HIGH LEVEL DESCRIPTION AND IMPLEMENTATION OF RESOURCE SCHEDULERS

Dennis W. Leinbaugh
The Ohio State University
Department of Computer and Information Science
Columbus, Ohio 43210

ABSTRACT

Resource sharing problems can be described in three basically independent components.

- The constraints the resource places upon sharing because of physical limitations and consistency requirements.
- The desired ordering of resource requests to achieve efficiency -- either efficiency of resource utilization or efficiency for processes making the requests.
- Modifications to the ordering, to prevent starvation of processes waiting for requests which might otherwise never receive service.

A high level description language to specify these components of resource sharing problems is introduced. An implementation that lends itself to mechanical synthesis is described. Synthesis of the scheduler code by-passes the long and error-prone process of someone doing the coding themselves. Proof techniques at the high level description level are introduced to show how to prove schedulers, synthesized from their description, are or are not deadlock and starvation free. Solutions to the classical resource sharing problems of producer/consumer, reader/writer, and disk scheduler (to the sector level) are shown to illustrate the expressiveness of this description language.

Key Words: Resource sharing, resource scheduling, protected resource, process synchronization, specification language, nonprocedural language, and process starvation.

INTRODUCTION

An important activity in any multiple user system is resource sharing. Many schemes have been proposed and developed to aid in resource sharing. Monitors [HOAR74] and serializers [HEWI79] were designed primarily to enforce cooperation among users sharing resources. These schemes provide primitives and language structures which make it relatively easy to write code to enforce the necessary rules and desired policies upon resource sharing.

This work describes how to directly specify the resource sharing rules needed and policies wanted for resource sharing problems. The code to enforce these rules and policies can then be automatically generated (synthesized) from the high level description provided. The advantages are clear. Since the rules and policies are specified directly, it is known exactly what they are and that they are enforced. Finally, proof techniques are shown to determine if solutions are deadlock and starvation free.

Ramamritham and Keller [RAMA80] attacked the same problem [LEIN81]. The specification language they describe for synchronizers is at a lower level than that described here and consequently may not be as easy to read or write. Synchronizers do not allow concurrency of operations which modify the same state variable, whereas the schedulers described here allow as much concurrency as the resource does.

Proof techniques for monitors [HOWA76] and serializers [HEWI79] involve program proving because these schemes use synchronization procedures. Synchronizers use predicate calculus with temporal operators [RAMA81]. Proofs with both these techniques can be long and complicated because of the low level upon which they are based. The structure and high level description of schedulers, on the other hand, make it possible to do proofs directly at that level.

The specification of resource sharing consists of three major independent components. First, the resource invariant to determine if an additional request can be accepted by the resource and still maintain resource consistency and correct servicing of requests. Second, the request ordering policy to determine which of several acceptable requests will be next serviced. Third, modifications to the ordering policy to avoid endless waits by some requests and hence avoid starvation of the processes waiting on the completion of these requests.

The ability to specify resource sharing in independent components has great advantages. It reduces the problem of resource sharing into simpler components, making it easier to correctly

specify each. These are natural divisions in the sharing problem and allow a person to more clearly deal with each separately.

HIGH LEVEL DESCRIPTION OF RESOURCE SHARING

A resource is viewed as a collection of operations upon an object. A scheduler for that resource is a process that has exclusive use of the object. The scheduler receives operation requests from other processes and forwards them to the resource in accordance with the description of resource sharing. A response message is sent to the requesting process after the scheduler receives the completion response from the operation (see Figure 1).

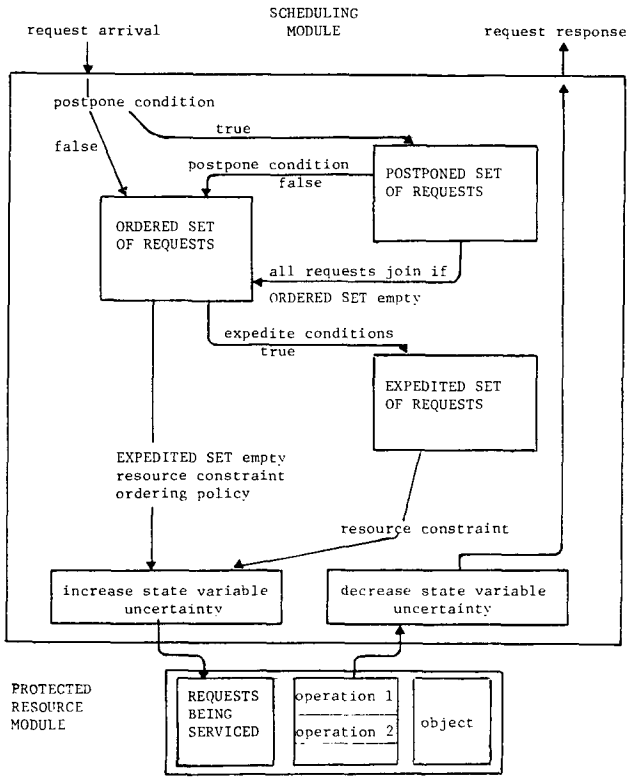

Figure 1: Structure and Scheduling Strategy of Schedulers

The description language allows specification of six modular components to solve resource sharing problems:

- Definition of request messages,
- Declaration and modification of resource state variables,
- The resource invariant,
- The ordering policy,
- The postponement policy, and
- The expedite policy.

The remainder of this section is a brief overview of the syntax and semantics of the description language. A complete syntax and semantics can be found in [LEIN81], although that syntax as well as that used here is intended to be only suggestive. The syntax is verbose to enhance understandability in reading the examples in this paper without first studying the syntax and semantics. Abstract resource types could easily be defined, just as some languages allow user defined abstract data types, but is omitted for brevity. While reading this section, Figures 2 through 4 can be examined for examples of the language features described.

Definition of Request Messages

The REQUEST DECLARATIONS section defines the fields of the request messages that make up the requests. A request is a record of fields. A request's type is identified by values in one or more of those fields.

Scheduler Variables

Variables are used in the remaining scheduler components so are described prior to those components. There are three categories of variables.

- Set Counting Variables: WAITING.*requesttype*, EXPEDITED.*requesttype*, and ACTIVE.*requesttype* keep track of the numbers of requests of type, *requesttype*, residing in the Waiting Set, Expedited Set, and Active Set, respectively. The number of requests postponed is not available, to be consistent with the concept that postponed requests are ignored.
- History Variables: LASTACTIVE.*requesttype*.*fieldname* keeps track of the value in the field, *fieldname*, of the last request of type, *requesttype*, that has been sent to the resource for service.
- Resource State Variables: Resource state variables are the only category of variables that are not built-in. State variables are to reflect the known state of the resource. The state of the resource changes as a consequence of operations performed upon it.

Declaration and Modification of Resource State Variables

Resource state variables are defined and assigned initial values. The PROCESSING section describes what operation each request type will perform and how the state of the resource will change UPON SERVICE of the request. However, the exact time of the change is not known; both because there are delays in sending requests to the resource and receiving replies and because even within the performance of the operation itself there will usually be a transitional period during which the resource state changes from the old value to the new value. The value of each state variable is therefore kept as an uncertainty range of the possible values it could be. This uncertainty is increased whenever a request is sent to the resource and decreased whenever a reply is received back from the resource. There should be no uncertainty in the value of a state variable when the resource is idle. This uncertainty reflects exactly the information the scheduler can know about the resource and greatly simplifies specifying schedulers that allow concurrency of operations.

Resource Invariant

The resource invariant describes the legal operating range of the resource. The only

consideration is that a request not be sent to the resource if it causes an inconsistency or other illegal action. To enforce this concept, only information about the resource itself is allowed in the resource invariant. This means WAITING and EXPEDITED counts are not allowed. A request fits the resource invariant if its entering the resource leaves the resource invariant true (for all values of state variables within their uncertainty ranges).

Ordering Policies

In addition to maintaining proper operation of the resource, it is often desired to impose an ordering between those requests that fit the resource invariant. This ordering is probably to achieve efficiency -- either efficiency in resource operation or efficiency in the response time to requesting processes. In the absence of any other ordering specifications, the default ordering is always first in first out. There are two types of ordering that can be specified.

- Ordering between different types of requests.
- Ordering between different requests of the same type.

Ordering between types is easily specified by defining the priority (weak priority) between the different types. Ordering between requests of the same type is determined by field values within the requests and information about the resource. This type of ordering could be achieved with built-in ordering algorithms; however, greater flexibility can be achieved by using a high level description to specify the needed ordering algorithms.

Specification of Ordering Algorithms

An ordering algorithm is nonprocedurally defined in three components.

- Declaration of sets to hold requests,
- Conditions to determine which set a request enters, and
- Search techniques for locating the next request for service.

Since the examples given later utilize only one set, managing additional sets is not discussed. The set declaration specifies an expression, based upon the fields in each request, to determine the request's position within the set. The search techniques consist of one or more modes of search. Each mode is defined in three parts.

- Initial positioning within the set to begin the search.
- For what condition the search position should be updated and to what.
- At the end of the search, what mode to switch to or how to re-initialize the search within the same mode.

This description is sketchy but adequate to understand the examples.

Ordering specifications, while they may improve efficiency, can easily introduce starvation of some requests. The resource invariant itself can make starvation possible. Starvation of a request is defined as the condition where that request, under possible circumstances, simply never gets serviced but under different circumstances of request arrivals and service completions could be serviced. Rather than complicate the ordering policy, modifications to the ordering are made to avoid starvation. Two types of modifications are possible.

- Postpone requests from consideration.
- Expedite requests to be next for service.

Postpone Policies

The main reason that some requests starve is that other later arriving requests keep being serviced before them. One method to avoid this starvation is to identify what arriving requests may be responsible for this starvation and ignore them (temporarily). The POSTPONE statement attaches a condition to each type of request to be postponed. The postponement conditions may involve the current resource state and a consideration of other waiting requests. A request can only be postponed when it initially arrives and then only until its postpone condition becomes false or until the ordered set becomes empty.

Expedite Policies

Another method to avoid starvation is to identify the requests that may otherwise starve and make sure they are serviced. The EXPEDITE statement attaches a condition to each type of request that can be expedited. If that condition ever becomes true while the request is waiting for service, it is expedited -- making it next in line for service ahead of any requests still waiting in the ordered set.

In the absence of other measures, one very useful measure of how unfairly a request is being treated (and hence possibly being starved) is a count of how many times the oldest request of one type has been passed over in preference to giving service to later arriving requests of another type. This measure is expressed as

TIMES *requesttype1* HAS PASSED *requesttype2*.

A postpone or expedite condition is considered true if it evaluates true for some value of each state variable within its uncertainty range. The resource may actually be in that state. Also, it is better to postpone or expedite an extra request than to miss one.

IMPLEMENTATION STRATEGY

The overall scheduling strategy is illustrated in Figure 1. A process issues a request for service by sending a request message to the Scheduling Module. The Scheduling Module implements the high level resource sharing specifications, and forwards a request to the Protected Resource Module when it is to be performed. When a request completes service, the Scheduling Module is notified and the process receives a response message.

Requests sent to the scheduling module can only reside as postponed requests, ordered requests, expedited requests, or requests being serviced. Only a request that leaves the resource invariant true when it starts service is accepted into the resource. As a request joins the requests being serviced, the uncertainty of the resource state is increased. When the

resource replies to the request, the resource has been updated so the uncertainty of the resource state is decreased.

If there are expedited requests, then the request that was earliest expedited is the only candidate for service. Otherwise, the ordering rules determine which of the serviceable requests will next be serviced. An ordered request will join the expedited requests if there is an expedite condition for it that evaluates true.

A request is postponed if a postpone condition for it evaluates true at the time the request arrives at the scheduler. A request remains postponed either until the postpone condition for it evaluates false, at which time it joins the ordered requests; or until the ordered set becomes empty, at which time all postponed requests join the ordered requests.

Requests are considered for postponement only upon entry to the scheduler for two reasons. It is difficult for the person describing the scheduling to visualize the cycling that can occur if postponement is allowed anytime. Not only that, but this cycling can introduce starvation of the very requests that are postponed to prevent starvation of other requests. All postponed requests join the ordered set if the ordered set becomes empty because otherwise the postponed set would itself need a postponed set to avoid starvation of requests in the postponed set.

Order of Condition Testing

Conceptually the conditions are checked in the following order and rechecked after each action taken.

1. check for request to expedite,
2. check for postponed request to lose postponed status,
3. check for request to enter service,
4. check for request to complete service, and
5. check for new arrival and whether it should join postponed requests or ordered requests.

Most solutions do not involve both expedite and postpone, but since they are usually included to help prevent starvation it is more important to recognize a request to expedite than a request to postpone. This is because we are expediting the request that is (potentially) being starved rather than the more indirect approach of postponing one that may be starving another. It is important that both expedite and postpone be checked before checking for other actions to insure that the condition for expedite or postpone does not become true but is changed before it can be checked.

Next in importance is to check for request completions and for requests that can enter service. It does not make much difference but is more orderly if one first checks for a request to enter service. Normally the scheduler is thought of as performing its functions in a short period of time in relation to length of service of requests; otherwise less energy should be spent in the scheduling by using a simpler scheduling policy. Consequently, most of the time a request will not complete while the scheduler is still working. By starting new requests before recognizing the completion of requests it is simply made uniform that requests never complete while the scheduler is working.

Finally, new arrivals are checked. This strategy enables the scheduler to "take care of old business" prior to starting new business.

Efficiency of Condition Testing

For efficiency it is necessary that re-evaluation of conditions not be done excessively. This is accomplished in two ways. First, a request needs to be re-examined only when an event described above occurs and then only if that event may have altered the conditions associated with the request. Second, requests can be organized so only a few requests (often only one) need be examined each time an event occurs. The remainder of this section describes this organization, but a more thorough explanation is in [LEIN81].

Postponed requests can be partitioned into subsets of requests where the postpone condition for all requests in each subset evaluates identically. To determine if any request should no longer be postponed, only a representative from each subset (the oldest request) need be examined. Usually the number of subsets will be small -- the disk example has only one postponed subset.

Ordered requests need to be organized two ways. The order/resource-invariant organization first partitions requests according to the ordering and then partitions each of those partitions according to the resource invariant. The ordering partitions are either the request types involved if the ordering is between types or the partitions defined by the ordering algorithm used. If more than one level of ordering is specified, then the partitions are themselves partitioned. The partitions imposed by the resource invariant are determined by using the preconditions derived from them. The precondition for a certain type of request is that condition that when true and then a request of that type is started then the resource invariant remains true.

The ordered requests are also separately organized according to the expedite conditions. Often this expedite organization is contained in the order/resource-invariant organization thus necessitating only one organization on the ordered set.

Code synthesis from a high level solution can be done mechanically to produce an implementation of that solution as described above. Such a synthesizer is being developed for this description language.

RESOURCE SCHEDULING PROBLEMS

Three examples are given to illustrate important aspects of the description language.

Producer/Consumer Problem with Bounded Buffer

This solution (Figure 2) to the Producer/Consumer problem [HOAR74] illustrates the use of the resource invariant and state variables. Assume operations are written to allow an insert and a remove operation to be performed concurrently.

```
REQUEST DECLARATIONS
            REQUEST FIELDS      type      CHARACTER(1)
                                item      CHARACTER(99)

            REQUEST TYPES       insertitem HAS  type = 'I'
                                removeitem HAS  type = 'R'

DECLARE STATE VARIABLES         #items    INITIALLY 0

PROCESSING
    insertitem PROCESSED BY insertroutine
                    UPON SERVICE   #items := #items +1

    removeitem PROCESSED BY removeroutine
                    UPON SERVICE   #items := #items -1

RESOURCE INVARIANT
    ACTIVE.insertitem ≤ 1      AND      ACTIVE.removeitem ≤ 1
AND 0 ≤ #items     AND     #items ≤ 10
```

Figure 2. Producer/Consumer Problem with Capability to Save 10 Produced Items

Consequently the resource invariant specifies that at most one insert request and one remove request can be active in the resource at once.

The state variable *#items* is used to keep track of the number of items in the resource buffer. The insert routine adds another item to the buffer; thus it is specified that upon service *#items* is increased by one. Likewise, upon performing the remove routine, *#items* decreases by one. Since the resource uses a 10 slot buffer, the resource limit $0 \leq \#items \leq 10$ is necessary to prevent overfilling or overemptying the buffer.

This looks (and is) straightforward; however, this is only because uncertainty-range state variables are used. Solutions with previous techniques, if they allowed concurrency, would require two state varaibles -- one to reflect the number of positions known to be empty and the second to reflect the number of positions known to be full. An insert in such a solution tests and decrements the known-empty variable before beginning the insert operation and increments the known-full variable when done. This is unnecessary here because the uncertainty-range variable, *#items*, reflects exactly what is known of the resource. Any request that might take *#items*' uncertainty outside the range 0 to 10 is forced to wait.

For example, if no requests are active and there are 9 items in the buffer, then the uncertainty range is [9,9] -- the possible values for *#items* is only 9. A remove request can start, making the uncertainty range [8,9], and an insert request can start, making the range [8,10]. This means there are anywhere from 8 to 10 items in the buffer. If the insert finishes first, the range is reduced to [9,10] and another insert will not be allowed to start. When the remove completes, however, the range becomes [9,9] and another insert request could be started.

Reader/Writer Problem, Reader Priority with Designated Writer

This solution (Figure 3) to the Reader/Writer problem [COUR71] illustrates the use of ordering for reader priority and the use of expedite to avoid starvation of write requests. It uses set counting variables to determine when to expedite.

```
REQUEST DECLARATIONS

    REQUEST FIELDS      type            CHARACTER(1)
                        directions      CHARACTER(63)

    REQUEST TYPES       readrequest  HAS type = 'R'
                        writerequest HAS type = 'W'

PROCESSING
    readrequest  PROCESSED BY readroutine
    writerequest PROCESSED BY writeroutine

RESOURCE INVARIANT
    ACTIVE.writerequest ≤ 1  AND  ACTIVE.readrequest = 0
    OR
    ACTIVE.writerequest = 0

ORDERING readrequest  BEFORE writerequest

EXPEDITE writerequest IF EXPEDITED.writerequest = 0
                      AND ACTIVE.writerequest = 0
                      AND WAITING.readrequest = 0
```

Figure 3: Reader/Writer Problem: Reader Priority and Designated Writer

The resource invariant specifies that either one write request can be active or any number of read requests may be active in the resource at once.

Ordering specifies that read requests are to come before write requests. This means that if either type can use the resource, then a read request becomes active next. This ordering could result in starvation of write requests if there were enough read requests continually desiring service. Expedite therefore is used to select a write request which is potentially starving and designate it to be next. In this solution, a write request is expedited if currently none are expedited or active and if there are no read requests waiting to be serviced. A later section proves that this solution is deadlock and starvation free.

Disk Scheduler

This disk scheduler solution (Figure 4) illustrates postponement and ordering algorithms. It schedules requests to the disk such that the elevator algorithm is used for cylinder head positioning [HOAR74]. Within the same cylinder, the requests are ordered by sector address to service as many requests as possible per disk revolution. Writes to the same sector are selected before reads to insure the most up-to-date information is read. Finally, starvation of requests is prevented by insuring that the disk cannot remain at the same cylinder forever.

Ordering is achieved by making the elevator algorithm on cylinder addresses the primary ordering.

The secondary ordering is the scan algorithm which, within the same cylinder, selects one request per sector. The tertiary ordering, within the same cylinder and sector, is write requests before read requests.

The ordering algorithms, rather than being built-in, are also defined. Both specify a single set in which requests can reside in address order. The scan algorithm has a single extraction mode of first selecting the request with lowest (sector) address, then working through the sectors selecting one request per sector, and finally resetting to the lowest address again after the end of the sectors is reached. The elevator algorithm has two

```
REQUEST DECLARATIONS
     REQUEST FIELDS  type          CHARACTER(1)
                     cyl-addr      CHARACTER(3)
                     sector-addr   CHARACTER(2)
                     data          CHARACTER(256)

     REQUEST TYPES
          readrequest  HAS type = 'R'
          writerequest HAS type = 'W'

PROCESSING
     readrequest  PROCESSED BY diskdriver
     writerequest PROCESSED BY diskdriver

RESOURCE CONSTRAINTS
     ACTIVE.readrequest  + ACTIVE.writerequest ≤ 1

ORDERING
     FIRST BY elevator      ALGORITHM ON cyl-addr
     THEN BY scan           ALGORITHM ON sector-addr
     THEN BY writerequest   BEFORE readrequest

POSTPONE
     readrequest  IF THIS-REQUEST.cyl-addr  = LAST-ACTIVE.cyl-addr
     writerequest IF THIS-REQUEST.cyl-addr  = LAST-ACTIVE.cyl-addr
```

```
scan ON address
DECLARE rset ORDERED BY address
ENTER   rset
SEARCH TECHNIQUE
       INITIALIZE TO MINIMUM
       UPDATE TO NEXT HIGHER POSITION AFTER EACH REQUEST
       AT END INITIALIZE TO MINIMUM
```

```
elevator ON address
DELCARE rset ORDERED BY address
ENTER   rset
SEARCH TECHNIQUE
upmode:  INITIALIZE TO MINIMUM
         UPDATE TO NEXT HIGHER POSITION WHEN POSITION EMPTY
         AT END SELECT downmode

downmode:  INITIALIZE TO MAXIMUM
           UPDATE TO NEXT LOWER POSITION WHEN POSITION EMPTY
           AT END SELECT upmode
```

Figure 4. Disk Scheduler

extraction modes: up-mode to go from low (cylinder) addresses to high addresses and down-mode to go back from high to low addresses. The algorithm stays on the same cylinder until all requests at that cylinder are serviced.

Notice, that when the ordering is combined as it is here, there is one set of requests as defined by the elevator algorithm with one partition per cylinder address. Each of these partitions is considered a set by the scan algorithm and subdivided one partition per sector address. Finally, each of these (sub)partitions is organized into a write subset and a read subset.

Starvation would be possible if not for the postpone statement because, if more requests for the same cylinder arrive often enough, the elevator algorithm could remain "stuck" at the same cylinder address forever and starve requests for all other cylinders. Also, if enough of these were write requests to the same sector, then any reads for that sector would also starve. Starvation is prevented by postponing newly arrived requests if they are to the cylinder currently being serviced. Consequently, all requests already outstanding for that cylinder will be handled. The elevator algorithm will then proceed to the next cylinder and those postponed requests will then join the other ordered requests to be handled on the next sweep of the elevator algorithm.

PROOF TECHNIQUES FOR SCHEDULERS

The ability to easily express the solution to a resource sharing problem and to synthesize its enforcement from that description is important because that makes it easier to write, easier to understand, easier to change, and less error-prone. Also important is the ability to prove properties about a solution. Only in this way can one be sure that the solution has the properties intended.

The structure and high level description of schedulers makes it possible to do proofs directly at that level. The reader/write problem of Figure 2 is shown to be deadlock and starvation free to illustrate the technique. Any request sent to the resource is assumed to complete eventually.

Deadlock Free

A request is deadlocked only if there is no future sequence of events which would result in the request being serviced.

This is trivial for this problem since the resource invariant consists only of concurrency limitations. Assume that some requests deadlock and also that requests quit arriving. Evenutally the resource will go idle and only the deadlocked requests will remain. However, since any request can be serviced when the resource is idle, no requests can be left waiting while the resource sits idle. Consequently, there is no deadlock.

Starvation Free

The technique to show lack of starvation of any request is to follow representative requests and show that they make progress and are eventually serviced.

In this reader/writer problem, different read (write) requests are differentiated only by arrival time. Hence, only the oldest read (write) request need be examined because if the oldest is not starved then none are because each in turn becomes the oldest.

Assume that the oldest *readrequest* is being starved. Then no *readrequest*s are active but *readrequest*s are waiting.

Case R.1 (*writerequest* active)
No *readrequest* can be started but *writerequest*s can not be expedited either, because *readrequest*s are waiting.

Case R.2 (*writerequest* finishing)
This makes the resource idle. No *writerequest*s can be expedited because *readrequest*s are waiting. If any *writerequest*s are already expedited, then each of these in turn is serviced. Eventually the last expedited *writerequest* has been serviced and the resource is idle. Now the oldest *readrequest* is selected for service because of ordering.

Therefore, *readrequest*s cannot starve.

Assume the oldest *writerequest* being starved. Then no *writerequest* is active, none are expedited,

but *writerequests* are waiting.

Case W.1 (*readrequest*(s) active, none waiting)
The expedite condition is tested first and consequently a *writerequest* is expedited. No requests can get ahead of the expedited one and eventually currently active *readrequests* finish and the *writerequest* begins service.

Case W.2 (*readrequest*(s) active, other *readrequests* waiting)
The waiting *writerequests* can make no progress but the waiting *readrequests* can each enter service since allowed by resource invariant. Eventually (soon) all waiting *readrequests* have entered the resource and no new *readrequests* will have been recognized since new requests is the last condition tested for. Consequently, when the last waiting *readrequest* enters service, the expedite condition for *writerequests* becomes true and the oldest *writerequest* is expedited. The expedited *writerequest* will be serviced.

Therefore, *writerequests* do not starve.

Notice that only the condition WAITING.*readrequest* = 0 in the expedite was actually needed in this proof to show no starvation.

Knowing the deadlock and starvation properties of individual schedulers does not resolve all questions about the processes which use them. For instance, if two schedulers are deadlock free but have starvation potential, then processes may use them in such a way that the processes deadlock.

CONCLUSIONS

The specification of resource sharing can be described modularity in components which are easy and natural for people to deal with: the resource invariant, ordering policy, expedite policy, and postponement policy. Resource state variables, kept as uncertainty ranges reflecting what is known of the actual resource state, are also responsible for straightforward specifications of schedulers. These specifications can then be mechanically converted into code thus synthesizing the scheduler specified. Any errors in the scheduler are in the specification itself. It is much easier to find errors at this level than the code level. Deadlock and starvation proofs for individual schedulers can be carried out at this description level.

Additional features are needed to make the language more complete such as allowing transformation of request messages from one format into another and allowing the generation of more than one resource operation from a single request message. The language as presented, however, is complete enough to demonstrate its power and the ease with which one can write and understand scheduling solutions.

REFERENCES

[COUR71] Courtois, P. J., Heymans, F., and Parnas, D. L. Concurrent control with "Readers" and "Writers", CACM 14, 10, (Oct. 1971), 667-668.

[HEWI79] Hewitt, C. E. and Atkinson, R. R. Specification and proof techniques for Serializers, IEEE Transactions on Software Engineering SE-5, 1, (Jan. 1979), 10-23.

[HOAR74] Hoare, C. A. R. Monitors: an operating system structuring concept, CACM 17, 10, (Oct. 1974), 549-557.

[HOWA76] Howard, J. H. Proving Monitors, CACM 19, 5, (May 1976), 273-279.

[LEIN81] Leinbaugh, D. W. High level specification and implementation of resource sharing, The Ohio State University, Technical Report OSU-CISRC-TR-81-3, (Feb. 1981), 1-22.

[RAMA80] Ramamritham, K. and Keller, R. M. Specification and synthesis of Synchronizers, Proc. 1980 International Conference on Parallel Processing, (Aug. 1980), 311-321.

[RAMA81] Ramamritham, K. Specification and synthesis of Synchronizers, Ph.D. Thesis, The University of Utah, (June, 1981), 1-214.