MEDIUM TERM SCHEDULING AND EQUIVALENCE OF SYNCHRONOUS AND ASYNCHRONOUS OPERATION

BY

E.A. AKKOYUNLU, A.J. BERNSTEIN, F.B. SCHNEIDER

State University of New York at Stony Brook
Department of Computer Science
Stony Brook, New York 11794

TECHNICAL REPORT # 72
JUNE 1977
MEDIUM TERM SCHEDULING AND EQUIVALENCE OF SYNCHRONOUS
AND ASYNCHRONOUS OPERATION*

BY

E.A. AKKOYUNLU
A.J. BERNSTEIN
F.B. SCHNEIDER

ABSTRACT

In a previous report [A77], the usefulness of the equivalence of
synchronous and asynchronous operation of a software system was discussed,
and restrictions to guarantee this equivalence were presented. Some of
those restrictions are more severe than necessary to preserve this equi-
valence property. A useful relaxation of one of those restrictions,
that allows flexible medium term scheduling of resources, is considered
in this paper.

* This research was supported in part by NSF grant MCS 76-04828.
INTRODUCTION

As software systems become increasingly complex it becomes difficult to insure correctness. Progress has been made in both the verification of systems by formal means ([F67],[H69]), and in the development of new methodologies and programming languages for writing these systems ([D68],[D92],[L74],[P74], to name a few). Systems that have parallel or concurrent activity are particularly hard to certify. This is because the behavior of the system in the concurrent mode of execution may be different than that obtained when each component process is run alone. In the concurrent mode of execution one asserts that each process will have a chance to execute (sometimes referred to as "finite progress"), but no specification is made as to the way concurrent process execution is interleaved or relative execution speeds. Various synchronization primitives can be used by the programmer to constrain the interaction of multiple processes so as to insure the integrity of shared data. (For example, Dijkstra's P and V, Brinch Hansen's Conditional Critical Regions, Monitors). Nevertheless, it is possible for a process to see the system in a state that reflects partial execution of other processes. Such a state may not have been anticipated by the system designers. In addition, because of the nature of parallel operation, it is likely that this state can not be easily reproduced during the debugging procedure.

In Akkoyunlu et al [A77] a set of restrictions on the structure of a system were presented that guarantee that the behavior displayed in the concurrent or parallel mode of operation would also be displayed in the synchronous or non-parallel operation of that system. This is called the
equivalence of synchronous and asynchronous operation. This equivalence property is useful in software systems as the designer need only be concerned that the system functions correctly in the synchronous mode since he is assured that this also completely describes asynchronous behavior. The result in [A77] is described in terms of the Concurrent Pascal [B75] programming language. This language is based on the monitor concept, [H74,B75] and has been used to write several small operating systems [B76], hence its utility is demonstrated. Since the equivalence result places restrictions on the language, it is only valuable if real systems can still be written. One restriction, that the wait statement be the first statement of a monitor, is not realistic in that respect. In this paper a language feature is proposed that permits a useful relaxation of this restriction.

Brinch Hansen [B75] introduces the delay and continue statements to facilitate what he refers to as medium term scheduling of resources. That is, these primitives permit the system designer to alter the order that execution occurs in a monitor. When a process enters a monitor, a computation is made based on the values of the parameters passed to the monitor (thus, the "nature" of the request for the resource), and the values of the permanent variables in the monitor (which is the "status" of the resource) which yields sufficient information to order the process on a queue of processes awaiting access to the resource. For example, in an implementation of the elevator algorithm for disk head scheduling ([H74],[K68]), the computation of priority involves a specification of the track to be referenced which is passed as a parameter to the monitor, as well as the current location of the disk head, and the current direction
of head motion which are both permanent monitor variables.

In [B75], Brinch Hansen illustrates medium term scheduling by defining an array (vector) of queues, and computing an index which indicates which particular queue a process should be suspended on. When a process is to be re-activated, the queue with the lowest index that has a process suspended on it is signaled ("continue-ed"). In Concurrent Pascal at most one process may be suspended on a queue at a time. Hoare [H74] provides an analogous construct in his monitor definition. The equivalent of a queue is called a condition, which differs from the Brinch Hansen [B75] notion in that more than one process may be suspended on a given condition at a time. To order the processes waiting on a condition the priority wait (wait(p)) statement is provided. This causes the suspension of the executing process. The "queue" associated with the condition is ordered such that the suspended process with the lowest priority p, is always the first to be awakened. Using this construct a priority is computed rather than an index into a vector as was the case previously.

The Generalized Condition

It would seem that in order to do medium term scheduling it must be possible to do some computation that uses both permanent monitor variables and parameters prior to the execution of the wait (or delay) statement. In Akkoyunlu et al [A77] it is required that the wait statement be the first executable statement in a monitor, in order to preserve the desired equivalence between synchronous and asynchronous operation. This seems to indicate that it is not possible to do medium term scheduling in a
monitor and still retain the equivalence property. Fortunately, this is not the case. A careful analysis of the proofs in Akkoyunlu et al [A77] shows that the order that suspended processes are awakened (which is just the medium term scheduling policy) is not specified, and in fact is irrelevant to the demonstration of the equivalence result. What is required is that a process does not save for future reference, or alter, the permanent variables of the monitor prior to executing the wait statement. The following language construct, the generalized condition, provides a means for the specification of medium term scheduling policies without sacrificing the desired synchronous - asynchronous equivalence property. It is still required that a wait statement be the first statement of a monitor procedure. The generalized condition however provides the means to do the computation of a priority which can be used to order the suspended processes on a particular condition.

The generalized condition may be thought of as a generalization of the Hoare priority wait scheme [H74]. A condition is declared as follows:

```
<cond name> : condition
  assertion {<bool exp>};
  priority (arg1 : t1, arg2 : t2, ..., argn : tn);
  .... declare local variables ....
  begin
      .... code to compute a "priority"
      value. This value is assigned to priority.
  end;
end ;
```
The syntax and semantics of the `wait` statement are as follow:

**Syntax of `wait` statement:**

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

**Semantics:**

Upon execution of this statement, the boolean expression `<bool exp>`, bound to `<cond name>` is evaluated. If the value is true execution continues; if false the process is suspended. (There is good reason to require that all `wait` statements have a boolean expression bound to them explicitly as well as that the `condition` be declared global to the monitor [K76]). If the process is suspended, an entry is added to a queue associated with the `condition`. Each entry on the queue represents a process suspended on that particular `condition`. In addition to the process name (which must be saved in the Brinch Hansen and Hoare schemes), the `values` of `arg1`, `arg2`, ..., `argn` are stored in the entry. These arguments may be parameters or permanent monitor variables.

The `priority` function of the `generalized condition` is a function procedure which computes an integer value from the values stored in a queue entry of a suspended process (the values of `arg1`, `arg2`, ..., `argn` at the time the process was suspended), and the current values of the permanent monitor variables. We require that evaluation of a `priority` does not alter the values of any permanent monitor variables. One way to insure this is to require that these variables never appear as the target of an assignment statement in the `priority` function, and that the function never calls other procedures. In addition, no flexibility
is lost by requiring that only one priority function be associated with each condition.

To reactivate a suspended process various primitives have been proposed: signal [H74], continue [B75], and the automatic signal [K76]. Whichever of these constructs is used a process will only be reactivated if the boolean expression, <bool exp>, upon which it was suspended would now evaluate to true. (The utility of this is discussed elsewhere [K76], [K77],[S77]). The choice of which process to reactivate is determined as if the priority function were evaluated on each queue entry associated with the condition that is now true, and the process that computes the lowest priority is reactivated. We note here that:

(1) The condition must be declared global to the monitor. This guarantees that if it is true for any process that is suspended, it will be true for all processes suspended on that condition.

(2) It is generally not necessary to evaluate the priority function on each element of the queue to ascertain which 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 those values stored in the queue elements, then it may be computed once, at the time the process is suspended, and the queue maintained in ascending order by this value. All of the medium term scheduling algorithms we have studied thus far fall into this category (as well as the examples in this paper).
(3) The use of medium term scheduling will invalidate any assumptions about finite progress usually made about 'wait' statements.

Examples

We now illustrate the use of generalized conditions by means of two examples. Notice that the 'wait' statement is the first statement of the monitor procedure, thus equivalence of synchronous and asynchronous system behavior will be achieved. We assume the automatic signal facility of Kessels [K76]. In addition we assume the existence of a system function, GET-TIME, which returns the time at call, perhaps in milliseconds.

Figure 1 illustrates a first come first served discipline. Both shortest job first, and last in first out disciplines can be constructed in much the same manner. Figure 2 is the elevator disk head scheduling algorithm mentioned earlier [K68],[H74]. Notice that neither of the examples reference permanent monitor variables in 'priority' functions (although their values at the time of suspension are used), hence a very efficient implementation is possible.

CONCLUSION

A language construct to augment Concurrent Pascal was proposed which permits medium term scheduling while preserving synchronous - asynchronous equivalence.

ACKNOWLEDGMENT

The authors would like to thank Bob Ensor for comments on an early version of this paper.
FIGURE 1. FIRST COME-FIRST SERVED

```plaintext
type resource = monitor

var inuse : boolean ;

available : condition

assertion {not inuse} ;

priority (time : integer) ;

begin

  priority := time

end ;

end ;

procedure entry acquire ;

begin

  available.wait(GET-TIME) ;

  inuse := true ;

  ............

  ..... use resource ...

  ..........

  inuse := false

end ;

begin

  inuse := false

end
```
FIGURE 2. DISK SCHEDULING

```plaintext
type resource = monitor

var inuse : boolean ;
incr, lastincr : boolean ;
fudge_prio : integer ;
curaddr, lastaddr : taddress ;
available : condition

assertion {not inuse} ;
priority (trkno, curtrkno : taddress) ;
begin
  if incr and (trkno ≥ curtrkno)
    then priority := trkno + fudge_prio
  else if incr
    then priority := disksize - trkno
       + disksize + fudge_prio
  else if not incr and trkno ≤ curtrkno
    then priority := disksize
       - trkno + fudge_prio
  else if not incr
    then priority := trkno + disksize + fudge_prio
end;
end;

procedure entry use (trkadder : taddress) ;
begin
  available.wait(trkadder, curaddr) ;
inuse := true ;
seek(trkadder) ;
```

CONTINUED
lastaddr := curaddr 
curaddr := trkaddr 
lastincr := incr 

if curaddr > lastaddr then incr := true 
else incr := false 

if not (lastincr = incr) then fudge_prio :=
    fudge_prio + disksize 

begin
    inuse := false 
end 

begin 
    inuse := false 
    incr := true 
    lastincr := incr 
    curaddr := 0 
    lastaddr := curaddr 
    fudge_prio := 0 
end 

Note: Assume "disksize" is a constant that is equal to the number of tracks on a disk. Also assume that "seek" is a procedure to cause a disk head seek operation.
REFERENCES


