LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Evaluating Synchronization Mechanisms

Computation Structures Group 187
December 1979

Toby Bloom

This paper was presented at the Seventh Symposium on Operating Systems and Principles, Asilomar, Pacific Grove, CA, December 1979.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Evaluating Synchronization Mechanisms[1]

Toby Bloom
Massachusetts Institute of Technology
Laboratory for Computer Science

Abstract In recent years, many high-level synchronization constructs have been proposed. Each claims to satisfy criteria such as expressive power, ease of use, and modifiability. Because these terms are so imprecise, we have no good methods for evaluating how well these mechanisms actually meet such requirements. This paper presents a methodology for performing such an evaluation. Synchronization problems are categorized according to some basic properties, and this categorization is used in formulating more precise definitions of the criteria mentioned, and in devising techniques for assessing how well those criteria are met.

1. Motivation

In recent years, much attention has been given to the development of high-level synchronization mechanisms. The need for a mechanism that is higher level than semaphores, and easier to use, is widely recognized. However, the requirements we expect such a mechanism to meet are not fully understood. Properties such as expressive power, ease of use, modularity, and modifiability, are agreed to be important, but these terms are vague; how they apply to synchronization constructs in particular is unclear. Because of this lack of clarity, and because our experience in concurrent programming is so limited, no standard methods have been established for evaluating synchronization constructs.

---

1. 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.

Currently, the power of new mechanisms is demonstrated by showing solutions to a number of standard synchronization problems. The problems selected to demonstrate the use of any construct are usually those most easily solved by the mechanism, and those that illustrate the most significant improvements over other constructs. Unfortunately, we currently have no way of selecting a set of problems to be used in evaluating mechanisms that we know will provide adequate information for choosing between mechanisms. When trying to objectively compare constructs, one has no way of judging which of the standard problems to use as a basis of comparison. Examining all of them is an impossible task; there is no way to determine which will provide new insights or additional information.

It is clear that some well-defined methodology for evaluating synchronization mechanisms is needed. Because the properties in which we are interested are so vague, we can not expect to develop completely objective techniques. Rather, we will make use of the examples so frequently cited. Our goal is to determine what makes each example important, and which properties of a mechanism can be evaluated by looking at particular examples. In this way, we can derive a set of examples that includes all of these properties with a minimum of redundancy; it will then be possible to tell when an evaluation is complete. If we have a specific set of examples to use in testing, and a specific set of characteristics to examine in those examples, our methods of evaluating and comparing synchronization constructs will be greatly improved.

In this paper, the requirements synchronization mechanisms should satisfy have been divided into three areas: modularity, expressive power, and ease of use. In Section 2, we discuss the modularity criteria: how shared resources should be structured, and how mechanisms can support this structure. Section 3 presents a categorization of synchronization problems that will enable us to select a set of examples for use in our evaluations. We also use this categorization when developing methods for measuring the expressive power and usability of synchronization

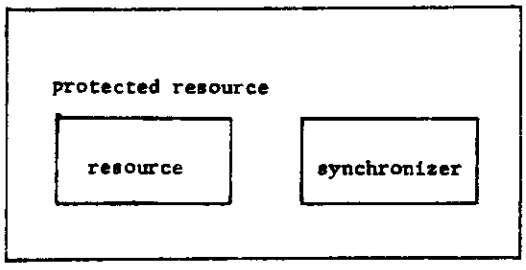
mechanisms (Section 4). The remainder of the paper examines specific synchronization constructs and illustrates how the evaluation techniques described in the previous sections are applied to actual mechanisms.

2. Modularity Requirements

Proper modularization of concurrent programs is essential if the software is to be easily understandable and maintainable. This section discusses the modularity requirements shared resources must satisfy and describes a model for shared resources, based on abstract data types[16], that is assumed throughout the remainder of this paper.

In this model, resources are considered to be objects of abstract types. A resource will therefore have a set of operations associated with it, and the only way to access the resource will be to invoke one of those operations. There are two modularity requirements that should be satisfied by concurrent programs accessing shared resources. The first follows from the principle that the definition of an abstraction should be separated from its use. As applied to shared resources, this principle implies that the shared resource abstraction should contain the implementation of the synchronization scheme, as well as the definitions of the internal structure and operations of the resource. This encapsulation of synchronization and resource will allow users of the resource to assume it to be properly synchronized; no synchronization code need be located at each point of access to the resource.

The other modularity requirement governs the structure of the shared resource definition. The module in which the shared resource is implemented serves two purposes. First, it defines the abstract behavior of the resource (by defining the resource operations). This behavior is independent of whether the resource will be accessed concurrently. Second, it provides the synchronization to control shared access. These two parts serve different functions and should be separable into two subsidiary abstractions, the unsynchronized resource, and the synchronization. The structure of protected resource objects is thus:



One of our requirements for synchronization mechanisms is that they support this structure.

In addition to reducing complexity and aiding in program design, the use of this structure has other important effects. For example, in the case of monitors, use of this structure greatly reduces the chance of deadlock from nested monitor calls [18]. We discuss the use of this structure in monitor solutions later.

3. Categorizing Synchronization Problems

Synchronization mechanisms serve two main functions with respect to shared resources. First, they enforce exclusion of certain processes from the resource when they will interfere with work already in progress. As such, these "exclusion constraints" ensure that consistency is maintained. Second, synchronization mechanisms schedule access to the resource and allow the specification that certain processes have priority over others in gaining entry to the resource. "Priority constraints" are usually concerned with efficiency rather than correctness criteria.

Synchronization schemes are thus composed of a set of constraints, each having the form:

If condition **then exclude** process A

or:

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

where the conditions may be any boolean expressions involving information about the shared resource and the accessing processes.

Within the two main classes (priority and exclusion), constraints differ mainly in the kinds of information referred to in these conditional clauses. This information falls into several categories:

1. the access operation requested :
   The resource is a data abstraction, so it can only be accessed 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 operations requested. In contrast, a strict first_come_first_serve ordering uses no information about the operations requested. We will often refer to this information as the *type* of the request.
2. the times at which requests were made:
   Though it is rarely necessary to know exact times of requests, the time of a request relative to other events is often important. Time information is most frequently used to determine the order of requests.
3. request parameters:
   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. For example, in the disk head scheduler presented in [13], the order in which access is granted is determined by the the track number requested.
4. the "synchronization state" of the resource:
   Synchronization state includes all state information

needed only for synchronization purposes; it would not be part of the resource state were the resource not being accessed concurrently. Included in this category is information about the processes currently accessing the resource, and the procedures those processes are executing. An example of synchronization state information frequently used is a count of the number of processes currently accessing the resource.

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, for example, whether a buffer is full or empty.

6. history information:

   History information is information about whether 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.

4. Evaluation Criteria

We have identified two major types of constraints, and several classes of information that distinguish different kinds of constraints within the two major categories. We can now define two basic requirements that a synchronization mechanism must meet. First, it must provide a straightforward means of expressing each type of constraint and using each type of information. This requirement is our measurement of expressive power. In addition, complex synchronization schemes are composed of many constraints. Such schemes will be easy to implement only if each constraint can be implemented without regard to which other constraints are present in the overall scheme. It must be easy to combine the implementations of all the constraints to construct the solution to the entire synchronization problem. If a mechanism supports this constraint independence or additivity property, and satisfies the expressive power criteria, synchronization schemes will be easy to implement as well as easy to modify. We thus consider constraint independence to be our criterion for ease of use.

4.1 Expressive Power

The first criterion in which we are interested is 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 way to test this ability is to use the mechanism to implement solutions to a set of examples that covers all information classes.[2] 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. By examining how various types of information are handled in each solution, we can draw conclusions about whether the mechanism can easily access each type of information.

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, monitor queues are a construct for handling request time information, while serializer crowds retain synchronization state information. The mechanism must provide some means of manipulating 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.

4.2 Ease of Use

Given that single constraints are easy to implement, complex synchronization schemes will be easy to implement only if they can be decomposed into individual constraints that can then be realized independently. We need to be able to break down complex problems into small parts that can be solved one at a time. If the implementation of any one constraint is dependent upon the other constraints present, solutions quickly become 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

---

2. In the evaluations we performed on several mechanisms, we used the following set of test cases: the bounded buffer problem to represent use of local state information, a first come first serve scheme for request time, a readers_priority database[8] for request type and synchronization state, the disk scheduler problem and alarmclock problem[13] to make use of parameters passed, and the one-slot buffer[7] for history information.

of which other constraints were present.

In addition to the difficulty of initially constructing solutions, if this constraint independence property is not met, the solutions will be very difficult to modify. A change in the specification of one constraint will necessitate reimplementation of the entire solution.

One way to test whether a mechanism allows independent implementation of constraints is to examine solutions to two similar synchronization problems. If the problems 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 problems, 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.

Two readers_writers problems can be used for this analysis. The readers_priority and writers_priority examples have the same exclusion constraints, but differ in priority constraints; however, the constraints in each make use of the same information types. The extent to which the exclusion constraint implementations differ in the two solutions, and the difficulty of modifying the priority constraint in one to obtain a solution to the other, will indicate how independent the constraints are.

To be sure that constraint implementations are independent, we should also check that the implementation of a constraint remains the same when the other constraints are modified to use different types of information. (The readers_priority and writers_priority problems used the same information for priority constraints.) Still another readers_writers problem could be used for this purpose. For instance, a first_come_first_serve scheme has the same exclusion constraint, but uses request time information for the priority constraint. We would expect the implementation of the exclusion constraint to remain unchanged when a modification from readers_priority to first_come_first_serve is made, although the overall change can be expected to be more difficult than a change from readers to writers priority.

It is also possible that usage of two particular types of information will conflict. In this case, constraint independence will be violated only in examples using both types of information. This case is not as serious as general inability to implement constraints independently, but it is not as easy to check. Although indications of such conflicts usually become apparent when analyzing how each individual type is used, the only complete method of evaluation seems to be to check all possible pairs of the six information types. We will see that a situation in which two particular constraint types conflict while all others are independent occurs in the monitor mechanism.

5. Evaluation of Existing Mechanisms

The methodology described has been used to evaluate three existing synchronization mechanisms [5]. While it is impossible to present complete evaluations here, some examples are given to illustrate the use of the method. We will also summarize the conclusions drawn from analyzing the three constructs.

5.1 Path Expressions

In this section we present examples to show how information about the power and usability of a mechanism can be derived from examining solutions to a few synchronization problems, using the methods described. The path expression solutions given here were presented by Campbell and Habermann in [7].

The path expression mechanism permits synchronization to be specified by stating the set of allowable orderings of operations that access the resource. If a request is made for an operation on the resource, and that operation does not occur next in any sequence allowed by the path expression, then the process executing the operation is blocked until a state is reached in which that operation could occur next. The mechanism provides a set of operators for specifying the allowable relationships among operations on the resource. The version of path expressions used here is that presented by Campbell and Habermann in [7]. The following relationships among operations may be specified: concurrency (denoted by "{ }"), selection (" , "), sequencing (" ; "), and repetition (denoted by the path-end pair). We will make the assumption that the selection operator always chooses the process that has been waiting longest. While this assumption is not made in [7], it is necessary for many problems, including some that appear in that paper.

The path expression mechanism is very appealing for several reasons. First, path expressions take a non-procedural approach to specifying synchronization. As such, they seem to take much of the burden of the implementation off the user. Second, they are designed specifically to be used as part of the definition of the abstract type of the resource. The synchronization is thus automatically associated with the resource, satisfying our first modularity requirement.

Unfortunately, path expression solutions to many standard synchronization problems are complex and difficult to understand. The fact that so many versions of the mechanism exist suggests that the designers have found some weaknesses and attempted to correct them. In this section, we will analyze two examples, one to show how expressive power is evaluated, and one to illustrate analysis of constraint dependencies. We then summarize conclusions drawn from a complete evaluation of the mechanism.

5.1.1 Expressive Power

The solution to the readers_priority problem, as given in [7], is shown in Figure 1. This problem states that readers may enter a resource concurrently, but a writer excludes all other users. In addition, if both readers and writers are waiting to access the resource, readers have priority over writers. (This specification allows writers to starve.) Thus, this problem has exclusion constraints based on request type and synchronization state information, and priority constraints based on request type. The implementation of the exclusion constraint is straightforward. In isolation, it would be implemented as:

path { read } , write **end**

(Its implementation in this solution is somewhat more complex due to the need for coordination with other paths. We discuss this in more detail later.)

The realization of the priority constraint in this solution is less straightforward than that of the exclusion constraint. Readers gain priority in two ways. 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. If, in addition, we assume that when a selection is made, the longest waiting process will be chosen, readers will also gain priority in the following way. The first path shown (in Figure 1) 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 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.

Figure 1. Readers Priority Solution

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

where

requestwrite = **begin** openwrite **end**
writeattempt = **begin** requestwrite **end**
requestread = **begin** read **end**
READ = **begin** requestread **end**
WRITE = **begin** writeattempt; write **end**

The interactions among the paths in this example are complex; it is not clear from looking at the solution how each resource operation is affected by the synchronization. It is therefore difficult to convince oneself that the solution handles all cases properly. In fact, it does not produce the same behavior as the readers_priority example presented by Courtois, Heymans, and Parnas[8].[3] It is obvious that the priority scheme is implemented in a rather indirect manner.

Thus, our evaluation of this example shows that path expressions allow straightforward implementation of exclusion constraints based on request type and synchronization state, but do not provide a direct means of specifying priority.

By examining path expression solutions to problems that make use of the other categories of information, we were able to draw the following conclusions about the power of the mechanism. The paths themselves are limited in the kinds of information they can use. Distinctions can be made on the basis of request type. Also, given our extra constraint on selection, request ordering information is accessible (although additional "request operations" may be needed). The automatic mutual exclusion among processes named in paths affords a means of expressing exclusion constraints, although not of directly accessing synchronization state information. There is obviously no way to use parameter values in paths, nor is local resource state information available. Furthermore, no direct means of expressing priority constraints is provided.

When paths cannot express the type of constraint needed, it is still possible to implement the solution. This is done by creating additional procedures in the resource module (which we will call synchronization procedures), and explicitly keeping track of the needed information. Synchronization is accomplished by calling other procedures to signal that the appropriate state has been reached. These procedures are named at crucial points in paths, and serve as gates, to keep the actual access procedures from executing until the appropriate time. The readers_priority example used synchronization procedures (requestread, requestwrite, openread) to maintain priorities. The

---

3. If a *write* is in progress, and another *WRITE* starts, the second writer can start writeattempt and requestwrite, and become blocked at the third path. If a reader enters before the end of the first *write*, it will be blocked at entry to the second path by the requestwrite in progress. The second writer will therefore gain access to the resource before the reader, though readers should have priority.

alarmclock example presented in [11] is another case in which synchronization procedures are used as gates.

Thus, extra synchronization procedures are needed to handle request order, local state, and parameter information. The use of these extra procedures adds a great deal of interaction between procedures and paths, and blurs the distinction between resource implementation and synchronization implementation. When synchronization procedures are needed, the implementor is forced to design the resource and synchronization together, making the task more complex. The implementation becomes more difficult to understand because no clear distinction exists between operations to access the resource and operations to synchronize it. Our modularity requirement that resource and synchronization should be separated stems from the need to avoid this complexity. Thus, the mechanism does not adequately support our second modularity requirement. The use of synchronization procedures also detracts from the non-procedural approach of the mechanism.

It is interesting to note the correspondence between weaknesses illustrated through use of our evaluation methodology, and those that the mechanism designers have attempted to correct in later versions of the mechanism. In the second version of path expressions[11], a priority operator was added, as was a conditional operator that allowed use of resource state information and synchronization state information in paths. The version presented in [10] introduced a numeric operator that improved explicit use of synchronization state information, as well as history information. Finally, Andler[2] has introduced predicates and state variables for use in paths. This version comes closest to satisfying our requirements, although synchronization procedures are still needed in some examples. We thus have evidence that the weaknesses revealed by this method of analysis correspond to some extent with those found in other evaluations. The advantage here is that we could immediately identify several weaknesses and avoid the many iterations that take place to correct the problems found from analysis of examples one at a time.

5.1.2 Ease of Use

The other property that must be examined, according to our methodology, is the additivity property of constraints. The use of synchronization procedures has a great impact in this area as well. Because of the interactions between synchronization procedures and paths, it is difficult to differentiate the implementations of various constraints. As stated earlier, the exclusion constraint for the readers_writers problems, when implemented in isolation, is:

path { read } , write **end**

In the readers_priority solution [Figure 1], the openread procedure, which is part of the priority constraint implementation, is in the path implementing the exclusion constraint. The openread operation is invoked from within the requestread operation named in the second path and serves to coordinate the exclusion constraint with the priority constraint. Thus, if the exclusion constraint were already written and we wanted to add the priority constraint, we would need to determine how the implementation of the new constraint interacted with the old and add the appropriate procedures to coordinate them. The constraints are therefore not independent. This conclusion is further supported by a comparison of the readers_priority solution with a writers_priority solution (Figure 2). The path implementing the exclusion constraint is different in the writers_priority solution. Furthermore, to modify a readers_priority solution to writers_priority involves changing every synchronization procedure and every path, even though the exclusion constraints are unchanged, and the priority constraints make use of the same kinds of information. A modification to one constraint involves changing the entire solution.

5.2 Monitors and Serializers

Similar analyses of monitors [13] and serializers [3] yielded very different results from that of path expressions. We summarize the results of those evaluations in this section; more detailed explanations of the application of our evaluation techniques to these constructs can be found in [5].

Monitors allow access to all of the information types described: the condition queue construct is obviously useful for maintaining request type and request time information; priority queues provide a means for using most needed information from arguments. Synchronization state, as well as any other needed information, must be explicitly kept by the user as local data of the monitor. The ways in which the information must be handled are, in general, direct and easy to understand. We also found that constraints were independent in most cases; the difficulty in making modifications corresponded to the extent of the change desired. One exception to the constraint

---

Figure 2. Writers Priority Solution

```
path readattempt end
path requestread , { requestwrite } end
path { openread ; read } , write end

where

readattempt = begin requestread end
requestread = begin openread end
requestwrite = begin write end
READ = begin readattempt ; read end
WRITE = begin requestwrite end
```

independence property is due to the explicit signal mechanism. Because of the use of explicit signals, a total ordering of processes must be defined by the priority constraints; thus, an exclusion constraint cannot be implemented without priority constraints. The implementor must decide in advance the order in which waiting processes will be signalled.

The other case in which constraint independence is violated is due to a conflict between two particular information types. The monitor mechanism uses queues to maintain both request type and request time information. Request type distinctions are made by placing processes with different types on separate queues, thus allowing them to be handled differently. Request ordering is maintained by placing the processes to be ordered on the same queue. Thus, a problem using request type information as well as request order requires that processes be placed on separate queues as well as the same queue. These two information types therefore conflict. The problem is solved by maintaining two stages of queuing: processes are first enqueued on a single queue, and, when they reach the head of that queue, separated onto distinct queues based on request type. Because the interference between constraints occurs only in this limited case, and since a standard solution is available, the problem is not serious. It is, however, an illustration of interaction among constraints in a synchronization scheme.

Monitors do not directly support the modularization suggested in Section 2. While the synchronization is located with the resource, rather than with users of the resource, the mechanism does not encourage separation of the resource implementation from the synchronization. In many examples shown in [13], the resource and the synchronizaiton data are both considered to be local data of the monitor, and no distinctions are made in the way each is accessed. Monitors do, however, allow the proper modularization, and a standard method for properly structuring shared resources is easily developed. Such a structure consists of three modules: a shared resource module, a resource, and a monitor. Shared resource objects contain two parts: a resource object and a monitor object. The operations of the shared resource invoke monitor operations before and after each resource operation; users have access only to the shared resource. Overall, this method of using monitors satisfies our modularity requirements, but is totally dependent on implementors properly using the mechanism.

This structure significantly reduces the problem of nested monitor calls [18]. The nested monitor call problem results when an operation in one monitor is always invoked from an operation within another monitor. If the second monitor waits, a deadlock will result because the second monitor is released by the wait, but the calling monitor is not. Therefore, no other process can enter the higher-level monitor to gain entry to the lower-level one and signal the waiting process. The higher-level monitor thus waits forever.

Such a situation is likely to arise when resources are hierarchically structured, and monitors are used at several levels, if the resource operations are the monitor operations. When shared resources are structured as described above, the monitor is released before the resource operation is invoked. Thus, even if monitors are used to protect several different levels in a hierarchically structured resource, each monitor is released before the lower level operation is called. Therefore, no deadlock will result. Deadlocks can, of course, still arise if resource operations are invoked from within monitor operations.

The serializer mechanism was proposed by Atkinson and Hewitt[3] to improve upon the modularity of monitors, and to enhance verifiability and ease of use by inclusion of an automatic signalling construct to replace monitors' explicit signalling mechanism.

The way in which the serializer mechanism developed [4] illustrates the need for defining the requirements synchronization constructs should meet. The monitor example in which weaknesses in modularity were perceived was the readers_writers problem. While the first version of serializers successfully improved modularity, it had several deficiencies in expressive power. It had essentially been created around the readers_writers problems, and so included data structures for handling request type, request time, and synchronization state information, but could not easily handle resource state, arguments passed to requests, or history information. Local variables and priority queues had to be added later. This situation emphasizes the need for a clear definition of the types of problems synchronization mechanisms should be able to handle.

With respect to our expressive power criteria, serializers are similar to monitors. Each of the information types is accessible. Serializers provide an additional data structure for maintaining synchronization state information - crowds. Crowds maintain information about processes currently accessing the resource in much the same way monitor conditions and serializer queues maintain information about processes waiting to enter the resource. This eliminates the need for explicitly keeping counts, as required in monitors. Because counts are not very difficult to maintain, the advantages of the additional mechanism are unclear.

The automatic signals of serializers have an unexpected benefit: they separate the means of using request time and request type information. In monitor solutions, as mentioned earlier, these two information types interfere with each other because both require queues. Serializers allow processes waiting for different conditions to be enqueued on the same queue. Request order is still maintained by enqueuing the requests to be ordered on a single queue; different request types on a queue can be

distinguished by differences in the conditions for which enqueued processes are waiting. Of course, this extra mechanism also comes at the expense of efficiency.

The other substantial benefit of serializers is the improved modularity. Serializers provide, as part of the mechanism, the means for structuring resources in the method suggested for monitors. Conceptually, the serializer surrounds the unsynchronized resource. The serializer and resource modules can be implemented independently, but the serializer object contains the resource. In addition, serializers provide a way of leaving control of the serializer while resource accesses are being performed. The "join_crowd" operation not only places the invoking process in a crowd, it releases control of the serializer, making it available to other processes. The "leave_crowd" operation reenters the serializer. This structure thus avoids the "nested monitor call" problem while providing a structure for automatically associating the synchronization with the resource. If the resource is created inside a serializer, users can only access the resource by going through the serializer. In monitor solutions, if the resource is inside the monitor, no concurrency is possible, and deadlocks are likely. If the resource is outside the monitor, deadlocks are avoided, but it is up to the implementor to guarantee protection by properly programming the extra shared resource module. Thus, serializers provide more mechanism than do monitors, at more cost, and a decision must be made as to which is more appropriate for given situations.

6. Conclusions

This paper has presented a method for evaluating synchronization mechanisms to determine how well they satisfy criteria such as expressive power, ease of use and modifiability. We have shown that by identifying the kinds of problems for which these mechanisms will be used, and carefully defining the properties in which we are interested, it is possible to develop a systematic method for assessing a construct's adherence to these requirements. Our evaluations of existing constructs show that the techniques described here have not only produced results that concur with our intuitive judgements about the mechanisms (drawn from long periods of experimenting and ad hoc testing), but have also provided additional information about weaknesses in mechanisms that allow us to predict which problems will be difficult to solve using a given mechanism. Thus, the information provided is important both to the designers of a mechanism, and to anyone needing to compare several mechanisms or select one for a given application.

Our analysis thus far has been limited to synchronization constructs for a shared resource model. We have not looked extensively at message-passing models, or more recent mechanisms, such as guarded commands[19] and the mechanism proposed by Hoare in "Communicating Sequential Processes"[20], which may be used for many of the same synchronization problems. Since these and similar constructs will probably be used extensively in distributed systems, it is important to be able to evaluate and compare them. The techniques presented in this paper may prove useful in these evaluations.

Acknowledgements

I would like to thank Barbara Liskov for her assistance in developing the ideas presented in this paper. My numerous discussions with Craig Schaffert and Russell Atkinson were also extremely helpful. Timothy Anderson, Gene Stark, Maurice Herlihy and Dean Brock provided many useful comments on earlier drafts of the paper.

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 and Proof Techniques for Serializers", IEEE Transactions on Software Engineering, (5, 1), Jan. 1979.

4. Atkinson, R., Private communication.

5. Bloom, T., "Synchronization Mechanisms for Modular Programming Languages", TR 211, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Jan. 1979.

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

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

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

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

10. 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.

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

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

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

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

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

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

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

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

19. Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM 18, 8 (August 1975), 453-457.

20. Hoare, C.A.R., "Communicating Sequential Processes", Comm. ACM (21, 8) August 78, 666-677.