priority for enqueuing processes. In the upsweep queue, the lowest track requested is first on the queue, while the downsweep queue is in the reverse order. The structure of the solution resembles that of the readers_priority solution in that operations are provided for synchronizing before and after the disk access. The primary function of the monitor is to ensure exclusion on the disk, and to move the diskhead in the proper sequence. The solution is shown in Figure 7.

Initially, the disk head is positioned at track 0, and is moving up. When a request to access the disk is made, the track requested is compared with the current track. If the track requested is the current track, the request is queued to be serviced on the next sweep across the disk; immediate service would allow starvation of processes requesting other tracks. If the requested track is greater than the current track, the process is queued on the upsweep queue; if less than the current track, the process waits on the downsweep queue.

When a process releases the disk, the next request on the queue for the current direction is served. If that queue is empty, the direction is changed and the first process on the queue for the new direction is signalled.

Figure 7. Disk_Scheduler Monitor

```
disk_scheduler = monitor is create, request, release;

pq=priority_queue;
rep = record[upsweep, downsweep: pq,
             busy: bool,
             direction: string,
             headpos: cylinder];

create = proc(cylmax: int) returns(cvt);
         return(rep${upsweep, downsweep: pq$create(),
                busy:false,
                direction:"up",
                headpos:0});
         end create;

request = proc(dest:cylinder, sched:cvt);
         if sched.busy
           then  if sched.headpos < dest | (sched.headpos = dest & sched.direction = "down")
                   then pq$wait(sched.upsweep, dest)
                   else pq$wait(sched.downsweep, dest)
                   end
           end;
         sched.busy := true
         sched.headpos:= dest;
         end request;

release = proc(sched: cvt);
         sched.busy :=false;
         if sched.direction = "up"
           then if pq$queue(sched.upsweep)
                  then pq$signal(sched.upsweep)
                  else sched.direction := "down"
                       pq$signal(sched.downsweep)
                  end;
           elseif pq$queue(sched.downsweep)
                  then pq$signal(sched.downsweep)
           else sched.direction := "up"
                pq$signal(sched.upsweep)
           end;
         end release;

end disk_scheduler;
```

3.1.1 Conclusions

We conclude from the analysis of the examples that monitors have the power necessary to express a wide range of synchronization problems. All but the writers_exclude_others problem had straightforward, easily derivable solutions. Furthermore, the analysis made apparent specific ways in which each type of information is handled within monitor solutions, and how each type of constraint is expressed. Request type and request time information are maintained via use of queues, as shown in the readers_priority and first_come_first_serve examples. Information from arguments passed can usually be handled by priority queuing. Synchronization state, history information, and some local state information must be explicitly kept by the user in "local variables" (in CLU, these local variables are additional components of the rep). While use of local variables is a rather low level method of maintaining information, and requires the synchronization procedures to explicitly keep and manipulate the information, it does provide generality. We can therefore be confident that any synchronization constraint can be implemented in a fairly straightforward manner.

The use of explicit signals is probably the weakest point in the monitor mechanism. It affects expressive power in problems such as the writers_exclude_others problem by forcing decisions about priority at every point where a process is restarted. In addition, correctness and understandability are undermined. When a process performs a wait, there is no indication of when or by whom it will be awakened, so it may be difficult to understand the conditions under which it will be resumed. The conditions tested before a wait may not be the same as those that must be true before the process resumes. An example of this situation appears in the fair_readers_priority solution. Readers must wait if there are any writers waiting, but they can be resumed even if some of those writers are still enqueued. It is therefore necessary to examine

all of the monitor procedures to determine when waiting processes will be signalled. Correctness is affected because the implementor must be careful to perform signals at all the necessary points. (It should be noticed in the examples presented that signals on a given queue must often be performed in several places.) Forgetting any point at which the conditions for signalling might become satisfied will lead to incorrect solutions.

It should be mentioned that explicit signals do have several advantages over automatic signalling constructs. Explicit signals are more efficient; they were included in the monitor construct precisely for this reason. Automatic signals, such as those found in serializers, are less efficient because the conditions associated with every queue must be checked each time possession of the synchronizer is relinquished. We can also be sure that explicit signals are powerful enough to implement any ordering scheme we choose. We will see in the serializer chapter that cases exist for which it is easier to write solutions using explicit signals, than using automatic signalling.

3.2 Modularity

In several of the solutions in the previous section the criteria for modularity discussed in Chapter 2 are not met. The bounded buffer solution combines the implementation of the buffer with the synchronization in a single module. The readers_priority example improves the situation by having a separate synchronization module, but provides no way of associating the monitor with the resource to protect against unsynchronized accesses. If monitors are to meet our requirements, we will need to develop a discipline for using them that produces reliable and properly modularized implementations of shared resources.

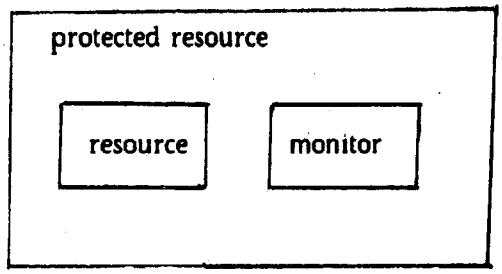
The protected_database module provided with the readers_priority monitor in the previous section (Figure 3) illustrates that an abstraction that encapsulates both the monitor and

resource modules will protect the resource from unsynchronized access, while allowing separate implementations of the resource and monitor. When such an abstraction is used, users of the resource will have access only to the protected object. The operations on the objects of the protected_resource type can ensure that the monitor is properly accessed before and after accesses to the resource. The form of protected_resource objects is shown in Figure 8.

In the general case, the method for producing this structure is as follows. A resource abstraction containing no synchronization should be defined. The synchronization constraints are implemented in a monitor, which will have operations to be called before and after each resource access. The operations to be called before an access must check that constraints are satisfied, and invoke waits if not. Before terminating, this "start" procedure should set monitor information about synchronization state to indicate that the process has entered the resource. It is assumed that the resource will be entered immediately upon leaving the monitor. The operations to be invoked following a resource access should reset the state information and signal any queues for which the associated conditions have become true. We are thus assuming, in designing this monitor, that operations will be called exactly in the order "monitor$start_access; resource$access; monitor$end_access".

We ensure that this order is upheld by creating a protected_resource abstraction, which will contain both the monitor and the resource. Thus, a create operation on the

Figure 8. Protected Resource Structure



protected_resource will create both a resource object and a monitor object, and neither will be accessible to any but protected_resource operations. The operations of the protected_resource correspond to the operations users may invoke on the resource. In other words, for every operation *access* of the resource type, there should be an operation *access* of the protected_resource type.[2] Each protected_resource access operation should contain exactly the three invocations mentioned earlier:

```
access = proc(pr:protected_resource);
            monitor$start_access(pr.mon);
            resource$access(pr.res);
            monitor$end_access(pr.mon);
            end access;
```

Thus, the protected_resource operations enforce the proper use of the monitor when the resource is not inside the monitor.

In addition to providing better modularity and allowing concurrent access, this structure has another advantage over solutions in which the resource is contained in the monitor: it reduces the possibility of deadlocks. Implementing resources inside monitors can lead to deadlocks in the following situation.[3] Suppose the resource were implemented in terms of another abstract type that contained a monitor. Resource operations would be invoked from the monitor containing the resource. A resource operation could then invoke an operation of the lower level monitor. If a wait is executed in the lower level monitor, that monitor will be released, but the higher level monitor will not. If the only place a signal can occur in the lower level monitor is in an operation invoked from the higher level, a deadlock will result.

---

2. There are cases in which the protected_resource operations need not be one-to-one with resource operations. We may want to hide more information than just the synchronization inside the protected_resource. For instance, an operation of the protected resource may perform several resource accesses. The general structure remains the same, however: the protected_resource operations coordinates monitor calls with resource invocations.

3. This problem is referred to as the "nested monitor call" problem in[28].

Separating the resource from the monitor eliminates the possibility of hierarchical deadlock in almost all cases. Because the resource invocations occur outside the monitor, the higher level monitor will be released before the second monitor is entered. Therefore, executing a wait in the lower level monitor will not tie up the other monitor, so no deadlock will arise. The only case in which the potential for hierarchical deadlock still exists is when the monitor must invoke a resource operation. Such a situation may occur when resource state information is needed in the synchronization scheme. This situation is rare, however, so the range of cases in which hierarchical deadlocks can occur has been greatly reduced. In general, therefore, structuring monitor solutions by separating the resource and monitor and providing a protected_resource abstraction substantially improves modularity and correctness.

All of the examples in the previous section, with the exception of the bounded buffer, use the method just described for structuring synchronized resources. The bounded buffer could, of course, be implemented in the same way. However, because mutual exclusion is needed, and buffer operations must be invoked from within the monitor anyway, most of the advantages of this structure do not apply. It therefore seems unnecessary to create three modules to implement this solution. One improvement in modularity that does seem worthwhile for the bounded buffer example is to separate the buffer implementation from that of the monitor, but leave the buffer object inside the monitor. The monitor for this buffer is shown in Figure 9. Since the monitor is not released during calls to the resource, mutual exclusion is still automatic. However, since the resource object is no longer part of the monitor,the modularity is better. The monitor no longer contains information that should be local to the resource, such as the buffer size; it can obtain the needed information by invoking the full and empty operations. The same monitor can now be used for any size buffer. Furthermore, the implementation of the buffer may be changed without modifying the monitor.

Figure 9. Bounded Buffer Monitor

```
protected_buffer = monitor is create, append, remove;

rep = record[ slots:buffer, nonempty, nonfull: condition ]

create = proc() returns (cvt);
        return (rep${slots:buffer$create(),
                nonempty,nonfull: condition$create()});
        end create;

append = proc(pb:cvt, x:message) ;
        if buffer$full(pb.slots) then condition$wait(pb.nonfull) end;
        buffer$append(pb.slots, x);
        condition$signal(pb.nonempty);
        end append;

remove = proc(pb:cvt) returns (message);
        if buffer$empty(pb.slots) then condition$wait(pb.nonempty) end;
        x:message := buffer$remove(pb.slots);
        condition$signal(pb.nonfull);
        return (x);
        end remove;

end bounded_buffer;
```

Conversely, the synchronization scheme for the buffer can be altered without changing the buffer implementation. Modifiability and understandability are therefore enhanced. This structure therefore seems most appropriate for the bounded buffer problem. However, this example is clearly an exceptional case. It is only because of the example's simplicity, and the fact that it uses mutual exclusion and needs resource state information, that this two-module structure seems better than the protected_resource structure described earlier.

In conclusion, we can define a technique for using monitors in a way which conforms to the model defined in the previous chapter. Unfortunately, there is no way to enforce the use of this technique. The lack of enforcement of modularity is one problem that must be recognized if monitors are to be included as a synchronization construct in a language

supporting software reliability.

3.3 Ease of Use and Modifiability

In the section on expressive power, we observed that it was possible to make use of each of our information types within monitor solutions. We could, in fact, identify the way in which each type had to be handled in implementations. We must now determine whether these individual constraints can be easily combined to form more complex solutions. To evaluate constraint independence in monitor solutions, we can compare the implementations of the exclusion constraints in each of the readers_writers problems (see Figures 2, 4, 5). In each, the constraint on reads is implemented in startread by making readers wait if a writer is in the resource, and by ensuring that no write is in progress before signalling the readers queue. Similarly, writes must wait if any process is in the resource. We can thus identify the parts of a solution associated with each constraint, and add new constraints without modifying already existing ones. Some interaction between the exclusion and priority constraints is noticeable in the first_come_first_serve solution, because the priority constraint causes writers to wait on two queues. The exclusion constraint has to be checked before waiting on each one. In most cases, however, it is clear how the exclusion constraints are to be implemented, and priority constraints may be added or changed without changing the existing implementation of mutual exclusion.

The independence of constraints within a solution is the primary determinant of how easily that solution may be modified to implement a slightly different synchronization scheme. We therefore expect monitors to support modifiability fairly well. To test this assumption further, we can compare the solutions to the readers_priority and writers_priority problems: both use the same information types and differ in only one constraint. Thus, the modifications required to change from one to the other should be small. The writer-priority monitor is shown

in Figure 10.

The priority constraint in readers_priority is implemented by signalling readers before writers at the termination of a write, and by allowing readers to enter the resource as long as the exclusion constraint is upheld, regardless of whether there are writers waiting. To change to writers_priority, the signalling in endwrite had to be changed to signal writers before readers, and startread changed to block readers if there are writers waiting. (In the readers_priority solution, startwrite did not have to check whether readers were waiting because reads only waited when a write was in progress, so if *busy* was false, there were no readers waiting.) The modifications necessary to alter the solution were minor and conceptually simple. Only those parts of the solution directly related to the constraint being changed had to be altered.

To determine whether more complex modifications can be made by altering only the parts of the solution related to the constraints being changed, we examine the modification of the readers_priority solution to a fair_readers_priority scheme. This solution combines request type information with information about time of request, thus adding an information type to the specification. Since a change from an unfair to a fair solution is one that seems likely to be made, it is important that modifications of this sort be easy to perform.

The modification requires the addition of request time information to the priority constraints. The needed information can be obtained by checking whether writers are waiting when a read is requested. Thus, to transform the readers_priority solution to a fair solution we need only add a test in startread to make readers wait if a write is already waiting. Readers still get priority when a write terminates. The fair_readers_priority monitor appears in Figure 11.

Though the actual textual changes made are small, it is conceptually more difficult to locate the changes needed in this example. This is to be expected, since an additional type of information is needed. It is still possible, however, to limit the modifications to small sections of

Figure 10. Writers_Priority Monitor

```
writers_priority = monitor is create, startread, endread,
startwrite, endwrite;

rep = record[readercount:integer, busy:boolean, readers,writers:condition]

create = proc() returns (cvt);
        return (rep${readercount:0, busy:false,
                readers:condition$create(),writers:condition$create()});
        end create;

startread = proc(m:cvt);
        if m.busy | condition$queue(m.writers)
          then condition$wait(m.readers)
          end;
        m.readercount:=m.readercount+1;
        condition$signal(m.readers);
        end startread;

endread = proc(m:cvt);
        m.readercount:=m.readercount-1;
        if m.readercount = 0
          then condition$signal(m.writers)
          end;
        end endread;

startwrite = proc(m:cvt);
        if m.readercount>0 | m.busy
          then condition$wait(m.writers);
          end;
        m.busy:=true;
        end startwrite;

endwrite = proc(m:cvt);
        m.busy:=false;
        if condition$empty(m.writers)
          then condition$signal(m.readers)
          else condition$signal(m.writers)
          end;
        end endwrite;

end writer-priority;
```

Figure 11. Fair_Readers_Priority Monitor

```
fair_rp = monitor is startread, endread, startwrite, endwrite,create;

rep = record[ readercount:int, busy:boolean, readers, writers: condition];

create = proc() returns(cvt);
          return(rep${readercount:0,
                     busy:false,
                     readers, writers:condition$create()});
          end create;

startread = proc(m:cvt);
          if m.busy | condition$queue(m.writers)
                     then condition$wait(m.readers)
                     end;
          m.readercount := m.readercount + 1;
          condition$signal(m.readers);
          end startread;

endread = proc(m.cvt);
          m.readercount := m.readercount -1;
          if m.readercount = 0
            then condition$signal(m.writers);
            end;
          end endread;

startwrite = proc(m:cvt);
          if m.readercount > 0 | m.busy
            then condition$wait(m.writers);
            end;
          m.busy := true;
          end startwrite;

endwrite = proc(m:cvt);
          m.busy := false;
          if condition$queue(m.readers)
            then condition$signal(m.readers)
            else condition$signal(m.writers)
            end;
          end endwrite;

end fair_rp;
```

the solution. The structure of the monitor remained unchanged.

From the examples shown in this section, we can see that monitors support modifiability and ease of use. It is easy to determine which parts of a solution are associated with any given constraint, and only these sections must be modified if the specification of that constraint is changed.

3.4 Correctness

There are two correctness issues with which we are concerned. One is the monitor mechanism's use of explicit signals. The other is the possibility of deadlocks due to hierarchical structuring of resources.

The disadvantages of explicit signalling were discussed briefly in the section on expressive power. The weakness of the signal construct lies in the inability of the mechanism to ensure its correct use. Though a queue is intuitively associated with some logical condition, the wait and signal operations provide no way to connect that condition with the actual use of the queue. There is no guarantee that a queue will be signalled when the condition associated with it is satisfied. Conversely, there is also no guarantee that the conditions associated with a signalled queue are true when a signal occurs.

Proof rules for the signal construct do exist, (see [18] and [19]). Thus, while it may be possible to verify that correct programs meet their specifications, the signal construct provides little support for producing the correct programs. Though proof rules are important, they are no replacement for a mechanism that provides more support for producing correct programs initially.

The other issue with which we are concerned is the hierarchical deadlock problem. This problem was discussed in the section on modularity. We have shown that by designing

the protected resource so that the resource is not part of the monitor, we alleviate much of the problem. There appears to be no way to guarantee against such deadlocks.

There has been much discussion about the deadlock problem and possible solutions[28, 22, 29,31], but at present, no solution has completely eliminated the problem. At best, we can minimize the likelihood of its occurrence by properly modularizing monitor solutions.

3.5 Conclusions

We have found that monitors meet our expressive power, ease of use, and modifiability requirements reasonably well. Only the writers_exclude_others problem lacks a simple, easy to construct solution. However, the support given modularity and correctness is weak. The use of the technique shown for properly modularizing monitor solutions overcomes the modularity problems and improves correctness by substantially reducing the possibility of deadlock due to hierarchical structuring of shared resources.

4. Path Expressions

The path expression mechanism was developed by Campbell and Habermann[8] to provide a way to specify the synchronization for a data abstraction as part of the definition of that abstraction. The mechanism is based on the following concept: since access to a resource may be gained only through operations of its type, the synchronization for the resource may be defined as the set of allowable orderings in which those operations may be performed.

A path expression is thus a specification of this set. It is included in the type definition for the shared resource type. A path "controller" keeps track of the operations executed on each object of the type, and ensures that the operations executed on that object conform to some legal ordering. When a process requests execution of an operation named in a path, if there is some allowable ordering in which this operation could occur next, then the process is allowed to proceed. Otherwise, the process is blocked until the path controller determines that the requested operation can execute. It is important to realize that the path expression does not cause the invocation of procedures. Rather, when an operation named in the path is invoked by a process, a check is made to determine whether there is some sequence defined by the path that would allow this operation to execute immediately. It should also be noted that the path is associated with a resource, not a process, and therefore has no control over which process executes which operations. The proper order of operations on a resource must be enforced, but each operation may be performed by a different process.

Several versions of path expressions have been proposed. The version presented here is taken primarily from [8]. This version was chosen because it provides a way to explicitly state that two resource operations may execute simultaneously. If synchronization is to be specified as a set of relationships among operations on the resource, we felt it imperative that

one be allowed to specify concurrency. The assumption that all operations named in paths are mutually exclusive is too strong to allow natural solutions to problems.

The path expression implementation of a synchronization scheme consists of one or more paths of the form:

path ordering specification **end**

where the *ordering specification* describes the set of allowable sequences of operations. The path-end pair, which must surround the ordering specification, denotes that the sequences allowed by the specification may be repeated any number of times. When the end of a path is reached, control returns to the beginning of the path, and waits for an operation request consistent with the start of a sequence allowed by the path. If there are several paths in a module, any operation executed must be consistent with *all* of the paths. If an operation is not named in a path, it is unsynchronized, and may occur at any time, regardless of whether any other operations are executing. Furthermore, unless concurrency is explicitly stated in a path, it is assumed that only one process may be executing an operation named in the path at any given time.

The ordering specification in the path is written in terms of four kinds of relationships between operations of the resource type: sequencing, selection, repetition, and concurrency. The sequencing operator, ";", allows the specification that a set of procedures must be executed in a given order. Thus,

path open; read; close **end**

indicates that *open* must occur before *read*, and *read* must occur before *close*. Since no concurrency is specified, all must be executed sequentially. After *close* executes, the "state" of the path expression is the point prior to *read*. Another *open* must occur before a *read* or *close*. Nothing is implied about which processes execute the operations. Each procedure may be

executed by a different process.

The selection operator, "+", allows only one of the specified procedures to execute at a time. The path

path read + write **end**

indicates that the path controller must select one process from among those waiting to execute *read* or *write* to proceed. The one chosen must also conform to the specifications in other paths. Although [8] states only that selection must be done in some fair order, we will explicitly require that if more than one process is ready to proceed and meets all requirements of the path expression, the one that has been waiting longest will be selected. We will need this first_come_first_serve property to meet our expressive power criteria.

Concurrency is denoted by braces surrounding the section of the path that may be executed concurrently by several processes. Thus, { read } signifies that several processes may execute the procedure *read* at the same time. Once one process starts executing *read* others may start, as long as some execution of *read* is still in progress. Once a point is reached at which no executions of the bracketed procedure are in progress, this portion of the path is considered to be complete. Further requests for *read* must wait until the next repetition of the path (even if the next operation in the path has not yet started).

Concurrency may also be used in conjunction with other path operators. The path

path { read } + write **end**

allows reads in parallel, while a single writer will exclude all other processes. The path

path write; { read} **end**

will allow any number of reads in parallel after a write has occurred. At least one read must occur between writes. As soon as all readers leave the resource, any further reads will be blocked until another write has completed.

The expression { write ; read } means any number of sequences of *write* followed by *read* may execute concurrently. An execution of *write* must complete before the corresponding *read* starts, but any number of *writes* and *reads* may actually be executing at once. The expression { *read* + *write* } means any number of *reads* and *writes* may execute simultaneously.

Repetition permits a pattern of operations to be repeated any number of times. As stated earlier, the path-end pair surrounding a path allows repetition of sequences allowed by the enclosed ordering specification.

Examples of the use of this mechanism will be presented in the next section.

4.1 Expressive Power

In this section, we evaluate expressive power by examining the path expression solutions to the problems described in Chapter 2. Each of these examples was chosen because it illustrates the use of some type of constraint that synchronization mechanisms must be able to express. In discussing the examples, we will also attempt to point out aspects that affect other criteria.

4.1.1 Examples

Writers_exclude_others

The writers_exclude_others problem is one for which path expressions are very well-suited. The solution is extremely simple. One need only include the path:

path { read} + write **end**

in the module defining the resource. If a user invokes a read operation while a write is executing, the path will block the user process; otherwise the read will be allowed to proceed. A

write request can proceed only when the resource is empty.

This example demonstrates that path expressions allow the straightforward implementation of exclusion constraints based on the synchronization state of the resource. The path expression solution is considerably simpler than the monitor solution. This difference can be attributed to the ability to implement nondeterminate specifications with path expressions. The writers_exclude_others problem contains no priority constraints. Thus, if both readers and writers are waiting when a writer leaves the resource, the next process to be served is not described by the specification. When using path expressions, the implementor of the solution need not include any definition of what to do in this case; the path controller will make a fair selection. Monitors, on the other hand, only define service to be first come first serve for processes waiting on a single queue. Since readers and writers are on different queues in the monitor implementation of this problem, explicit information about which queue to serve first must be part of the monitor solution. That solution is therefore more complex.

First_come_first_serve

The path expression solution to the first_come_first_serve problem is shown in Figure 12.[1] READ and WRITE are the operations available to users of the resource. The procedures that actually access the resource are read and write.

When READ or WRITE is called, the corresponding request operation is immediately invoked. The path will allow only one of these requests at a time to proceed, and in the order in which they were invoked. When a requestwrite starts, it invokes write, which must wait until the resource empties. (While it is impossible for other writes to be executing, there may be

---

1. This solution appears in [8], but is characterized there only as a fair solution. Our first_come_first_serve constraint on selection is needed to guarantee the first_come_first_serve property.

Figure 12. First_Come_First_Serve using Path Expressions

```
database = cluster is  READ, WRITE;
rep = ...

path requestread + requestwrite end
path { openread ; read} + write end

requestread = proc(db:database);
        openread(db);
        end requestread;

requestwrite = proc(db:database, k:key, d:data);
        write(db, k, d);
        end requestwrite;

openread = proc(db:database);
        end openread;

READ = proc(db:database, k:key) returns(data);
        requestread(db);
        return( read(db, k));
        end READ;

WRITE = proc(db:database, k:key, d:data);
        requestwrite(db, k ,d);
        end WRITE;

read = proc(db:cvt, k:key) returns (data);
        .
        .
        end read;

write = proc(db :cvt, k:key, d:data);
        .
        .
        end write;

end database;
```

reads in progress.) No other requests can start until the write completes, since write is called from requestwrite, and the requestwrite excludes other requests. When a requestread starts, it invokes openread. When the openread completes, the requestread will terminate, and read will proceed, thus allowing another request to start. Several reads may execute simultaneously, but they can only start if there are no requestwrites waiting.

This example shows that it is possible to use information about time of request in path expression solutions. However, the paths no longer contain only operations that access the resource. Requestread, requestwrite, and openread are "synchronization procedures". Though they are operations of the resource definition module, they are not intuitively operations on the resource, and do not access the resource. They are included solely for purposes of synchronization.

The invocations of requestread and requestwrite serve to record information about time of request in a manner usable by the path expression. If paths contained only operations that were intuitively procedures of the resource type, there would be no way to distinguish between time of request and time of entry into the resource. There would thus be no way to separate request time information from synchronization state information. By separating the user-invoked operations (READ and WRITE) from the actual resource access operations (read and write), and by executing a request operation immediately upon invocation of a user operation, the path expression mechanism can separate request time from entry time.

The openread operation has a different function. It does not provide additional information for use in the paths; rather it forces reads and writes to occur in the same order as their corresponding requests. Without openread in the second path, the following improper sequence of operations could occur: *requestread; requestwrite; write; read* .

Thus, synchronization schemes requiring information about time of request can be

implemented using the path expression mechanism. However, we have evidence that the concept of expressing synchronization via relationships among operations on the resource is not sufficiently powerful. The burden of finding a way to obtain request time information in a form usable by the path expression mechanism has been placed on the resource implementor. While this requirement might be acceptable if there were an easily understandable method of obtaining the information, no such method seems to exist. It is never clear whether the "request" operations should contain the invocation of the resource access operation. (In this example, for instance, requestwrite contains the write invocation, but requestread does not contain the call on read, although both requests are being used to obtain the same kind of information.) Furthermore, using operations such as openread, which coordinate progress through paths, is a conceptually difficult task. Therefore, the need for synchronization procedures should be considered a weakness in the path expression mechanism.

Readers_priority

The readers_priority solution as given in [8] and translated into CLU is presented in Figure 13. This example is more complicated than the previous one; it is easiest to understand if we trace the progress of user requests for access to the resource through the various operations in the module. A READ results in the following sequence of invocations: READ, requestread, read. WRITE causes the invocations: writeattempt, requestwrite, openwrite, write.

Readers gain priority in two ways in this solution. First, since requestreads may execute concurrently, but requestwrites may not, a requestwrite may be blocked indefinitely while requestreads are allowed to proceed because other requestreads are already executing. In addition, readers will get priority in the following way. The first path allows only one writeattempt at a time. Therefore, since requestwrite is invoked from writeattempt, there will be at most one requestwrite waiting at the second path at any time. All other WRITEs in progress

Figure 13. Readers_Priority Database using Path Expressions

```
database = cluster is READ, WRITE;
rep = ...

path writeattempt end
path { requestread} + requestwrite end
path { read } + (openwrite ; write) end

requestwrite = proc(db: database);
        openwrite(db);
        end requestwrite;

writeattempt = proc(db: database);
        requestwrite(db);
        end writeattempt;

requestread = proc(db: database, k: key) returns (data);
        return (read(db,k));
        end requestread;

openwrite = proc(db:database);
        end openwrite;

READ = proc(db:database, k:key) returns(data);
        return (requestread (db,k));
        end READ;

WRITE = proc (db:database, k:key, d:data);
        writeattempt(db);
        write(db,k,d);
        end WRITE;

read = proc(db:cvt,k:key) returns(data);
        .
        .
        end read;

write = proc(db:cvt,k:key,d:data);
        .
        .
        end write;

end database;
```

will be blocked at the first path. However, while a requestwrite or write is in progress, any number of requestreads may enqueue at the second path, awaiting their turn to execute. Thus, during execution of a requestwrite, any number of READs and WRITEs may have started. The READs will have been allowed to proceed as far as the second path; no other WRITEs could have reached that point. Since the selection operator in the second path will restart the process that has been waiting longest at that path, any number of requestreads may have priority over the next requestwrite, regardless of the order of invocation of the corresponding READs and WRITEs.

This solution is difficult to understand; there are complex interactions among the paths, and it is not clear how each resource operation is affected by the paths. It therefore is difficult to convince oneself that the solution handles all cases properly. In fact, there is one case in which this solution does not satisfy the definition of readers_priority as presented in Chapter 2. Consider the case in which there are two WRITEs invoked, followed by a READ, and assume the resource was empty at the time of the first WRITE invocation. The first WRITE will enter the resource. The second WRITE will invoke writeattempt. Suppose the READ occurs after the second write invokes requestwrite but before the first write completes. The requestread will be blocked until the requestwrite terminates. When the first WRITE terminates, there will be a reader and a writer waiting, but the writer will proceed first, violating our definition of readers_priority. The fact that it is so difficult to determine whether the solution satisfies our specifications implies that solutions are difficult to understand and prove correct.

The reason for the complexity of the solution to the readers_priority problem may be the lack of a way to express priority constraints directly. Priorities must be established by designing path expressions that force lower priority operations to wait at additional points in

paths, thus delaying their progress through selection operators when higher priority operations are executing. (Thus, in the readers_priority solution, writers are synchronized at invocations of writeattempt, in addition to requestwrite and write.) The conditions expressed in the priority constraint are not directly reflected in the structure of the solution. This indirect method of expressing priority constraints makes solutions less clear.

Alarmclock problem

This example illustrates the use of arguments to synchronization procedures as a means of determining priority. The solution is taken from [15]; it has been translated into CLU, but conforms as closely as possible to the original. The solution makes use of three data abstractions: wakeuptime, alarmclock, and a list abstraction. Wakeuptime and alarmclock, which contain synchronization, are presented in detail. The specifications for the list abstraction used appear below; the implementation of the list is not provided. The operations and behavior of the abstraction are not those of a standard list; they are closer to those of a stream. A *current* pointer keeps track of the list element currently being processed. It is possible to move this pointer down the list, or reset it to the beginning. New elements may be inserted at the current point, or the current element may be deleted. The list abstraction has the following operations:

advance (list) - sets *current* of list to the next element of the list or nil.

reset(list) - sets *current* back to the first element of the list.

new(list) - inserts a new element preceding *current*. This element becomes *current* of list.

free(list) - deletes *current* of list, and sets *current* to next element.

create() returns(list) - returns a new list.

current(list) - returns the *current* element of the list.

Wakeuptime objects record the time at which processes wish to be awakened. New wakeuptime objects, or those no longer associated with processes, have the value ∞. The operations available on wakeuptime objects are:

create () - creates a new wakeuptime and gives it the value *infinity*.

val(wakeuptime) - returns the time saved in the object.

pass(wakeuptime) - records the fact that the current time has exceeded the wakeuptime.

set(wakeuptime, time) - sets the value of the wakeuptime to the time given.

wakeup(wakeuptime) - executed when the process associated with the wakeuptime is awakened.

Alarmclocks are represented as lists of wakeuptimes. They have two external operations, wakeme and tick. Tick is invoked by a hardware clock at every time unit. Wakeme(n) is called by processes wishing to be awakened in n time units. The implementation of these two abstractions is given in Figure 14.

In this solution, the blocking of processes until the appropriate time is accomplished in the following way. Wakeme calls the internal operation setalarm, which inserts a wakeuptime object into the list representing the alarmclock. The value of the wakeuptime object is the current time plus the number passed as an argument to wakeme. Wakeme then calls wakeuptime$wakeup. However, according to the path in the wakeuptime abstraction, wakeup may only execute after a pass operation has been performed on that object. Therefore, the process that invoked wakeme will be blocked until a pass is called on the wakeuptime object.

Figure 14. Alarmclock

```
wakeuptime = cluster is set,pass,wakeup,val;
        rep = record[wt: int];
        path set ; pass ; wakeup end

        set = proc(n:int,u:cvt)     %sets value of wakeuptime object to n.
            u.wt := n;
            end set;

        pass = proc(u:cvt)                  % when the wakeup time is reached
             u.wt := 0;                     % the value is reset to 0.
             end pass;

        val = proc(u:cvt) returns (int);  % returns the value of wakeuptime u.
            return (u.wt);
            end val;

        wakeup = proc(u:cvt)                % when the process is awakened,
               u.wt:= ∞ ;                   % the corresponding wakeuptime object
               end wakeup;                  % is reset to infinity

        create = proc() returns (cvt);
               return (rep${wt: ∞});
               end create;

        end wakeuptime;


alarmclock = cluster is wakeme,tick,create;
          rep = record[now, first: int; wl: lt];
          lt = list[wt];
          wt = wakeuptime;
          path setalarm + tick end;

            create = proc() returns(cvt);
                   return (rep${now:0, first: ∞, wl:lt$create()});
                   end create;
```

```
%setalarm creates a new element in the list of wakeuptimes
%corresponding to the time at which the calling process
%wishes to be awakened.

        setalarm = proc(x:cvt,n:int) returns(wt);
                time:int := n + x.now;
                lt$reset(x.wl);
                while wt$val(lt$current(x.wl)) < time
                    do lt$advance(x.wl);
                            end;
                if x.first > time then x.first := time; end;
                lt$new(x.wl);
                wt$set(current(x.wl),time);
                return (lt$current(x.wl));
                end setalarm;

%wakeme calls setalarm, then invokes wakeup,
%which will be blocked until the value of the
%wakeuptime object is less than the current time.

        wakeme = proc(x:alarmclock,n:int)
                w:wt := setalarm(x, n);
                wt$wakeup(w);
                end wakeme;

%tick increments the current time and checks whether
%any processes should be awakened.

        tick = proc(x:cvt)
                x.now:=x.now+1;
                lt$reset(x.wl);
                while  lt$current(x.wl) <= x.now do
                        lt$pass(lt$current(x.wl))  %invoking pass will allow wakeup to
                                                   %continue,thus unblocking the waiting process.
                        lt$free(lt$current(x.wl))
                        end;
                end tick;

end alarmclock;
```

Pass will be invoked by the tick operation only when the current time exceeds the value in the wakeuptime object. Thus the process that invoked wakeme (and indirectly, wakeup) will be blocked until the time it asked to be awakened.

The synchronization needed to block processes for the appropriate length of time is therefore found in the wakeuptime abstraction, rather than in the alarmclock abstraction. The path in the alarmclock module is needed only to ensure mutual exclusion on the list of wakeuptimes, so that tick does not access the list while setalarm is updating it.

The user must create and manage the queue explicitly, employing the synchronization mechanism only to awaken the first process at the appropriate time. There is no direct means for handling priority based on arguments passed to the protected resource operations. The monitor mechanism, by contrast, provides a priority queuing option, freeing the user from explicitly maintaining the queue. While the monitor solution is deficient in that it awakens the first process on the queue at every tick, an easy modification to the monitor mechanism allows a solution equivalent in effect to the path expression solution, but far easier to understand.

We therefore conclude that though path expressions have the power to express priority constraints based on explicitly stated priorities, they do not provide enough aid to the user wishing to do so. Synchronization in this example was handled by synchronizing wakeuptime operations appropriately; such an indirect method does not model the structure of the problem specification and thus makes solutions more difficult to understand.

One_slot Buffer

The path expression solution to the one_slot buffer problem is given in Figure 15. The needed information about the history of accesses to the resource can be acquired simply by stating, in the path, the set of allowable histories. The path expression mechanism thus provides a direct way to solve synchronization problems in which we can specify the set of legal histories of operations. This solution is more direct than the monitor solution, which must store history information in local variables.

Bounded Buffer

Figure 15. One_Slot Buffer using Path Expressions

```
buffer = cluster is read, write;
rep = record[m:message];

path write ; read end

read = proc(b:cvt) returns (message);
        return(b.m);
        end read;

write = proc(b:cvt, msg:message);
        b.m := msg;
        end write;

end buffer;
```

Our last example is the bounded buffer. This problem is solved in [15] by placing synchronization at the level of the slots in the buffer, rather than at the level of the buffer itself. While this provides more parallelism than the higher level synchronization, we would like a solution to the problem as defined in Chapter 2, so that it may be compared with the solutions shown for monitors and serializers. In Figure 16, we present a bounded buffer solution that implements mutual exclusion on the entire buffer.

There are three constraints in this scheme: mutual exclusion of appends and removes, exclusion of removes when the buffer is empty, and exclusion of appends when the buffer is full. The implementation of the mutual exclusion constraint, in the first path, is straightforward. (Check_full appears in this path to prevent processes from checking the buffer state while an append or remove is in progress.) The second constraint has been translated to an equivalent constraint that uses history information instead of local state, because path expressions handle history information so easily. The constraint that each remove be preceded by an append is equivalent to prohibiting removes on an empty buffer. The path *path {not_empty; remove} end* implements this constraint, since not_empty is invoked at the end of

Figure 16. Bounded Buffer using Path Expressions

```
bounded_buffer = cluster is REMOVE, APPEND, create;
rep = record[slots : am,
             max : int
             waiting: int];
am = array[message];

path remove + append + check_full end
path { not_empty ; remove } end
path { not_full ; append } end
path APPEND end

create = proc(n: int) returns (cvt);
        return(rep${slots: am$new(),
                max: n,
                waiting: 0});
        end create;

REMOVE = proc (b: bounded_buffer) returns (message);
        return(remove(b));
        end REMOVE;

APPEND = proc(b:bounded_buffer, m:message);
        check_full(b);
        append(b, m);
        end APPEND;

check_full = proc(b:cvt);
        if am$size(b.slots) ~= b.max
                then not_full(b)
                else b.waiting := b.waiting + 1
                end;
        end check_full;

not_empty = proc(b:rep);
        end not_empty;

not_full = proc(b:rep);
        end not_full;
```

```
remove = proc(b:cvt) returns(message);
        m:message := am$reml(b.slots);
        am$set_low(b.slots, 0)      %this sets the index of the first element to 0.
        if am$size(b.slots) = b.max-1 & b.waiting > 0
          then not_full(b);         % only execute not_full if appender is waiting
          end
        return(m);
        end remove;

append = proc(b:cvt, m:message);
        if b.waiting >0
          then b.waiting := b.waiting - 1
          end;
        am$addh(b.slots, m);
        not_empty(b);
        end append;

end bounded_buffer;
```

---

every append operation.

The implementation of the third constraint is somewhat more complex. Because it is dependent on the size of the buffer, this constraint cannot be converted to one using history information. It is implemented by requiring a not_full operation to precede every append. If the buffer has empty slots available, check_full will invoke this operation before calling append. If not, append will be called and will have to wait for a remove operation to invoke the required not_full. Remove checks whether any appends are waiting, and, if so, invokes not_full after freeing a slot.

The synchronization associated with the not_full constraint is not handled directly by the path expression mechanism; instead, the synchronization decisions are made in the procedures. Either check_full or remove decides when another append can execute. The invocation of not_full is used as a *signal* to allow a waiting append to proceed, by providing the first member of the *not_full; append* sequence in path 3. The path is being used only to

block and restart processes according to decisions made in procedures. The management of synchronization in this problem is similar to that in the alarmclock problem. Since paths cannot directly make use of either the arguments to resource operations, or local state information, such information is handled explicitly in implementations of synchronization schemes using it, and the decisions made in the procedures are enforced by the paths.

In addition to the synchronization's being handled primarily in procedures, rather than in paths, the structure of the solution is rather awkward. There are four paths and eight procedures being used to implement a shared resource that intuitively has two operations. There is no clear distinction between synchronization procedures and resource accessing procedures. The remove operation accesses the resource, then calls not_full, which is a synchronization procedure. Check_full accesses the resource and, instead of returning a boolean to indicate whether the buffer is full (as one would expect), invokes not_full also.

These problems are not peculiar to this particular implementation of the bounded buffer. Rather, they reflect problems in using the path expression mechanism. A better-structured solution is not easily derivable. The procedures in the solution do not represent intuitive functional units; they are implemented as such to define the critical sections necessary for correct implementation of the synchronization. The difficulty stems from the fact that, if each constraint is implemented independently, the paths that seem most natural will interact to cause a deadlock when combined. The implementation of the mutual exclusion constraint (independent of other constraints) is:

path append + remove **end**

The implementation of the not_full constraint is:

path not_full ; append **end**

The problem arises if the append operation contains the check on buffer state, and waits if the

buffer is full. Waiting inside append will not release exclusion on the first path; therefore no remove can execute to invoke not_full, and a deadlock results. We are thus forced to create a check_full procedure separate from append. However, the *check_full; append* sequence must be executed uninterrupted to preserve the integrity of the buffer. We therefore need an APPEND procedure that calls both of these operations, and excludes other APPENDS. For similar reasons, the remove and check_full operations must contain both buffer accesses and synchronization invocations to define the needed critical sections. Thus, the modularization for the resource is essentially dictated by the path expression mechanism. More important than the poor modularization, however, is the problem that arises if the implementor does not see the potential conflicts between constraint implementations; deadlock situations are easily created.

The basic conclusion about expressive power, drawn from analyzing the bounded buffer example, is that local state information can be used in path expression solutions, but that it is not directly accessible in paths. Solutions are therefore not very straightforward. As a result, correct implementations can be difficult to construct.

4.1.2 Conclusions

We have now examined solutions to synchronization problems making use of each of the types of information discussed in Chapter 2. Based on this analysis, the following conclusions may be drawn about the expressive power of path expressions.

To be sufficiently powerful, a mechanism must provide a means of directly expressing both priority and exclusion constraints; information about request time, resource state, synchronization state, access history, type of request and parameters passed with each request must be available in implementing a synchronization scheme. Path expressions provide an easy way to express simple exclusion constraints, but no direct means of expressing priority

constraints is provided.

Expressive power is further hampered because certain types of information, such as resource state and arguments passed to the request, cannot be easily used. Paths are intended to express relationships among *procedures* of the resource. Yet, only some of the information classes we have defined are procedure-dependent. To express the other types of information requires use of synchronization procedures. Examples of these procedures appear in the first_come_first_serve, readers_priority, alarmclock, and bounded_buffer problems. These procedures are difficult to use, and tend to increase interaction among paths, making solutions difficult to understand without actually tracing the flow of control.

The most attractive feature of the path expression mechanism is its non-procedural approach to defining synchronization schemes. The need for synchronization procedures clearly undermines this feature. In later sections, the impact of these procedures on modularity and correctness will be discussed.

In conclusion, there are certain classes of problems for which the path expression mechanism seems ideally suited. However, the inability to express other kinds of constraints without the use of synchronization procedures is a severe limitation of the mechanism.

4.2 Modularity

In Chapter 2, several different modularity criteria were discussed. The first of these was the requirement that the synchronization for a shared resource be associated with the implementation, rather than with the use, of that resource. Because path expressions assume the existence of data abstractions, this criterion is incorporated into the path expression mechanism. Path expressions are written in terms of the operations on the resource type, and can occur only within the module implementing the resource abstraction. Thus, users may

assume that the synchronization is handled properly by the protected resource. This structure is in contrast to that of monitors, where we must impose additional constraints on the style in which monitors are used in order to enforce this modularity requirement.

The second modularity requirement is the distinction between the unprotected resource data abstraction and the synchronization abstraction associated with that resource. In simple synchronization schemes, the use of path expressions to implement the synchronization for a data abstraction requires only the addition of paths to the module defining the abstraction. The second requirement is met in these cases: the synchronization is completely implemented by the paths and is therefore clearly identifiable and separable from the implementation of the resource abstraction.

In solutions requiring the use of synchronization procedures, the division is less clear. The synchronization and resource operations are then in the same module. It is more difficult to distinguish between the two. As a result, readability and modifiability are impaired.

A more serious consequence of the use of synchronization procedures in resource modules, is the interaction among operations named in paths. The hierarchy problem in monitors was virtually eliminated by placing monitor operations in a module separate from the resource. This solution will not help in path expressions, because even if the synchronization procedures are placed in a separate module, the resource operations must still be called from synchronization operations. (To show how the synchronization could be put in a separate module, the first_come_first_serve synchronizer is presented in Figure 17.) The hierarchical deadlock problem in path expressions can occur, not only between modules, but within modules

Figure 17. First_Come_First_Serve Synchronization Module

```
protected_database = cluster is READ,WRITE;
rep = database;

path requestread + requestwrite end
path { openread ; read} + write end

requestread = proc(db:database);
        openread(db);
        end requestread;

requestwrite = proc(db: database, k: key, d: data);
        write(db, k, d);
        end requestwrite;

READ = proc(db: database, k: key) returns(data);
        requestread(db);
        return( read(db, k));
        end READ;

WRITE = proc(db:database, k:key, d:data);
        requestwrite(db, k ,d);
        end WRITE;

read = proc(db:cvt, k:key) returns (data);
        return(rep$read(db,k));    % rep$read and rep$write contain the actual
        end read;                  % resource accesses.
                                   % The read and write operations in this module
                                   % are synchronization procedures needed
                                   % to ensure that accesses are performed
                                   % at the correct time.

write = proc(db:cvt, k:key, data);
        rep$write(db,k,d);
        end write;

end protected_database;
```

as well, because synchronization operations within a module often call one another.[2]

The hierarchy problem arises in path expression solutions in the following case. Suppose request1, request2, op1, and op2 are operations named in the path expression shown in Figure 18. Whenever an execution of request2 starts before a corresponding execution of request1, a deadlock results. Request2 will attempt to execute op2 and be blocked awaiting execution of op1. But op1 is only called from request1, and all executions of request1 will be blocked until the current request2 terminates. We thus have a deadlock situation.

This situation is precisely what had to be avoided in our implementation of the bounded buffer problem (see the bounded buffer example in the expressive power section). To emphasize that such interactions among synchronized procedures occur in actual path expression solutions, we will again examine the alarmclock solution taken from [15], and discussed in the previous section. The example appears in Figure 14. The paths in the example show exactly the structure described.

Figure 18. Hierarchical Deadlock in Path Expressions

```
path request1 + request2 end
path op1 ; op2 end

request1 = proc();
        op1();
        end request1;

request2 = proc();
        op2();
        end request2;
```

---

2. This problem is not (theoretically) limited to interaction between synchronization operations, but resource access operations in a module are unlikely to call each other in a way that would cause deadlock.

The two path expressions in the solution are:

path setalarm + tick **end**

path set ; pass ; wakeup **end**.

Since setalarm calls set and tick calls pass, if tick ever executes before setalarm, a deadlock will arise in exactly the way described above. Nothing in the path expression prevents this ordering. An examination of the code for tick will show that tick never calls pass unless the current time is greater than the first wakeuptime in the list. Because wakeuptimes are initialized to infinity, pass will never execute before a setalarm. While the code is correct, this example shows the problems arising from the lack of modularity. To understand how this solution works, and to convince oneself it is correct, requires understanding, and simultaneously dealing with, the implementations of two data abstractions, and the synchronization for both. It was precisely the need to be able to understand each abstraction separately that led to our criteria for separating the synchronization from the data abstraction definition for resources. Thus, path expressions do not uphold our modularity criteria.

Furthermore, because the synchronization operations are used together with resource operations in paths, and because synchronization operations often call other operations named in paths, it is difficult to define conventions for using path expressions that would improve modularity without limiting expressive power.

Thus, monitors and path expressions vary greatly in their support of our modularity criteria. Path expressions guarantee that synchronization for a shared resource is associated with the definition of the resource, rather than with its use. However, they do not provide a means for separating the synchronization from the implementation of the unsynchronized resource. Monitors, in contrast, do not ensure that the synchronization will be separated from the use of the resource. However, it is easy to develop a style of usage that supports both the

association of synchronization with implementation of a shared resource, and the separation of the implementation of the synchronizer from that of the unsynchronized resource. Assuming monitors are used properly, they support modularity far better than path expressions. We will see in the next chapter that serializers offer a still better structure.

4.3 Ease of Use and Modifiability

Ease of use and modifiability are largely dependent upon expressive power. If the tools needed to construct straightforward solutions are not available, it cannot be easy to implement those solutions.

The synchronization problems presented in the section on expressive power provide evidence of the effect of weaknesses in power on ease of use. The need to create synchronization procedures to obtain required information increases the difficulty of constructing solutions because it is difficult to decide what procedures are needed and how they interact with one another. The derivation of the solution to the bounded buffer problem in the expressive power section exemplifies these difficulties.

In this section we will compare the readers_priority and writers_priority problems to evaluate both ease of use and modifiability. The solution to the writers_priority problem is shown in Figure 19. The readers_priority solution was given in the expressive power section, in Figure 13.

While the two solutions are almost symmetric, the amount of code changed in converting from one to the other is large in proportion to the size of the solution: four procedures and all of the paths have to be changed. Even requestread and requestwrite, which are used to obtain the same information in both solutions, must be completely rewritten. Though the exclusion constraint has not changed, the path implementing it has, because it must

Figure 19. Writers_priority Database using Path Expressions

```
database = cluster is READ, WRITE;
rep = ...;

path readattempt end
path requestread + { requestwrite} end
path { openread; read} + write end

readattempt = proc(db: database);
        requestread(db);
        end readattempt;

requestread = proc(db: database);
        openread(db);
        end requestread;

requestwrite = proc(db: database, k: key, d: data);
        write(db, k, d);
        end requestwrite;

READ = proc(db: database, k: key) returns (data);
        readattempt(db);
        return( read(db, k));
        end READ;

WRITE = proc(db: database, k: key, d: data);
        requestwrite(db, k, d);
        end WRITE;

read = proc(db, k) returns (data);
        .
        .
        end read;

write = proc(db, k, d);
        .
        .
        end write;

end database;
```

interact differently with the new priority constraint. When path expression solutions are designed, there is often a problem of finding an implementation of each constraint that will properly interact with the other constraints present. As a consequence, path expressions are often difficult to use.

Since the priority constraint in the two problems presented are exactly reversed, one can reasonably expect their solutions to be symmetric. In the general case, however, when the relationship between the two synchronization schemes is less obvious, the required changes can be much less apparent. The need to change almost all of the code to effect a change in one constraint, even when the change did not require a change in the type of information used, indicates a high degree of interaction among constraint implementations, as well as a lack of support for modifiability.

4.4 Correctness

Many of the correctness issues with which we are concerned have been referred to earlier in conjunction with discussions of modularity and ease of use. Our major concern in the area of correctness is the ease with which a programmer can decide whether an implementation meets its specifications. Whether solutions written using a mechanism can easily lead to deadlock, and whether those deadlocks are easily detectable is part of this problem.

In our evaluation of modularity, we have noted that separation of the synchronization from the resource abstraction is difficult. As illustrated by the alarmclock example, proofs of correctness of the synchronizer cannot be performed independently of the resource implementation. Furthermore, when hierarchically structured resources are involved, proof of termination (absence of deadlock) may involve implementation details from several levels of abstraction. If verification of complex programs is to be possible, it is essential that each

module be independently verifiable, using only external specifications of other modules. Path expressions do not support this property.

Our analysis of expressive power and ease of use also has implications for correctness and verifiability. In particular, consider the readers_priority example. While a correct solution is possible, the fact that it was very difficult to determine whether the solution given met its specifications, and whether all special cases had been covered, leads us to believe that it will in general be very difficult to convince oneself that a solution involving path expressions is correct.

Path expressions do aid verification in one important way. Possible deadlock situations, such as the one arising in the alarmclock solution, are easily detectable at compile time, if they arise in paths in a single module. While an algorithm exists for detecting the same situations occurring between modules, as in the alarmclock case, it requires flow analysis; detection would therefore be rather costly. It should also be noted that the situations detected are *possible* deadlocks. It is far more difficult to determine whether the deadlock is inevitable, or, as in the alarmclock case, will never arise. Thus, at best, the programmer could be warned that the possibility exists, and that proof of termination is impossible.

We conclude that if path expressions supported separation of synchronization from resource implementations, and the independent verification of modules, they would meet our requirements. While easy detection of deadlocks within a module is certainly an important feature, we feel that deadlocks due to conflicts between paths in different modules are too likely to arise. Furthermore, the difficulty of understanding solutions in even a single module leads us to believe that proofs that those solutions meet specifications will be difficult. We therefore feel that the version of path expression presented here does not support correctness of concurrent programs.

4.5 Conclusions

Path expressions are based on the idea of expressing synchronization constraints as sets of relationships among operations of the resource type. This approach appears attractive because it automatically associates the synchronization with the data abstraction defining the resource. It seems natural that synchronization be expressible in terms of operations of the resource type.

Unfortunately, path expressions as defined in [8] do not satisfy all the criteria set forth in Chapter 2. We have found that expressive power is lacking; several types of information needed are not readily accessible. This problem in turn causes awkwardness in solutions, making the mechanism more difficult to use and impeding verification. Synchronization operations are needed in paths, undermining the premise that synchronization is expressible in terms of operations on the resource. The use of these operations also makes it difficult to separate the implementation of a synchronization scheme from that of the resource, which is a modularity requirement we established.

The designers of the mechanism have attempted to overcome some of these problems in later versions of the mechanism[15, 14]. However, none of these has been completely satisfactory. Another version of path expressions now under development[2] promises to show improvements in both expressive power and verifiability. However, unless expressive power can be extended enough to eliminate the need for synchronization procedures, it is doubtful that the new version of the mechanism will meet our criteria either.

5. Serializers

Serializers[3] are similar to monitors but are intended to improve upon those features of monitors that seem poorly structured. There are two significant differences between the two mechanisms. First, serializers incorporate into the mechanism a means for invoking resource operations outside the control of the synchronizer, thus allowing concurrency, while ensuring that all resource accesses are properly synchronized. Second, they replace the monitor signal construct with an automatic signalling mechanism.

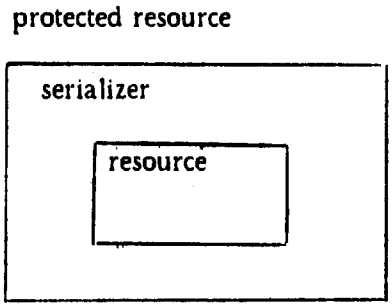
5.1 Mechanism Description

Like monitors, serializers are modules defined by a set of operations and a description of the internal structure of the serializer objects. Serializers may be thought of as encapsulating the resource to form a protected resource object. The structure of this protected resource is shown in Figure 20. Users see only the protected resource; the operations users invoke to access the resource are actually the operations of the serializer.

As in monitors, the operations of the serializer are mutually exclusive. Only one process has access to the serializer at a time. It is not necessary to exit a serializer before

---

Figure 20. Structure of Serializer Objects



accessing the resource in order to obtain concurrency. Serializers provide a means for leaving the serializer temporarily, to perform the resource operations. The invocations of resource operations are textually contained in the serializer operations, but if they are within a 'join_crowd' statement, they will be executed outside the control of the serializer. Other processes may execute serializer operations concurrently with these resource accesses. After the resource access is completed, control automatically returns to the serializer. This structure is similar to that of the modularized monitor scheme proposed earlier (Figure 8). However, leaving and reentering the synchronizer is done automatically in serializers, so an additional 'protected resource' module is unnecessary.

There are two built-in data types used in serializers: queues and crowds. Queues differ from monitor queues in several ways. Rather than wait and signal operations, there is an enqueue operation that specifies, not only the queue on which to wait, but also the condition for which the process is waiting. The serializer mechanism will automatically restart the process when it becomes first on the queue and the condition is satisfied at a time when possession of the serializer is relinquished. No dequeue or signal operation is necessary. The form of the enqueue command is:

enqueue(queue_name) **until** condition

A process executing an enqueue is placed on the end of the specified queue; the condition is not checked until the process reaches the head of the queue.

Crowds are unordered collections of processes used to handle synchronization state information: they keep track of what processes are in the resource and what operations are currently being executed. Though conceptually a crowd contains the identities of the processes involved, it can be implemented simply as a count, since the only information needed is the number of processes using the resource. In addition to the create operation, crowds have a join

operation. Join serves two functions: it puts the process executing the join into the specified crowd, and it releases possession of the serializer. The form of the join command is:

join(crowd) **then** body **end**

where body is a list of statements to be executed by the process when possession of the serializer is relinquished. At the completion of the body, a *leave_crowd* operation is automatically executed. This has the effect of regaining possession of the serializer, and removing the process from the crowd.

Thus, the normal sequence of events for a process requesting access to a shared resource is :

enter (gains possession of the serializer)
enqueue (release possession of the serializer)
dequeue (regains possession)
join_crowd (release possession of serializer and enter resource)
leave_crowd (leave resource, reenter serializer)
exit (releases the serializer)

A set of priorities exists for gaining possession of the serializer. Processes waiting to dequeue have priority over those waiting to enter the protected resource or leave crowds. Processes waiting to enter the protected resource or leave crowds will be handled in first_come_first_serve order.

The solution to the first_come_first_serve problem shown in Figure 21 is an example of a serializer. The resource object is created inside the serializer, so it can be accessed only through invocations of serializer operations. Protection is therefore guaranteed. It is not necessary, as it is in the monitor case, to create a separate protected resource module to associate the resource with the synchronizer and hide it from users.

In the read operation, the process requesting the read must wait on a queue until the writers_crowd empties and all processes preceding it on the waiting_q have continued. Then it

Figure 21. First_Come_First_Serve Serializer

```
first_come_first_serve = serializer is read, write, create;

rep = record[     waiting_q: queue,
                  readers_crowd: crowd,
                  writers_crowd: crowd,
                  db: data_base];

create = proc() returns (cvt);
         return (rep${waiting_q: queue$create(),
                              readers_crowd: crowd$create(),
                              writers_crowd: crowd$create(),
                              db: data_base$create()});
         end;

read = proc(s: cvt, k: key) returns (data);
         queue$enqueue(s.waiting_q) until (crowd$empty(s.writers_crowd));
         d: data
         crowd$join(s.readers_crowd) then
                  d := data_base$read(s.db, k);
                  end;
         return (d);
         end read;

write = proc(s: cvt, k:key, d:data);
         queue$enqueue(s.waiting_q) until (crowd$empty(s.readers_crowd)
                                          & crowd$empty(s.writers_crowd));
         crowd$join(s.writers_crowd) then
                  data_base$write(s.db, k, d);
                  end;
         end write;

end first_come_first_serve;
```

is dequeued (automatically) and proceeds to the statement following the enqueue, where it enters the readers_crowd. Entering the crowd causes possession of the serializer to be released so that other processes may obtain it. Statements in the *then* clause are executed outside the control of the serializer. The read operation is performed and the value is assigned to d; control must then return to the serializer so that the process may be removed from the crowd and leave the protected resource. At termination of the statement in the *then* clause, the process is blocked

until it can obtain possession of the serializer. The priorities defined for obtaining possession of the serializer guarantee that the process will eventually be continued. When execution resumes, the value of d is returned, and the process exits the serializer, allowing another process to gain possession. The write operation differs in the conditions in the *until* clause and the statements in the *then* clause but its basic structure is the same as that of the read.

In the first_come_first_serve example, the only predicates used in *until* clauses are empty tests on queues or crowds. These predicates are sufficient to handle synchronization schemes based on request type and synchronization state. Time ordering of requests is handled by the queuing mechanism. Thus a serializer mechanism using just these predicates is powerful enough for most synchronization problems. This *restricted serializer* is much easier to analyze and construct correctness proofs for than the complete serializer mechanism. To handle other classes of synchronization schemes, however, the mechanism has been generalized. Local variables may be used to store any kind of state information. Priority queues have also been added to handle explicitly passed priorities.

5.2 Expressive power

The first_come_first_serve example was shown in the preceding section. In this section we will present the other examples in which we are interested, evaluate the power of the mechanism, and compare it to monitors and path expressions.

The basic writers_exclude_others readers_writers solution is shown in Figure 22. This solution was difficult to implement using monitors because the specification does not determine a total ordering for requests in all cases. Here, due to the automatic signalling in the serializer construct, the solution can be written without the user specifying the ordering in these cases. However, the way in which the serializer mechanism will handle the situation is unclear. The

Figure 22. Writers_Exclude_Others Serializer

```
writers_exclude_others = serializer is create, read, write;

rep= record[    read_q:queue,
                write_q:queue,
                readers_crowd: crowd,
                writers_crowd: crowd,
                db:data_base]

create = proc() returns (cvt);
        return (rep${read_q: queue$create(),
                     write_q: queue$create(),
                     readers_crowd: crowd$create(),
                     writers_crowd: crowd$create(),
                     db: data_base$create()});
        end create;

read = proc(s: cvt, k: key) returns(data);
        queue$enqueue(s.read_q) until crowd$empty(s.writers_crowd);
        d: data;
        crowd$join(s.readers_crowd) then
                d := data_base$read(s.db, k);
                end;
        return(d);
        end read;

write = proc(s: cvt, k: key, d: data)
        queue$enqueue(s.write_q) until(crowd$empty(s.writers_crowd)
                                        & crowd$empty(s.readers_crowd));
        crowd$join(s.writers_crowd) then
                data_base$write(s.db, k, d);
                end;
        end write;

end writers_exclude_others
```

definition of serializers does not explain how to handle the case in which the conditions governing two queues are true when the serializer is released by some process. Some fair method for dealing with this problem, such as first_come_first_served, should be included in the mechanism definition. The claim made in [3] that solutions should be constructed to avoid having two queues ready at the same time is invalid, since it fails to recognize situations such as

the above, in which the designer really does not need to specify a total ordering of operations. Thus, path expressions seem to be the only mechanism that allows 'incomplete' specifications such as these and guarantees that they will be handled in some fair manner.

The readers_priority solution is shown in Figure 23. (Only the read and write operations are shown; the internal structure of the serializer, and the create operation are that of the previous examples.) Writers are now far more restricted in when they can enter the resource. It can be seen from this example that serializers can easily express priorities based on the type of request. Such priorities are usually expressed by testing empty conditions on queues for operations with higher priority. In this case, for example, readers are given priority by inserting a test in the *until* clause of the write operation to make sure the readers queue is empty before writers proceed. A comparison of this solution to the fair_readers_priority and the writers_priority solutions will be made in the section on modifiability. From the previous two examples it appears that modifications are localized and consistent with changes in the specifications.

Figure 23. Readers_Priority Serializer

```
read =proc(s: cvt, k: key) returns(data);
        queue$enqueue(s.readers_q) until (empty(s.writers_crowd));
        d: data;
        crowd$join(s.readers_crowd) then
                d:= data_base$read(s.db, k);
                end;
        return (d);
        end read;

write = proc(s:cvt, k: key, d: data)
        queue$enqueue(s.writers_q) until (empty(s.readers_q)
                        & empty(s.readers_crowd)
                        & empty(s.writers_crowd));
        crowd$join(writers_crowd) then
                data_base$write(s.db, k, d);
                end;
        end write;
```

Bounded Buffer

The bounded buffer solution is shown in Figure 24. The resource state information is obtained by calls on the resource operations not_full and not_empty. These invocations are made only after checking that no processes are accessing the resource. Since mutual exclusion within a serializer is automatic, we can be sure that no one will enter the resource between the empty test and the invocation of not_full or not_empty. This is important in ensuring the consistency of the resource. The result of a full or empty test performed while another process

---

Figure 24. Bounded Buffer Serializer

```
protected_buffer = serializer is append, remove, create;

rep = record[append_q, remove_q: queue, c: crowd, bb: bounded_buffer];

create = proc() returns (cvt);
        return({append_q: queue$create(),
                remove_q: queue$create(),
                c: crowd$create(),
                bb: bounded_buffer$create()});
        end create;

append = proc(s:cvt,m:message);
        queue$enqueue(s.append_q) until (crowd$empty(s.c)
                                        CAND bounded_buffer$not_full(s.bb));
        crowd$join(s.c) then
                bounded_buffer$append(s.bb,m);
                end;
        end append;

remove = proc(s:cvt) returns(message);
        queue$enqueue(s.remove_q) until (crowd$empty(s.c)
                                        CAND bounded_buffer$not_empty(s.bb));
        m: message;
        crowd$join(s.c) then
                m:= bounded_buffer$remove(s.bb);
                end;
        return (m);
        end remove;

end protected buffer;
```

is updating the buffer is not well defined.

The problem of potential deadlocks resulting from invocations of resource operations from within synchronization modules was explained in detail in the chapter on monitors. The problems arising in serializer solutions are the same. The programmer must be very sure that no deadlocks arise from resource invocations within a synchronizer. Certain synchronization schemes require knowledge of resource state. This state information can be obtained only by invoking resource operations or by keeping the resource state in local variables. The second alternative, while avoiding the deadlock problems, violates the separation of resource from synchronization which is one of our goals. The first alternative, invoking resource operations from within the synchronizer, is not safe unless it can be guaranteed that the resource is empty at the time of invocation.[1]

Thus, serializers handle resource state information in much the same way monitors do, by use of local variables or invocations of state-testing operations on the resource. It must be realized that the operations of the synchronizer are "unsafe areas": the synchronizer can itself access the resource incorrectly. Care must be taken to ensure that these operations impose the necessary restrictions on themselves, as well as user processes.

One_Slot Buffer

Serializers, like monitors, provide no special way of handling history information; it must be handled by local data. The easiest way to solve the one_slot buffer problem is to store the needed information in a boolean describing whether an unread message is in the buffer. The difference between this solution and the bounded_buffer is that we are assuming there is no operation on the resource abstraction, equivalent to full or empty, that the serializer may

---

1. The monitor solution to the bounded buffer problem guarantees mutual exclusion because the buffer is inside the monitor.

invoke to obtain the required information. The information must therefore be deduced by keeping track of past operations. When an insert is executed, the boolean *full* is set to true; it is reset to false when remove takes the message. This solution is shown in Figure 25.

The one_slot buffer is the first serializer example we have seen in which local variables are used in conditions. These variables are set explicitly in the serializer operations. Once general information, rather than just empty tests on queues and crowds, is allowed in conditions, the automatic signalling of serializers loses its advantage over monitor's explicit signals. Programmers are as likely to incorrectly set a local variable, or not set it at all, as they are to forget to explicitly perform a signal.

Figure 25. One_Slot Buffer Serializer

```
protected_single_buffer = serializer is create, insert, remove;

rep = record[insertq, removeq: queue, c:crowd, sb: buffer, full: bool];

create = proc() returns(cvt);
        return(rep${insertq, removeq: queue$create(),
                    c: crowd$create(),
                    sb: buffer$create(),
                    full: false});
        end create;

insert = proc(b: cvt, m: message);
        queue$enqueue(b.insertq) until (~b.full & crowd$empty(b.c))
        b.full := true;
        crowd$join(b.c) then buffer$insert(b.sb, m) end;
        end insert;

remove = proc(b: cvt) returns(message);
        queue$enqueue(b.removeq) until(b.full & crowd$empty(b.c));
        m: message;
        b.full := false;
        crowd$join(b.c) then m:= buffer$remove(b.sb) end;
        return(m);
        end remove;

end protected_single_buffer;
```

Disk Scheduler

The other class of problems to be examined are those requiring user-specified priorities (priorities given by arguments passed to serializer operations). The disk scheduler problem is representative of this group. Priority queues were added to serializers because such problems were difficult to implement without them. The disk scheduler solution using priority queues is given in Figure 26.

When a request to read or write from the disk is made, the request is enqueued in order of track number. The *up* queue holds processes to be serviced as the disk head sweeps up across the disk, the *down* queue as it sweeps down. The variable *current* stores the current track position of the head.[2] If the current position is greater than the requested position, the request will be processed on the next down sweep, so it is enqueued on the down queue. If the current position is lower than the one requested, the request will be placed on the up queue. Requests for the track at which the head is currently located must wait until the current sweep is completed, and the head returns to that track on the next sweep.

A request will be served when there are no other processes preceding it on the queue and the disk head is moving in the proper direction. Whenever a queue empties, the direction changes. When a process gains possession of the serializer after dequeuing, it joins the users crowd, and the appropriate operation on the disk is performed. When it re-enters the serializer, a check is made to see if the queue being serviced is empty; if so, the direction is changed so that the other queue may be serviced.

---

2. Notice that we could have called an operation *current_track* on the disk to obtain this information instead of using local variables. However, this would have led to synchronization problems, since it could only be invoked when the disk was empty. In our solution, the variable *current* in the serializer can be accessed while another process is moving the disk head.

Figure 26. Disk Scheduler Serializer

```
disk_scheduler = serializer is create, read, write;

rep = record[direction:string,
             up:queue,
             down:queue,
             current:int,
             number_of_tracks,
             d:disk,
             users:crowd]

create = proc(n:int) returns (cvt);
         return (rep${direction:"up",
                 up:priority_queue$create(),
                 down:priority_queue$create(),
                 number_of_tracks:n
                 current:0,
                 users:crowd$create(),
                 d:disk$create()});
         end create;

request = proc(s: rep, track_num: int);
         if track_num > s.current | (track_num = s.current & s.direction = "down")
           then  priority_queue$enqueue(s.up,track_num) until
                 (crowd$empty(s.users) &
                 (priority_queue$empty(s.down) | s.direction="up"))
           else   priority_queue$enqueue(s.down,number_of_tracks - track_num) until
                 (crowd$empty(s.users) &
                 (priority_queue$empty(s.up) | s.direction = "down"));
           end;
         s.current := track_num;
         end request;

release = proc(s: rep);
       if s.direction = "up" & priority_queue$empty(s.up)
           then s.direction := "down"
           elseif s.direction = "down" & priority_queue$empty(s.down)
                 then s.direction := "up"; end;
           end;
         end release;
```

```
read = proc(s:cvt, track_num:int) returns(data);
        request(s, track_num);
        d: data;
        crowd$join(s.users) then
          d := disk$read(s, track_num)
          end;
        release(s);
        return(d);
        end read;

write = proc(s:cvt, track_num:int, d: data);
        request(s, track_num);
        crowd$join(s.users) then
          disk$write(s, track_num, d)
          end;
        release(s, track_num);
        end write;

end disk_scheduler;
```

This solution is very similar to the monitor solution. The main difference is that in the serializer solution, the functions of the protected_resource module and the monitor are combined into the single serializer module. (We never saw the read and write operations of the monitor solution, because they are in the protected_resource module, which was not shown.)

This is one example in which the extra module of structured monitor solutions may be beneficial. In cases such as the disk scheduler, where the synchronization does not depend on the operation requested and in fact is the same for all operations, there is actually a distinction between synchronization procedures and protected_resource operations. The function of the synchronizer is to move the disk head to the appropriate track and implement exclusion on disk access. The function of the protected_resource module is to associate the appropriate synchronization operations with each resource operation. In the serializer solution, these two functions are combined in a single module, though the operations are clearly separable into two groups. The user-invoked operations of the serializer look very much like the protected

resource operations in the structured monitor. The read operation, for example, has the form *request, read, release.* Request and release, the two synchronization operations in the monitor solution, are defined as internal operations of the serializer, to be called before and after the resource accesses.

There is thus little difference between the two solutions. Because the synchronization is independent of the operation requested, the monitor structure seems to better model the structure of the problem, and may therefore make it slightly easier to construct the solution. While the distinction between the two structures is relatively minor, and does not represent a serious weakness in the serializer mechanism, it indicates that there are some cases in which the extra modularity of structured monitor solutions is useful.

5.2.1 Conclusions

We can conclude from the examples presented that serializers are sufficiently powerful. The way in which each type of constraint is handled is straightforward. As in the monitor mechanism, request time and request type information are handled by use of queues. Serializers also provide a *crowd* construct to handle synchronization state information, eliminating the need to explicitly keep track of the number of processes in the resource by the use of local variables. History information and some local state information must still be explicitly maintained in local variables.

The only example that illustrates a weakness in the mechanism is the writers_exclude_others problem. The behavior of serializers in cases of incomplete specifications such as the writers_exclude_others problem needs to be more clearly defined.

5.3 Modularity

The most important contribution of serializers is in the area of modularity. The structure for protected resources provided by the serializer mechanism is far more conducive to the development of properly modularized synchronized resources than is the monitor structure. As was stated earlier, we are interested in two distinct properties relating to modularity. One is how easily the synchronization can be separated from the resource implementation and localized in a synchronization module. The other is how well the mechanism supports the use of modularization and hierarchical structure in constructing the resource, and whether the synchronization construct can be used with hierarchically structured resources. Serializers represent an improvement in both of these areas.

In monitor solutions, the only way to allow concurrent access to a resource is to create the resource independently of the monitor. The monitor construct does not provide a mechanism for maintaining an association between the monitor and the resource in this case. The user is responsible for ensuring the correct use of the monitor when accessing the resource. Though a method for ensuring correct access exists, it is the programmer's responsibility, when using monitors, to create a module that encapsulates the resource and the monitor, and invokes the proper synchronization operations when a user of the resource attempts access.

Serializers represent an improvement because they provide this encapsulation automatically. The programmer need only make the resource a component of the serializer construct. The essential difference between monitors and serializers is that serializers allow the resource to be created inside the synchronization module without restricting the access scheme to be mutual exclusion. Because a join_crowd operation releases the serializer while the resource operations are executing, the resource object can be part of the serializer object and still be

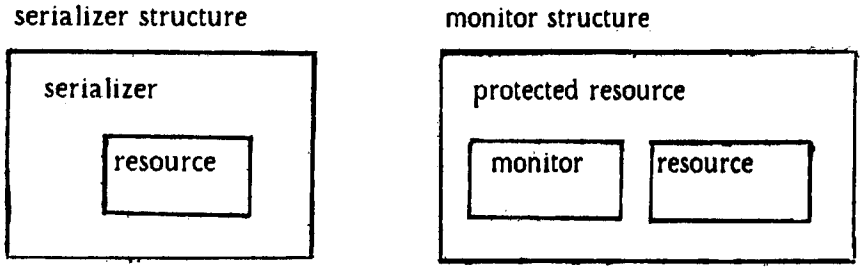
accessed concurrently without violating the constraint that only one process at a time have possession of the serializer. Thus, the programmer need only define the serializer and resource modules, and can assume that the resource is protected (it cannot be accessed without going through the serializer).

The difference may be clearly seen by comparing the structure of a serializer solution with that of a monitor structured as described in the previous chapter. Both are shown in Figure 27.

Though the monitor forces the user to do more work, it also provides some additional modularity. There is a protected resource abstraction separate from the synchronizer. In complicated schemes this additional modularity may be useful, since it allows the designer to deal with the synchronization without worrying about what the actual resource operations are. This is especially helpful when the synchronization scheme is independent of the operation requested, as in the disk scheduling problem. It also makes it easier to change synchronization schemes, or to use the same synchronizer for more than one resource. However, the advantages of the structure provided by serializers outweigh the small improvement in modularity found in the structured monitor solution.

Overall, serializers improve upon the modularity supported by the monitor mechanism.

Figure 27. Comparison of Monitor and Serializer Structures



Because users have an easier way to properly structure solutions, and will find it more difficult to do things incorrectly, software reliability should be enhanced by use of the serializer mechanism.

5.4 Ease of Use and Modifiability

Serializers also satisfy our ease of use and modifiability criteria well. We can easily locate the implementation of each constraint within the solutions presented. In the readers_writers problems (Figures 21, 22, 23, 28), the exclusion constraint on readers is enforced by the condition *crowd$empty(writers_crowd)* in the until clause in *read*, and the constraint on writers is enforced by the condition that both the readers_crowd and writers_crowd must be empty. Other conditions may be added to enforce other constraints, but the implementation of these constraints remains unchanged. The constraint independence criterion we established for evaluating ease of use and modifiability is therefore met.

We can also examine modifications that might be made to synchronization schemes we have discussed to determine how easily those changes can be implemented. In this section we discuss two modifications to the readers_priority scheme.

One modification is to change to a writers_priority scheme. As indicated by our analysis of synchronization problems in Chapter 2, the exclusion constraints remain the same, and there is no change in the types of information used to specify the priority constraints, so the changes needed are expected to be minimal. Conceptually, the difference between the two schemes is that in the writers_priority problem, readers must wait if any writers are waiting, while the reverse is true in the readers_priority problem. In serializer solutions, all of this information is contained in the *until* clause of enqueue statements, so the only parts of the solution that should need modification are these clauses. The dequeue conditions for read must

be changed so that readers must wait until no writers are waiting. The enqueue statement for readers becomes:

```
queue$enqueue(readers_q) until (crowd$empty(writers_crowd)
                                & queue$empty(writers_q));
```

The dequeue condition for writers no longer has to check that no readers are waiting. Thus, the enqueue statement in the write procedure becomes:

```
queue$enqueue(writers_q) until (crowd$empty(readers_crowd))
```

Thus, the changes made were minimal. In addition, it was possible to easily identify those parts of the solution needing modification. Because the conditions for which an enqueued process is waiting are specified at the point of the wait, and restarting is done automatically, constraint implementations are even easier to identify than in monitor solutions. It is no longer necessary to search for signal statements in all of the procedures; the entire implementation of the constraint occurs in the enqueue statement. Changing one constraint in an implementation is therefore straightforward.

A more difficult modification is the change from readers_priority to fair_readers_priority. The fair solution will not allow a reader to enter the resource if a writer is already waiting. Only one writer will proceed at a time, though; so if several writers are waiting when a reader enters, the reader will precede all but the first writer. This solution requires use of request times as well as request type in the priority constraints. The solution is shown in Figure 28.

This solution is fair because the serializer mechanism gives dequeues priority over enters for gaining possession of the serializer. When the resource is empty, the dequeue condition for readers will be satisfied, so all readers on the readers queue will be dequeued and enter the resource before any more read requests can enter the serializer. The readers queue

Figure 28. Fair_Readers_Priority Serializer

```
fair_rp = serializer is create, read, write;

rep = record[readers_q, writers_q:queue,
             readers_crowd, writer_crowd: crowd,
             db:data_base];

create = proc() returns (cvt);
        return(rep${readers_q:queue$create(),
             writers_q:queue$create(),
             readers_crowd:crowd$create(),
             writers_crowd:crowd$create(),
             db:data_base$create()});
        end create;

read = proc(s:cvt,k: key) returns(data);
        queue$enqueue(s.readers_q) until (crowd$empty(s.writers_crowd));
        d: data;
        crowd$join(s.readers_crowd) then
             d:= data_base$read(s.db,k);
             end;
        return (d);
        end read;

write = proc(s:cvt, k:key, d:data);
        queue$enqueue(s.writers_q) until (queue$empty(s.readers_q)
                                    & crowd$empty(s.writers_crowd));
        queue$enqueue(s.readers_q) until (crowd$empty(s.readers_crowd)
                                    & crowd$empty(s.writers_crowd));
        crowd$join(s.writers_crowd) then
             data_base$write(s.db,k,d);
             end;
        end write;

end fair_rp;
```

will then be empty, so the condition for dequeuing writers from the writers queue becomes satisfied, and the first writer on that queue will have highest priority for gaining possession of the serializer. This writer is then enqueued on the (still empty) readers queue. Since the writer is now first on the readers queue, it will enter the resource before any more readers. Assuming read accesses terminate, the readers in the resource will eventually finish and the resource will

empty, allowing the writer at the head of the readers queue to proceed. At the termination of this write, the process just described repeats: all waiting readers will enter the resource, but the first writer on the writers queue will get priority over any new readers entering the serializer. Thus readers still have priority, but writers will not starve, because only a finite number of readers can enter the resource before any write. Note that if several writers are waiting when a reader enters the serializer, only the first of these will enter the resource before the reader.

The change in code from the readers_priority to fair_readers_priority solution is small; only the write operation has changed. One additional enqueue statement has been added to maintain the needed information about relative times of read and write requests. Enqueuing writers on the readers queue is one way to establish a first_come_first_serve order in the necessary cases.

From examining the set of readers_writers problems, we can conclude that minor changes to synchronization specifications result in only minor changes to serializer implementations of those specifications. Identifying the parts of the solution that need modification is straightforward, and our constraint independence criterion is upheld.

5.5 Correctness

In our discussion of correctness in monitors, we were primarily concerned with two issues: explicit signalling and deadlocks due to hierarchical structuring of resources. Serializers have reduced the problems due to explicit signalling by associating conditions with each queue, and automatically restarting waiting processes. For synchronization schemes in which the conditions associated with queues can be expressed in terms of empty tests on queues and crowds, automatic signalling represents a significant improvement in the support given correctness. For synchronization problems that involve resource state information, arguments

passed, or history information, more complex conditions are needed. In these cases, serializers lose their advantage. Local variables are as easily misused as explicit signals. We have also seen a case, in the bounded buffer example, where the integrity of the resource could be easily undermined by incorrectly using resource state information in a condition. If the resource invocations were incorrectly ordered, a condition would have appeared true, and a process would have been dequeued, when the condition was false. Despite these weaknesses, in most cases, the automatic restarting of processes in serializers is superior to explicit signalling.

The problem of hierarchical deadlocks in serializer solutions is equivalent to that in properly structured monitors. Since resource operations are almost always executed outside the control of the serializer, the problem will rarely occur. The only time a hierarchical deadlock can arise is when a resource operation is invoked outside of a join_crowd statement in a serializer operation. As in the monitor solutions discussed, this situation can occur if the serializer is obtaining resource state information via invocations of resource procedures. However, it is unlikely that such an operation would be forced to wait at a lower level. While serializers and "properly used" monitors both avoid the hierarchical deadlock problem in almost all cases, the structure of serializers ensures that the potential for deadlock is minimized, while in monitor solutions, safety is dependent on the programmer properly using the construct. We therefore conclude that, by eliminating the explicit signal construct, and providing more aid in producing better modularized programs, serializers provide better support for developing correct programs than do monitors.

5.6 Conclusions

Serializers have succeeded in improving upon many of the poorly structured features of monitors. Modularity, and thus understandability and ease of use, are enhanced by use of serializers. The use of automatic signalling improves reliability by eliminating one source of programming errors.

The one drawback to the construct is that it is more complex mechanism (since so much more is done automatically) than the monitor mechanism. It is therefore less efficient. Efficiency can be improved by changing from the use of crowds, which actually store process identities, to counts. There appears to be no need for any more information about a crowd than how many processes are currently in it.

The other feature detrimental to efficiency is the automatic signalling. Because monitors allow explicit signalling, processes can often be restarted without any tests on conditions at all, and when tests are needed the programmer can use his or her knowledge about the possible current states to limit the number of conditions that need to be tested. Conceptually, automatic signalling means that the conditions at the head of every queue must be tested each time possession of the serializer is relinquished. Whether such tests actually cost a great deal remains to be determined. Most synchronization schemes do not require very many queues, so the overhead may not be great. While we consider the use of automatic signals to be an improvement over monitors, the reduction in efficiency may make serializers unsuitable for some purposes.

Overall, serializers represent an improvement over monitors. Of the mechanisms evaluated in this thesis, serializers come closest to satisfying our requirements.

6. Summary and Evaluation

This thesis has addressed two issues related to software reliability and synchronization of shared resources. One is how synchronization mechanisms can be evaluated to measure how well they support such criteria as expressive power, ease of use, modularity and correctness. The second is how well existing synchronization constructs meet these criteria.

6.1 Summary and Conclusions

Several results have been derived from our study of evaluation techniques. The development of methods for evaluating expressive power led to a study and definition of the kinds of problems we feel synchronization mechanisms should handle. It has been shown that a synchronization problem may be defined as a set of constraints, which fall into two basic categories, priority constraints and exclusion constraints. In addition, these problems can be categorized according to the kinds of information used to express the constraints. We have identified six categories of information needed in synchronization constraints: the time at which requests are made, the procedure requested, the local state of the resource, the synchronization state of the resource, the arguments passed with the requests, and the history of invocations of resource operations. Furthermore, the categories of information used in a synchronization scheme largely determine how easily that scheme may be implemented using a given mechanism. Thus, by analyzing a mechanism to determine whether it provides access to each type of information and a method for expressing each type of constraint, we can measure its expressive power. In addition, we can estimate the difficulty of implementing a particular class of problems using the mechanism. Methods for evaluating ease of use and modifiability based on this categorization of problems are also described.

The other major result of this study is the application of modularization techniques to the structuring of shared resources. We have shown that significant benefits accrue when a shared resource is implemented as the composition of a synchronization module and an unsynchronized resource module, that is, when all synchronization is handled within the synchronized resource, but is independent of the unsynchronized resource. Not only does this structure improve usability and understandability, but it also reduces deadlock problems in many cases.

The remainder of the thesis is devoted to evaluating monitors, path expressions, and serializers, the three existing mechanisms that seem most likely to satisfy the requirements of good software engineering. Based on this evaluation, we have drawn the following conclusions about these three mechanisms. While the approach taken by path expressions seems very attractive, our analysis has revealed some serious shortcomings. Path expressions do not provide access to several types of information needed in synchronization constraints, and thus lack sufficient expressive power. In particular, it is difficult to use the resource state and the arguments passed to procedures. To maintain information about time of request, or to express priority constraints in general, requires additional synchronization procedures, thus increasing the solution's complexity. In addition, the modularity requirements we find necessary to ensure ease of use and verifiability are not well supported by the mechanism. We therefore conclude that the mechanism does not contribute to the production of reliable, easily maintainable software. The construct might be substantially improved if the need for synchronization procedures could be reduced. Given our enumeration of the kinds of constraints the mechanism must be able to express, it may now be possible to produce a version that incorporates the means for obtaining the necessary information. The use of extra procedures might then be unnecessary, and expressive power, ease of use, and modularity would be greatly enhanced.

Both monitors and serializers satisfy our criteria reasonably well. If asked to select one mechanism for inclusion in a modular programming language now, we would select serializers. Though certain tradeoffs are involved in selecting one of these mechanisms over the other, serializers seem superior in two important respects. First, they meet our modularity requirements more closely. The proper use of monitors requires a special protected-resource module in addition to the synchronizer and resource modules; the resource implementor must also follow specific guidelines for defining monitor operations. Serializers depend less on such rules: the protected-resource module is not needed, and serializer operations are precisely the user-accessible operations on the protected resource. Serializers are thus more likely to be used correctly. The other important distinction between the two mechanisms is the use of automatic signalling in serializers. Though proof rules for the monitor signal construct have been developed, an automatic signalling feature is more likely to aid in constructing correct programs, and in easing the burden placed on the verifier. These differences between monitors and serializers indicate that serializers better support the construction of reliable concurrent programs than do monitors. The tradeoff made in selecting serializers over monitors is one of efficiency for structure.

6.2 Evaluation and Extensions of this Work

There are several areas related to this thesis that we feel warrant further study. The principal contribution of this work has been in outlining a method for evaluating synchronization constructs to determine how well they support the goals of good programming methodology. The method is dependent upon the recognition of classes of synchronization constraints based upon the kinds of information needed to specify a constraint. While this categorization of constraints appears valid, and has proved useful in the evaluations presented

here, a more detailed investigation of synchronization problems may yield more finely grained divisions that could isolate weaknesses in mechanisms still further.

For example, the thesis is limited in the model of shared resources with which it deals. It is assumed that the resource to be synchronized is an object of an abstract data type, and that we are synchronizing individual accesses to that object. A more general analysis would have included several classes of problems omitted here. One such group of problems takes the form of a protected resource whose operations contain several invocations of resource procedures, rather than just one. The bank account problem in [20] is a member of this group. Another set of problems has one synchronizer controlling access to more than one resource. We need to know whether these problems can be reformulated to fit the model used here. If not, it is important to determine what properties synchronization mechanisms must satisfy to handle these problems adequately.

One further extension to the analysis of requirements for synchronization mechanisms is the determination of the properties needed for such a mechanism to be usable in a distributed environment. We believe the modularization of a shared resource, and the association of the synchronization scheme for that resource with the resource definition, is a valid model in both centralized and distributed systems. However, the need for communication between a protected resource and users in a distributed environment may impose further restrictions on the kinds of mechanisms acceptable.

Appendix I - Specification of Synchronization Problems

In this appendix, we present formal specifications for those problems defined informally in Chapter 2. The notation used is that of Laventhal[24]. In this formalism, each invocation of a synchronized operation has associated with it three events: request, enter, and exit. Request is the time at which the synchronizer first becomes aware that a user wishes to execute the operation. Enter is the time at which the process gains access to the resource, and exit is the time at which it leaves. In addition procedure activations are numbered uniquely for each resource object. For example, p_2 denotes the second activation of procedure p. The specifications are written in terms of events, such as p_i^{enter}, which describes the enter event associated with the ith activation of the procedure p. The symbol "→" means temporally precedes.

Writers_Exclude_Others

$$((write_i^{enter} \rightarrow write_j^{enter}) \supset (write_i^{exit} \rightarrow write_j^{enter})) \;\&$$
$$((write_i^{exit} \rightarrow read_k^{enter}) \mid (read_k^{exit} \rightarrow write_i^{enter}))$$

Readers_Priority

$$(read_i^{request} \rightarrow write_j^{enter}) \supset (read_i^{enter} \rightarrow write_j^{enter})$$

Though not explicitly stated, the following two constraints are usually assumed. They state that reads are taken first_come_first serve with respect to each other, as are writes.

$$(read_i^{request} \rightarrow read_j^{request}) \supset (read_i^{enter} \rightarrow read_j^{enter})$$
$$(write_i^{request} \rightarrow write_j^{request}) \supset (write_i^{enter} \rightarrow write_j^{enter})$$

First_come_first_serve

$$(p_i^{request} \rightarrow q_j^{request}) \leftrightarrow (p_i^{enter} \rightarrow q_j^{enter})$$

Here, p and q represent any resource operations. Whichever activation is requested first is the one to enter first.

Fair_Readers_Priority

$$((read_i^{request} \rightarrow write_j^{exit}) \supset (read_i^{enter} \rightarrow write_{j+1}^{enter})) \;\&$$

$$(((write_j^{exit} \rightarrow read_i^{request}) \;\&\; (write_{j+1}^{request} \rightarrow read_i^{request})) \supset$$

$$(write_{j+1}^{enter} \rightarrow read_i^{enter}))$$

One_Slot Buffer

$$(insert_i^{exit} \rightarrow remove_i^{enter}) \;\&\; (remove_i^{exit} \rightarrow insert_{i+1}^{enter})$$

Bounded buffer

$$(insert_i^{exit} \rightarrow remove_i^{enter}) \;\&\; (remove_i^{exit} \rightarrow insert_{i+N}^{enter}) \;\&$$

$$(insert_i^{exit} \rightarrow insert_{i+1}^{enter}) \;\&\; (remove_i^{exit} \rightarrow remove_{i+1}^{enter})$$

Alarmclock

$$((tick_i^{enter} \rightarrow wakeme_j(n)^{request}) \supset (tick_{i+n}^{enter} \rightarrow wakeme_j(n)^{enter})) \;\&$$

$$((wakeme_j(n)^{request} \rightarrow tick_{i+1}^{enter}) \supset (wakeme_j(n)^{enter} \rightarrow tick_{i+n+1}^{enter}))$$

Disk head scheduling

$$((a_z^{enter} \rightarrow a_y^{enter}) \supset (a_z^{exit} \rightarrow a_y^{enter})) \;\&$$

$$((a_i(x2)^{request} \rightarrow a_k(x1)^{exit} \rightarrow a_i(x2)^{enter}) \;\&$$

$$(a_j(x3)^{request} \rightarrow a_k(x1)^{exit} \rightarrow a_j(x3)^{enter}) \;\&$$

$$(a_m(x0)^{exit} \rightarrow a_k(x1)^{exit})) \;\&$$

$$\sim \exists(n)\,((a_m(x0)^{exit} \rightarrow a_n^{exit} \rightarrow a_k(x1)^{exit})) \;\&$$

$$((x0 < x1 < x2 \;\&\; (x2 < x3 \mid x3 < x1)) \mid$$

$$(x0 > x1 > x2 \;\&\; (x2 > x3 \mid x3 > x1)))$$

$$\supset (a_i(x2)^{enter} \rightarrow a_j(x3)^{enter}))$$

References

1. Andler, S., "Synchronization Primitives and the Verification of Concurrent Programs", Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., May 1977.

2. Andler, S., Private communication, May, 1978.

3. Atkinson, R., and C. Hewitt, "Synchronization in Actor Systems", 4th SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang., Jan. 1977, 267-280

4. Berzins, V. and D. Kapur, "Denotational and Axiomatic Definitions of Path Expressions", Computation Structures Group Memo 153, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Oct. 1977.

5. Brinch Hansen, Per, "Concurrent Programming Concepts", Computing Surveys, (5, 4), December 1973.

6. Brinch Hansen, Per, Operating Systems Principles, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1973.

7. Brinch Hansen, Per, "The Programming Language Concurrent Pascal", IEEE Trans. on Software Engineering, vol SE-1, no 2, June 1975.

8. Campbell, R.H., and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science 16, Springer-Verlag, 1974.

9. Campbell, R.H. and P.E. Lauer, "A Spectrum of Solutions to the Cigarette Smokers Problem", TR 63, University of Newcastle upon Tyne, May 1974.

10. Courtois, P.J., F. Heymans, and D.L.Parnas, "Concurrent Control with 'Readers' and 'Writers'", Comm. ACM 14, 10 (Oct 1971), 667-668.

11. Dahl, O.J., "Hierarchical Program Structures", Structured Programming, Academic Press, New York, 1972.

12. Dijkstra, E.W., "Cooperating Sequential Processes", Programming Languages, (F. Genuys, ed.), Academic Press, N.Y. 1968.

13. Dijkstra, E.W.,"Hierarchical Ordering of Sequential Processes", Acta Informatica 1,2 (1971), 115-138.

14. Flon, L. and A.N. Habermann, "Toward the Construction of Verifiable Software Systems", Proceedings of the Conference on Data Abstraction, Definition, and Structure, Sigplan Notices (8, 2) 1976.

15. Habermann, A.N., "Path Expressions", Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1975.

16. Haddon, B.K., "Nested Monitor Calls", Operating Systems Review (11,10), Oct. 1977.

17. Hoare, C.A.R., "Towards a Theory of Parallel Programming", Operating Systems Techniques (C.A.R. Hoare and R.H. Perrott, Eds.), Academic Press, New York, 1973

18. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Comm. ACM (17,10) Oct. 74, 549-557.

19. Howard, J.H.,"Proving Monitors", Comm. ACM (19,5), May 1976, 273-279.

20. Howard, J. H., "Signalling in Monitors", Proceedings of the Second International Conference on Software Engineering, 1976, 47-52.

21. Jammel, A.J. and H.G. Stiegler, "Managers versus Monitors", IFIP Congress Proceedings, 1977.

22. Joseph, P. and V.R. Prasad, "More on Nested Monitor Calls", Operating Systems Review (12,2), April 1978.

23. Kessels, J.L.W., "An Alternative to Event Queues for Syncrhonization in Monitors", Comm. ACM(20,7) July 1977.

24. Laventhal, M.S., "Synthesis of Synchronization Code for Data Abstractions", TR-203, Laboratory for Computer Science, M.I.T., Cambridge, Mass., June 1978.

25. Liskov, B.H., Snyder,A., Atkinson, R., Schaffert, C., "Abstraction Mechanisms in CLU", Comm. ACM (20, 8), August 1977, 564-576.

26. Liskov, B.H., "An Introduction to CLU", Computation Structures Group Memo 136, Laboratory for Computer Science,M.I.T., Cambridge, Mass., Feb. 1976.

27. Lister, A.M. and P.J. Sayer, "Hierarchical Monitors", Proceedings of the 1976 International Conference on Parallel Processing", 1976.

28. Lister, A., "The Problem of Nested Monitor Calls", Operating Systems Review (11,2), July 1977.

29. Parnas, D.L., "The Non-problem of Nested Monitor Calls", Operating Systems Review (12,1), Jan. 1978.

30. Reed, D.P. and r. Kanodia, "Synchronization with Event Counts and Sequencers", Sixth Symposium on Operating Systems Principles, Nov. 1977.

31. Wettstein, H., "The Problem of Nested Monitor Calls Revisited", Operating Systems Review (12,1), Jan. 1978.

32. Wirth, Niklaus, Modula: A Language for Modular Multiprogramming, Institut fur Informatik ETH, Report No. 18, March 1976.

33. Wirth, Niklaus, The Use of Modula, Institut fur Informatik ETH, Report No. 19, June 1976.

34. Wirth, Niklaus, The Design and Implementation of Modula, Institut fur Informatik ETH, Report No. 19, June 1976.

35. Wulf, W.A., R.L.London,and M. Shaw, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Carnegie-Mellon University and USC Information Sciences Institute Tech. Reports, 1976.