MECHANISMS FOR SPECIFYING SCHEDULING POLICIES*

by

F. B. Schneider
and
A. J. Bernstein

TR 79-365

Computer Science Department
Cornell University
Ithaca, New York 14853

Computer Science Department
SUNY at Stony Brook
Stony Brook, New York 11794

*This work is partially supported by NSF grant MCS 76-04828 and MCS 76-22360.

MECHANISMS FOR SPECIFYING SCHEDULING POLICIES*

F. B. Schneider[1]
A. J. Bernstein[2]
Jan. 8, 1979

ABSTRACT

Extensions to concurrent programming languages are presented which allow control of scheduling policies previously defined by the run-time support system. It is shown that the use of these mechanisms simplifies the solutions of concurrent programming problems. In addition, the proposed extensions allow easy identification of those aspects of a program concerned with performance, thereby making programs easier to read and understand.

Keywords: Concurrent Pascal, Monitors, Communicating Sequential Processes, Operating Systems, Scheduling Algorithms.

---

* This work is partially supported by NSF grant MCS 76-04828 and MCS 76-22360.

1. Computer Science Department, Upson Hall, Cornell University, Ithaca, New York 14853.
2. Department of Computer Science, S.U.N.Y. at Stony Brook, Stony Brook, New York 11794.

MECHANISMS FOR SPECIFYING SCHEDULING POLICIES

F. B. Schneider

A. J. Bernstein

1. Introduction

It is well known that the construction of software is made considerably easier by the use of high level languages. Programs written in high level languages are easier to understand, modify (hence maintain) and formally verify than their assembly language counterparts. Consequently, a good deal of research has been devoted to the definition of high level languages suitable for various environments and applications. Of particular interest here are those languages and language proposals intended for writing asynchronous systems. Notable examples of such languages are Concurrent Pascal [Brin75] and Modula [Wirt77], which are based on the monitor construct [Brin73] [Hoar74], and the language proposals contained in papers titled "Communicating Sequential Processes" [Hoar78] and "Distributed Processes" [Brin78a], which are based on a message passing synchronization scheme.

Common to many recent programming languages is the inclusion of syntactic features that make explicit, and often

help to enforce, various correctness criteria. For example, in Concurrent Pascal access to variables declared within a monitor is guaranteed to be mutually exclusive. It follows from this and certain syntactic constraints of the language that all shared data in a Concurrent Pascal program is protected from simultaneous access [Bern78] [Brin75].

The guarded command language proposal of Dijkstra [Dijk76] provides another illustration of the trend to make explicit various correctness criteria. Central to this language proposal is the enumeration of the logical preconditions necessary before the execution of a command can proceed. A guarded command has the form:

G -> C

where G, the guard, is a boolean expression, and C is a command list. C is only executed provided G is true. Guarded commands may be combined into an alternative command having the form (the notation in [Hoar78] is used)

[G1 -> C1 G2 -> C2 ... Gn -> Cn]

This specifies the execution of exactly one of its constituent guarded commands. Note that if more than one guard is true then one of the corresponding command lists is selected arbitrarily. The alternative command is undefined if none of the guards of the constituent guarded commands (G1, G2, ..., Gn) is true. Guarded commands may also be combined

into a repetitive command

*[G1 -> C1 G2 -> C2 ... Gn -> Cn]

which specifies as many executions of the alternative command as possible. Consequently, when all constituent guards are false, the repetitive command terminates. Notice that in a guarded command program the logical assertions that would be part of a formal proof of that program are actually embedded in the program code as the guards. Typically, a guarded command program is developed along with its proof of correctness.

In addition to syntactic features that make correctness criteria explicit, high level languages usually provide facilities for the localization of portions of code associated with the various facets of the program. For example, the monitor construct may be viewed as a form of abstract data type [Lisk74]. Users of a monitor need only know the semantics of the operations supported by that monitor. They need not know the details of the implementation of those operations. Similarly, when reading the monitor code, one need only be concerned that the procedures implement the abstract operations, not with the environment from which procedures are invoked, or how they are used. This separation of concerns allows a progammer to deal only with details relevant to the object being considered.

Despite these advantages, high level languages deprive

the programmer of the ability to specify certain policies. In many situations this is an advantage, as the programmer has many fewer details with which he must be concerned. For example, most compilers decide register allocations during code generation, thus freeing the programmer from this burdensome task. In other situations, however, the inability to specify policy leads to awkward system structure. A discussion of some of the difficulties encountered by the Concurrent Pascal programmer because memory management and process scheduling can not be controlled in that language can be found in [Lohr77].

This paper addresses one aspect of this problem by proposing mechanisms that allow the programmer to control certain run-time scheduling policies. In particular, syntactic extensions to Concurrent Pascal and Communicating Sequential Processes are discussed which allow the programmer to specify the ordering of certain events that would otherwise be controlled from within the kernel. It is shown that the use of these extensions simplifies the structure required for the solution of concurrent programming problems. In addition, the mechanisms force the programmer to localize all computations relevant to scheduling. This enhances the readability of such programs, as it allows easy identification of those aspects of a program that are concerned with performance optimization.

Two extensions to Concurrent Pascal are discussed

first. A generalization of the priority wait construct of Hoare [Hoar74] is presented, which allows localization of the program code used to specify the order in which processes suspended on a given condition queue are awakened. Secondly, a facility to allow an ordering to be specified for processes suspended at monitor entry is discussed. Then, an extension to Communicating Sequential Processes that allows specification of the order in which guards are evaluated is proposed.

2. Extensions to Concurrent Pascal

In this section, two syntactic constructs are presented to allow the binding of priority computations to an operation that may suspend processes.

In a Concurrent Pascal program a process may be suspended at wait statements as well as at the mutual exclusion code used to guard monitor entry. Wait statements are employed to delay a process in a particular monitor when the state of that monitor is not conducive to continued execution. Mutual exclusion is used to guarantee the integrity of the permament variables of a monitor, since they may otherwise be subject to concurrent access.

The inability to specify the order in which suspended processes are awakened at monitor entry or wait statements can create difficulties in the design of operating systems. One may wish to control this order to maximize throughput or

to accomodate real time constraints imposed on a system by its operating environment. Hoare [Hoar74] has proposed a wait statement that includes a priority parameter to control this order. This notion can be generalized by allowing the declaration of a priority function in a monitor and binding it to either a wait statement or to monitor entry. The function computes an integer-valued priority based on the state of the variables in the monitor.

Under normal circumstances the designer of a monitor must guarantee that it functions correctly for all possible choices of the sequence in which suspended processes are awakened. The imposition of a priority on the awakening of these processes serves only to eliminate some of those sequences. Thus, assuming the monitor functions correctly without priority functions if priority functions have no side effects and always terminate, then the correctness of the monitor cannot be affected by their introduction. It is simple to guarantee at compile time that these functions will execute without side effects. This is done by insisting that all parameters are passed by value, prohibiting assignments to any variables other than those local to the function itself (which are reinstantiated each time the function is invoked) and by prohibiting it from invoking other functions. Termination may be proved by using one of the standard techniques [Dijk76].

2.1. Wait Statements with Priority Functions

Consider a monitor that controls a number of shared buffers to support interprocess communication. A process wishing to deposit a message should be suspended if no buffer is free. A wait statement can be employed to accomplish this task. Assuming the conditional wait proposal of Kessels [Kess77], a condition of the form:

```
notfull : condition (num_empty_buff > 0);
```

could be declared where the variable num_empty_buff records the number of empty buffers. The first statement of a monitor procedure for depositing a message would then be:

```
notfull . wait
```

Note that in this example it has been assumed that the order in which suspended processes are reactivated is immaterial. On occasion, however, one might wish to control that order. In this case, when a request enters a monitor, a computation can be made that yields sufficient information to order the waiting processes using the values of the parameters passed to the monitor (the attributes of the call) and the values of the permanent variables in the monitor (the state of the monitor). Such computations will be called scheduling policies. Scheduling policies are implemented in Concurrent Pascal [Brin75] by defining an array

(vector) of queues and computing an index which indicates on which particular queue a process should be suspended. When a process is to be reactivated, the queue with the lowest index that has a process suspended on it is signalled. At most one process may be suspended on a queue at any time. Hoare [Hoar74] provides an analogous construct for his monitor definition. The equivalent of a queue is called a condition and differs from a queue in that more than one process at a time may be suspended on it. To order the processes waiting on a condition, the priority wait statement, wait(p), is provided. This causes suspension of the executing process and orders the "queue" associated with the condition so that the suspended process with the lowest priority, p, is always the next to be awakened. Using this construct, a priority is computed rather than an index into a vector of queues.

Notice that in both of these schemes, the priority computation is not separated from the computation that implements the abstract operation supported by the monitor procedure. To accomplish an explicit separation of these concerns, the following mechanism is proposed. A use clause is added to the syntax of the condition declaration to (optionally) allow a priority calculation to be associated with a wait statement. This may be thought of as a generalization of the Hoare priority wait scheme [Hoar74] and the Kessels conditional wait proposal [Kess77]. A condition is declared as follows:

<cond name> : condition [<bool exp>] use <priority function>;

The syntax and semantics of the wait statement is as follows:

Syntax:

<cond name> . wait(arg1, arg2, ..., argn)

Semantics:

> Upon execution of this statement, the boolean expression, <bool exp>, bound to <cond name> in its declaration is evaluated. If the value is true, execution continues; if false the process is suspended. In the latter case an entry is added to a queue associated with the condition. Each such entry represents a process suspended on that particular condition. In addition to the process name the values of arg1, arg2, ..., argn are stored in the entry. These are treated as value parameters and may be monitor entry parameters, expressions, or permanent monitor variables.
>
> Whenever a process exits the monitor or is suspended at a wait statement, the boolean expressions bound to any conditions upon which there are suspended processes are evaluated. If all have value false, then a new process may be granted entry to the monitor. If one is found that has value true, then the priority function referenced in the declaration of the condition is used to compute an integer value for each waiting process, using the values stored in the corresponding queue entries. The process with lowest priority suspended on the corresponding condition is granted control of the monitor.

Kessels [Kess77] requires that all conditions be declared among the permanent variables of a monitor and hence the associated boolean expression cannot involve parameters passed to the monitor by calling processes. This guarantees

that when a boolean expression has value true, any of the processes suspended on the associated condition is eligible to continue. Consequently, context switches are not required to evaluate conditions. This makes the mechanism fairly efficient.

A priority function, on the other hand, will generally be a computation that involves information local to each process and, as a result, will compute a different priority value in each case. This local information is passed to the function through its (value) parameters. It is generally not necessary to evaluate the function with each element of the queue to ascertain which process will have the minimum value every time a process is to be reactivated. If the priority function does not reference permanent monitor variables, but is written only in terms of the parameters passed to it, it may be computed once, at the time the process is suspended, since its value will not change. In this case, only the priority and the process name must be stored in the queue entry and the queue can be maintained in ascending order by priority. Note that permanent variables may be among the parameters passed to the priority function. Under this restriction, no additional context switches are required for priority function computations. If direct reference to the monitor's permanent variables occurs in the priority function body, then their changing values may result in changes to the priorities of the waiting processes and recomputation would be unavoidable. For reasons of ef-

ficiency, this is not recommended.

Lastly, note that the use of a scheduling algorithm will invalidate any assumptions about finite progress that are usually made about wait statements. This results from the fact that a scheduling discipline may continually pass over low priority processes in deference to those of higher priority. It is the programmer's responsibility to avoid indefinite overtaking when it is undesirable.

It appears that priority wait statements find most of their application in a particular type of system structure. This structure is characterized by the fact that a process that is suspended at such a wait statement is not awakened by a process returning to that monitor after having completed a nested call, but rather by a process that has just entered the monitor (usually at a different procedure). This is typical of applications where permission to access a resource is scheduled, but each access (i.e., call) to the resource is not. This situation is illustrated by the following solution to the first readers/writers problem [Cour71]. Figure 1 is the access graph of a program that synchronizes processes that are manipulating a shared database. A process wishing to read the database calls REGULATOR.STARTREAD, returns to the calling module, performs the read in one or more calls to the module containing the database, and then invokes REGULATOR.STOPREAD. A process wishing to write, first calls REGULATOR.STARTWRITE, returns

to the calling module, writes in the database in one or more calls, and then calls REGULATOR.STOPWRITE. The code for the REGULATOR monitor appears as Figure 2. When more than one writer is suspended, the next writer to be awakened is determined by the value of bid_amt, an entry parameter that indicates how much money the writer is willing to "pay" for the privilege of being the next writer of the database. A priority function bound to a wait statement is used to implement this ordering. Notice that suspended requests in REGULATOR are awakened by other requests entering the monitor (invoking STOPREAD or STOPWRITE), rather than a request returning to REGULATOR from a nested call.

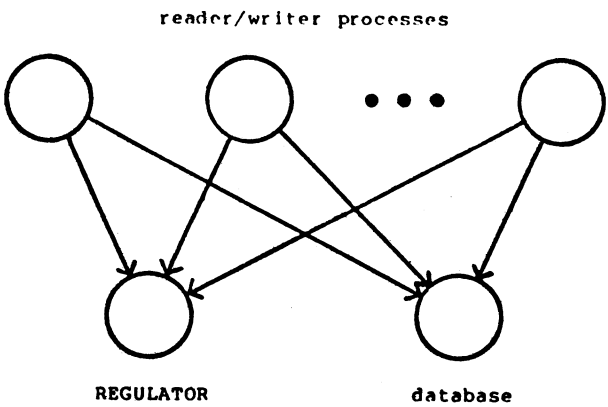

Figure 1 - Readers/Writers Solution

```
type REGULATOR = monitor;
     var w, r : integer;
       nowriters : condition (w = 0);
       nobodyatall : condition (w = 0 and r = 0)
            use highestbidder;
     function highestbidder (dollars : integer) : integer;
       begin
         highestbidder := - dollars
       end;
     procedure entry STARTREAD;
       begin
         nowriters . wait;
         r := r + 1
       end;
     procedure entry STOPREAD;
       begin
         r := r - 1
       end;
     procedure entry STARTWRITE (bid_amt : integer);
       begin
         nobodyatall . wait (bid_amt);
         w := 1
       end;
     procedure entry STOPWRITE;
       begin
         w := 0
       end;
     begin
       r := 0; w := 0
     end;
```

Figure 2 - REGULATOR Monitor Definition

2.2. Priority Functions at Monitor Entry

Consider a system in which a number of concurrently executing requests access some shared resource, such as a disk. Further, assume that at most one request may be using the disk at any time. Disk head seek times are usually very long compared with the actual data transfer time associated with a disk access. Consequently, more efficient disk utilization, as well as improved average waiting time for requests attempting to access the disk, can be realized by us-

ing a disk head scheduling algorithm such as the one described in [Hoar74]. In this algorithm, requests are ordered so that the disk head sweeps across the disk in one direction, then the other, analogous to the operation of an elevator in an office building.

The scheduling algorithm might be implemented using the same structure as discussed in the previous reader/writer example. In that case, in order to access the disk, it is required that a request first call the scheduler, which may then cause the caller to be suspended until a time when a disk transfer can be performed efficiently. Upon returning from the scheduler, the disk is called to perform the actual I/O operation. Lastly, the scheduler is called again to report the completion of the transfer.

For this application (and many others), this structure is undesirable for several reasons. First, note that the functions of scheduling and disk I/O are viewed by higher levels of the system as separate. This adds to the complexity of higher levels (i.e., users must observe the protocol of calling the scheduler before and after an access to the disk). Furthermore, it is not possible in general for a compiler to check that the required protocol will be observed. Undebugged programs that violate the protocol may cause the system to crash in a unpredictable way or may simply circumvent the scheduler and degrade system performance.

These difficulties would be overcome if a single func-

tion that both scheduled and accomplished an access to the resource were provided. User processes would not have an access right to the resource; this would be granted only to the scheduler. It would then be impossible for a user to directly access the resource, except through the scheduler.

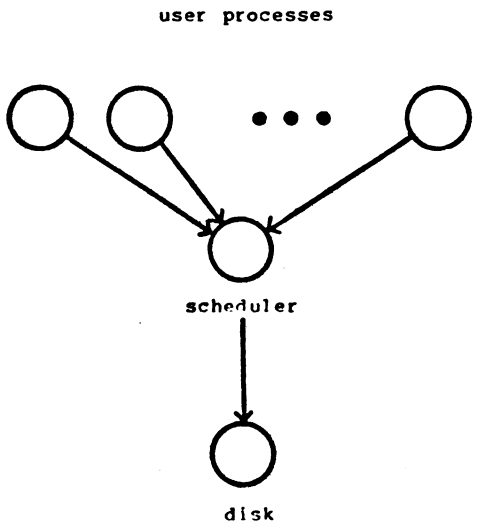

Figure 3 - Scheduling a Disk

Figure 3 exhibits this type of structure. This organization is unacceptable, however, because entry to the scheduler is prevented if an I/O operation is in progress (mutual exclusion is not released when the nested call from the scheduler is made). Thus, no real scheduling can take place because processes would be suspended at scheduler entry, instead of

within the monitor where an ordering on the suspended processes could be imposed by some priority wait mechanism. A monitor-like construct in which mutual exclusion is released when nested calls are made has been proposed to address this problem [Andr78]. However, the question of when to allow a process to reenter such a monitor after returning from a call has not been adequately resolved.

A priority function invoked at monitor entry can be used to solve this problem. A priority function is associated with each entry procedure by the use clause in the procedure heading. Whenever a process attempts to enter a monitor, while another process is actively executing within, the caller is blocked by the mutual exclusion associated with monitor entry. When the active process exits or is suspended at a wait statement in the monitor, entry by one of the blocked processes is permitted. This selection is made by evaluating the appropriate priority function for each blocked process and choosing the one with highest priority.

Figure 4 illustrates the use of this feature in implementing the disk head scheduling algorithm discussed earlier. Notice that the scheduler and disk modules have been combined. However, the code associated with scheduling is localized and disjoint from the code required to access the disk. The aesthetic advantages of two separate modules is therefore preserved.

```
type diskscheduler = monitor (diskdrive : disk);
     const disksize = D; (*number of tracks on disk*)
     var incr, lastincr : boolean; (* current and last
                                      direction of scan *)
         curaddr, lastaddr : integer; (* track number of
                                         current and last transfer *)
         tracks_scanned: integer; (* number of sweeps * disksize *)

     function priority (trkno : integer) : integer;
          begin
               if incr and (trkno > curaddr)
                  then priority := trkno + tracks_scanned
                  else if incr then
                     priority := disksize - trkno + disksize
                                 + tracks_scanned
                     else if not incr and trkn <curaddr
                        then priority := disksize - trkno
                                 + tracks_scanned
                        else priority := disksize + trkno
                                 + tracks_scanned
          end;

     procedure entry accessdisk (trkaddr : integer;
                                 var block : page;
                                 iotype : (read, write))
                    use priority (trkadder);
          begin
            lastaddr := curaddr;
            curaddr := trkadder;
            lastincr := incr;
            if curaddr > lastaddr then incr := true
              else incr := false;
            if not (lastincr = incr) then
              tracks_scanned := tracks_scanned + disksize;
          call io(trkadder, block, iotype);
          end;
     begin (*initialization*)
       incr := true;
       curaddr := 0;
       tracks_scanned := 0;
     end;
```

Figure 4 - Scheduling a Disk

The evaluation of the priority function for each waiting process can only be performed at a time when no process is executing in the monitor because permanent variables may be referenced. Thus, each time a process actually executing

in a monitor relinquishes control of it, the priority functions associated with all processes that called the monitor while that activity was taking place are evaluated and an entry is made on an entry queue associated with the monitor. Entries are arranged in ascending priority order and the process associated with the head entry is allowed to proceed.

It should be noted that when a priority function is used in this way, two context switches are required for each entering process if more than one process is attempting to enter the monitor at a given time; one for priority evaluation and a second for monitor entry. This contrasts with monitors that do not have priority functions associated with entry procedures and only require one context switch.

It appears that associating a priority function with monitor entry is useful in those situations where a suspended process is awakened by a process that is returning from a nested call. Execution in the nested module (e.g., the disk in Fig. 3) blocks further entry to the calling monitor (e.g., scheduler in Fig. 3) and creates the need to schedule processes that call that monitor prior to the return. This often occurs in applications where every access to a resource must be scheduled, as in disk I/O.

3. Extension to Communicating Sequential Processes

In a recent paper, Hoare [Hoar78] extended the concept of a guarded command for use in an asynchronous environment. In that language proposal processes communicate by using input and output commands. These commands have the form: <process name> ? <target variable> and <process name> ! <expression>, respectively. The <process name> identifies the process with which the communication is to take place. Communication occurs between two processes whenever:

(1) an input command in one process is executed that specifies as its source the process name of the other process;

(2) an output command of the other process is executed that specifies as its destination the process name of the first process; and

(3) the target variable of the input command matches the value denoted by the expression of the output command.

Thus, when two processes communicate by input and output commands either process may have to wait for the other. Input and output commands may appear in command lists. In addition, input commands may appear in guards. A guard, therefore, may consist of a (possibly empty) list of boolean expressions (possibly) followed by an input command. The guard fails if any of the boolean expressions have value false or if it contains an input command and the named pro-

cess has terminated.

If more than one guard is true in an alternative or repetitive statement, then the command list corresponding to one of them is arbitrarily selected and executed. The programmer has no control over this selection process. Hoare [Hoar78] specifies that an implementation should "ensure efficient execution and good response. For example, when input commands appear as guards, the command which corresponds to the earliest ready and matching output command should in general be preferred; and certainly, no executable and ready output command should be passed over unreasonably often." However, a programmer's inability to specify an order on the evaluation of guards in non-deterministic control structures may cause problems. Consider the following scenario [Kieb78]:

> Two processes, Update and Display, control the position of a point on a CRT screen. The update process periodically computes a new position for the point and communicates it to the Display process. Thus, Update has the form:
>
> ```
> Update:: [var x,y: integer;
>     *[... compute new values for x,y ...;
>      Display!(x,y)]    ]
> ```

The Display process constantly refreshes the

screen, changing the position of the point if Update has so instructed it.

```
Display:: [var u,v: integer;
     *[Update?(u,v) --> skip
       true --> ... refresh screen with (u,v) ...
     ] ]
```

If Update is ready to output the coordinates of a point then both guards in the repetitive command of Display are true. In this case the designer has has no control over which command list will be executed and must rely on the "fairness" of the implementation. However, if "fairness" implies selecting the first true guard (the Update output request) then if Update produces points at a rapid rate, it could prevent Display from refreshing the screen. If the guards are evaluated in the opposite order then the output from Update will always be ignored. If the selection is random then no bound can be placed on the number of successive times a particular true guard is selected. The situation illustrated here is typical of any application where a process has a background computation which can always be performed, in addition to one or more foreground computations which are dependent on the arrival of messages from other processes.

A designer is often able to specify a selection algorithm for the order in which guards are to be evaluated that

will realize some performance goals of the system. In the above example, round-robin evaluation of the guards would suffice. However, it is not difficult to imagine a more complex situation where some non-trivial computation that involved the recent history of the program is needed to determine the order in which to evaluate the guards. Thus, a general facility for this purpose appears to be useful.

As the Communicating Sequential Processes proposal is merely a partial specification of a language, the syntax for such a mechanism will not be offered here. Rather, only the semantic properties of a guard selector function (GSF) will be discussed.

First, provisions must exist to bind a GSF to a non-deterministic control structure (alternative or repetitive command). A mechanism similar to the use clause described above would suffice. Each GSF encapsulates two types of variables: local and permanent. The permanent variables are instantiated when the process in which the GSF is defined is created, and their values persist throughout the lifetime of that process. Local variables are instantiated when the execution of the non-deterministic control structure, to which the GSF is bound, is to be initiated. In addition, a GSF encapsulates three pieces of code. There is an initialization routine, called permanent initialization, that executes only once, when the process in which the GSF is defined is created. Presumably this routine will be used

to initialize the permanent variables of the GSF. Similarly, there is a routine, called local initialization, that executes when the command to which the GSF is bound commences. This routine can be used to initialize GSF local variables. Lastly, there is the selector routine, a routine that is invoked by the run time support software whenever the next guard to be evaluated is to be selected. This routine returns a value between 0 and N-1 each time it is invoked for a given non-deterministic control structure composed of N guarded commands. Note the difference between a GSF and a priority function, which yields an arbitrary integer.

As with priority functions, the introduction of a GSF should not complicate the formal verification of the correctness of a program. Thus, if a proof of correctness exists for the program without the GSF, it should still apply after the GSF has been introduced. Three constraints must be placed on the GSF to ensure this:

(1) Termination of each routine in the GSF must be proven.

(2) The GSF must have no side effects. Although the GSF may reference any variables visible to the process, it may only alter variables encapsulated in the GSF definition. Furthermore, these latter variables may be accessed only from within the GSF.

(3) After local initialization, N successive executions of the selector routine should produce a permutation of the integers between 0 and N-1. (Actually, the less stringent requirement that the selector produce each integer between 0 and N-1 after a finite number of invocations is all that is really required. In general, this would be more difficult to prove, however.)

Requirement (3) prohibits GSFs in which certain guard alternatives are never attempted. An alternative command referencing a GSF that violated requirement (3) might never complete execution, even though there were true guards. Requirements (1) and (2) are analogous to the restrictions on priority functions discussed previously. Note that (2) is easily checked at compile time.

The introduction of the GSF mechanism does not alter the meaning of a program. The GSF merely provides a way for the run time support system to choose guards for evaluation. Since the semantics of the non-deterministic control structures admit any selection order, the one generated by a GSF is surely permissable. In addition to specifying the guard selection order, which allows control of program performance, the GSF encapsulates those computations concerned with performance and scheduling, and segregates them from the rest of the program.

4. Conclusions

It should be noted that priority functions and guard selector functions have an additional benefit. The use of such functions aids in the construction of concurrent programs that exhibit reproducible behavior. It is surely in the interest of a programmer to have a language in which it is possible to reproduce any particular program execution, at will. Problems of this nature are treated in [Schn78] [Bern78b]. The constructs described above can be used to guarantee that functions and predicates constructed with non-deterministic control structures always yield identical results when evaluated with the same arguments [Deme78].

The addition of syntactic features to languages must be justified. In this paper, mechanisms to allow specification of scheduling policies have been proposed and a justification for them has been given. Although the constructs have been presented as additions to specific languages (Concurrent Pascal and Communicating Sequential Processes), they are intended to be general, and hence applicable to many concurrent programming languages. The priority function of Section 2 could be added, to good advantage, to any of the numerous language proposals [Andr79] [Wirt77] that employ monitors as a synchronization construct. Similarly, recent message passing based language proposals (E.g., [Brin78]) have problems which are similar to the one outlined above.

Acknowledgement

The authors are grateful to Professor G. Andrews for comments on an earlier draft of this paper.

REFERENCES

[Andr78]

Andrews, G. and J.R. McGraw, "Language Features for Process Interaction," Proc. of an ACM Conference on Language Design for Reliable Software, pp. 114-127.

[Andr79]

Andrews, G., "Synchronizing Resources" submitted for publication.

[Bern78a]

Bernstein, A.J. and F.B. Schneider, "The Integrity of Classes in Concurrent Pascal," submitted for publication.

[Bern78b]

Bernstein, A.J., F.B. Schneider, "Language Restrictions to Ensure Deterministic Behavior in Concurrent Systems," Proceedings of the 3rd Jerusalem Conference on Information Technology, Jerusalem, Israel, August 1978.

[Brin73]

Brinch Hansen, P., Operating System Principles, Prentice Hall, Inc., Englewood Cliffs, N.J., 1973.

[Brin75]

Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, SE-1,2, pp. 199-206.

[Brin78]

Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," CACM 21, 11 (Nov. 1978) pp. 934-941.

[Cour71]

Courtous, P.J., F. Heymans, D.L. Parnes, "Concurrent Control with Readers and Writers," CACM 14, 10 (October 1971) pp. 667-668.

[Deme78]

Demers, A. and J. Donahue, "Report on the Programming Language Russell," in preparation.

[Dijk76]

Dijkstra, E.W., A Discipline of Programming, Prentice Hall Inc., Englewood Cliffs, N.J., 1973.

[Hoar74]

Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM 17, 10 (October 1974) pp. 549-557.

[Hoar78]

Hoare, C.A.R., "Communicating Sequential Processes," CACM 21, 8 (August 1978) pp. 666-677.

[Kess77]

Kessels, J.L.W., "An Alternative to Event Queries for Synchronization in Monitors," CACM 10, 7 (July 1977) pp. 500-503.

[Kieb78]

Kieburtz, R.B., "Comments on Communicating Sequential Processes," submitted to CACM.

[Lisk74]

Liskov, B.H. and S.N. Zilles, "Programming with Abstract Data Types," SIGPLAN Notices 9, 4.

[Lohr77]

Lohr, K.P., "Beyond Concurrent Pascal," Operating Systems Review 11, 5 pp. 173-180.

[Schn78]

Schneider, F.B., "Structure of Concurrent Programs Exhibiting Reproducible Behavior," Ph.D. Thesis, S.U.N.Y. at Stony Brook, August 1978.

[Wirt77]

Wirth, N., "Modula: a Language for Modular Multiprogramming," Software-Practice and Experience, Vol. 7, pp. 3-35.