SENTINELS:

A CONCEPT FOR MULTIPROCESS COORDINATION

by

Robert M. Keller

UUCS - 78 - 104

June 1978

This work was supported by the National Science Foundation through grant MCS 77-09369.

SENTINELS:

A CONCEPT FOR MULTIPROCESS COORDINATION

Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

June 1978

Abstract

The *sentinel* construct is introduced, which provides a certain syntactic and semantic framework for multiprocess coordination. The advantage of this construct over others is argued to be semantic transparency, efficiency, ease in implementation, and usefulness in verfication.

Key words and phrases: parallelism, concurrency, synchronization, mutual exclusion, monitors, semaphores, correctness proofs, semantics.

CR categories: 4.31, 4.32, 4.35, 5.24, 6.2

Introduction

The area of process coordination in operating systems has seen a wide variety of constructs. A partial list includes events and queues [Witt 66], semaphores [Dijkstra 68], supervisory computers [Gaines 72], conditional critical regions [Brinch Hansen 73], monitors [Hoare 74], path expressions [Campbell and Habermann 74], serializers [Atkinson and Hewitt 77], atomic actions [Lomet 77], and undoubtedly the list will continue to grow. This paper introduces another entry to the list, which we argue has most of the good features of its predecessors, few of the bad features, and some advantages of its own. No claim is made that this construct does not overlap ideas with others in the list.

The sentinel construct uses a *queuing primitive* as a basic form of synchronization. More elaborate forms of synchronization are then built up by constructing a sequential process (a *sentinel*) which coordinates other processes via the basic queuing primitive.

Instead of being a passive *object*, wherein processes being coordinated are expected to carry out certain clerical operations (e.g. causing other processes to be scheduled), a sentinel is an *active process* and carries out such operations itself. This is not to say, however, that a sentinel will have no periods of inactivity. Indeed, it can be made active just when the appropriate conditions hold, thereby avoiding busy-waiting.

Finally, rather than just exchanging *data* with processes being coordinated, a sentinel can be put in control of the execution of statements of such processes. This has certain advantages in "structured" concurrent programming. For example, it can eliminate the need for the programmer to specify instructions for both entry and exit of a critical section. Using

an appropriate sentinel, he need only specify that a certain block (the critical section) be controlled by the sentinel.

Many variations on these themes are possible. For example, the "queues" could be restricted in length for implementation convenience. What we sketch in this paper is, therefore, just one possible development of the concept.

Undoubtedly, the idea of an active synchronizing primitive has occurred to others. It first occurred to the author while working on hardware modules [Keller 68], but this idea did not get written attention until [Keller 74]. A software version for achieving mutual exclusion appeared in [Holt 71]. Why no one has sought to develop it further is a mystery. Perhaps the overhead of using an additional process for synchronization is viewed as being too great. However, since such a process can be dormant (or "sleeping") most of the time, a carefully optimized version should be no less efficient than the other elaborate synchronization schemes. Furthermore, there is some precedent for being generous with the number of processes. For example, [Hoare 73] suggests using a process for each page in memory.

Process Creation

Prior to introducing the sentinel notion itself, we must discuss the specification of processes, since a sentinel is just a special type of process.

In order to have a means for creating processes, we assume the underlying mechanics for a detached mode of execution, e.g. as with the "task" option in PL/I [IBM 68]. For concreteness, we assume that any syntactic statement entity, <statement>, can be executed as a process by the statement

detach(<process reference>) <statement>
(<process reference>: optional)

which will create a process for the statement which then runs concurrently with the creating process. <reference> is a variable of type *process reference* and is assigned a reference to the created process.

Whether or not the process has completed can be determined by evaluation of the Boolean

completed(<process reference>)

The process is complete when the corresponding statement is completely executed. For example, if <statement> is a block, completion is when control leaves the block. In some cases, processes will be created which will be deliberately non-terminating (but which could be aborted if the job creating them terminates).

We assume a *wait until* statement, which will delay a process until a specified condition becomes true. To avoid busy waiting, the condition will be evaluated when the statement is first encountered and, if the result is *false*, again whenever an event occurs which could change the result to *true*. Restrictions on the form of the condition would likely

be imposed to improve the efficiency of this evaluation, but this is not our primary concern here. Most typically, we would expect to find

wait until completed(<process reference>)

where the reference is to some earlier-created process.

An additional related option provides additional convenience. This is the "count" option. We let

detach <statement> count(c)

mean that the designated integer variable c will be incremented by 1 when this statement is executed, and decremented by 1 when and if the detached process terminates. When using this option, we would expect to find statements of the form

wait until $c = 0$

We do not wish to be distracted here with issues such as "completeness" with or without the count option. Such discussions are best saved for future investigation.

Sentinels

A *sentinel* is a special kind of process set up to provide a tailored communication discipline between other processes. It does so by being the *unique server* of a set of *queues* which are associated with it. The use of queues for communicating *data* between processes is well understood. Sentinels add a unique feature of allowing a *statement* to be placed on the queue, in the sense that the sentinel can determine when that statement is to be executed, thus executing synchronization control over the *enqueuing* process.

In order to set up the queues, an additional option, the *queue list* option, is specified in the *detach* statement. The latter would then take the form

detach(<process reference>) <queue list> <statement>
(<process reference>: optional)

It is the queue list option which indicates that a process is a sentinel. More precisely, <queue list> is of the form

queues(<identifier list>)

where the identifiers are of type *queue reference*. This means that when the process is created, a queue is established for each entry in the list, and the reference identifiers are set so as to reference these queues. The process created is the *server* of those queues.

It is expected that the statement to be executed by the process has a *declaration* of its queues. These queues are referenced through reference variables local to the process. Thus the declaration would appear as

queues(<queue entry list>)

where each entry in the list is of the form

<queue reference> (<identifier list>)
optional

which resembles a procedure header. The <queue entry list> is expected to correspond with the <queue list> specified when the process is created.

The items which are communicated via queues are called *tokens*. A token is a pair, comprised of a *statement* and a *parameter list*. Either of these items may be *null* in various applications. The identifiers in each <identifier list> correspond with parameters in the <parameter list> part of a token.

A token gets created by a process, called the *enqueuing process*, through a statement of the form

queue(<queue reference> <parameter list>) <statement>
optional optional

The execution of this statement specifies that a token with the components

<statement>, <parameter list>

should be placed on the referenced queue. The placement of such a token puts the execution of <statement> in control of a unique process *serving* the queue. It also makes any parameters in <parameter list> accessible by this *server*.

If it is "data" which is to be communicated from one process to another, chances are that the statement part of the token would be *null* and the data would be either contained in, or referenced through, the parameter list. On the other hand, we shall see instances where the data, and hence the parameter list, is null, but the statement part is important.

By convention, omission of <statement> implies the *null statement*.

The server decides that <statement> is to be allowed to execute by itself executing

execute <queue reference> [n]

where n is an integer variable whose value indicates the position from the *head* of the queue, i.e. the end containing the statement having been on the queue the longest. Once the *execute* statement is executed, the statement on the queue at that position cannot be stopped (at least not at the level of the language we are describing). It is removed from the queue and cannot be re-executed.

The number n above always refers to the position among the remaining entries. Thus always using

execute <queue reference> [1]

provides a FIFO discipline. Similarly, if we introduce

last(<queue reference>)

which evaluates to the position of the last remaining entry,

execute <queue reference> [last(<queue reference>)]

provides a LIFO discipline when used universally.

It is quite possible that one is interested only in FIFO disciplines, in which case [n] could, of course, be omitted from the language.

We adopt the convention that the expression

last(<queue reference>) = 0

is true exactly when the corresponding queue is empty. We use the abbreviation

empty(<queue reference>)

for this expression, and

non-empty(<queue reference>)

for its negation.

We allow the detached mode of execution for an *execute* statement, viz.

detach(<process reference>) execute <queue reference> [*n*]

Since the token is already a statement in another process, namely the enqueuing one, execution can be optimized so that no new process is actually created.

In order for a server to reference the parameter list of a token, we use the form

<queue parameter> [*n*]

to refer to the named parameter of the *n*-th entry.

Interim Summary

Before proceeding to examples, we briefly summarize the concepts put forth in the preceding sections. First, we gave a way of representing process creation. Then we introduced the concept of a sentinel process, which may be created with a number of queues and which becomes the server of those queues. Other processes (called enqueuing processes) interact with the sentinel by specifying a queue, possibly with some parameters, and a statement. The statement and parameters comprise a token which is placed on the queue. This allows the server of that queue to interact with the enqueuing process through the parameters, and to control execution of the statement. The enqueuing process does not proceed until the statement is completely executed.

Examples

We now attempt to clarify the preceding informal definitions by programming a number of standard examples.

Example Semaphores [Dijkstra 68]: A minimum acceptability requirement for a synchronizing construct is that it be able to implement a semaphore. The semaphore implementation shown below uses a sentinel with two queues. The usual P and V operations are represented as calls on a null statement with one of these queues specified. The private local storage used in the sentinel corresponds to the usual "semaphore data structure". We thus have the following correspondences:

(next page)

To set up the semaphore, execute:

detach queues(*P*, *V*) call *semaphore*(<initial value>)

To do the P operation on the semaphore, execute

queue(*P*)

Similarly, to do the V operation, execute

queue(*V*)

A second semaphore might be set up by

detach queues(*P1*, *V1*) call *semaphore*(<initial value>)

and the corresponding statements would be queue(*P1*) and queue(*V1*).

The code for a semaphore sentinel is as follows:

```
procedure semaphore(natural number initial-sem-val) queues(P, V);
     integer sem-val;

     sem-val := initial-sem-val;

     loop
          wait until non-empty(P) v non-empty(V);
          if non-empty(P)
               then
                    if sem-val = 0
                         then
                              wait until non-empty(V);
                              detach execute V[1]
                         else
                              sem-val := sem-val - 1
                              fi;
                    detach execute P[1]
               else
                    detach execute V[1];
                    sem-val := sem-val + 1
               fi
          pool
     end semaphore;
```

Example Mutually-exclusive execution of procedures. To cause a set of procedures to be executed mutually-exclusively, call each according to the following:

queue(*M*) <procedure call>

where *M* is the queue of a sentinel formerly created by

detach queues(*M*) call *mutex*

The code for the *mutex* sentinel is as follows:

```
procedure mutex queues(port);
    loop
        wait until non-empty(port);
        execute port[1]
        pool
    end mutex;
```

The fact that the *execute* is *not* detached is what provides mutual exclusion.

In the semaphore and mutual exclusion examples, no information was passed to the sentinel in the form of queue parameters. The following example is the first we shall see in which this feature is used.

Example Message buffer.

To create buffer of size *n*:

detach queues(*inq*, *outq*) call *message-buffer*(*n*)

To enter message *mes*, execute:

queue(*inq*(*mes*))

To remove message *mes*, execute:

queue(*outq*(*mes*))

```
procedure message-buffer(natural number n)
     queues(inq(message inmes), outq(message outmes));
          array buffer[0..n-1] of message;
          integer in, out, count;
          in := out := 0;
          count := 0;
          loop
               wait until non-empty(inq) v non-empty(outq);
               if non-empty(inq)
                    then
                         if count < n
                              then
                                   buffer[in] := inmes[1];
                                   detach execute inq[1];
                                   in := (in + 1) mod n;
                                   count := count + 1
                              else
                                   wait until non-empty(outq)
                              fi
                    fi;
               if non-empty(outq)
                    then
                         if count > 0
                              then
                                   outmes[1] := buffer[out];
                                   detach execute outq[1];
                                   out := (out + 1) mod n;
                                   count := count - 1
                              else
                                   wait until non-empty(inq)
                              fi
                    fi
               pool
          end message buffer;
```

Example FIFO readers/writers

To create sentinel:

detach queues(*RW*) call *readers-writers-1*

There will be a single queue, and the parameter value 0 or 1 will indicate *reading* or *writing* respectively.

To "read" via a statement, use

queue(*RW*(0))

To "write" via a statement, use

queue(*RW*(1))

```
procedure readers-writers-1 queues(entry(integer type));
     integer counter;
     counter := 0;
     loop
          wait until non-empty(entry);
          if type[1] = 0
               then
                    detach execute entry[1] count(counter)
               else
                    wait until counter = 0;
                    execute entry[1]
               fi
          pool
     end readers-writers-1;
```

Example Readers/writers with various types of priority. For each of the following versions of readers/writers, two queues are used;

To create sentinel:

detach queues(*read, write*) call *readers-writers-n*

where *n* is a version number (2 or 3).

To "read" with procedure, use

queue(*read*)

To "write" with procedure, use

queue(*write*)

The code for various versions follows:

```
procedure reader-writers-2 queues(read, write);
     comment:  writers have priority;
     integer readers;
     readers := 0;
     loop
          wait until non-empty(read) v non-empty(write);
          if non-empty(write)
               then
                    wait until readers = 0;
                    execute write[1]
               else
                    detach execute read[1] count(readers)
               fi
          pool
     end reader-writers-2;
```

```
procedure reader-writers-3 queue(read, write);
     comment:  all readers and writers have a fair chance;
     integer readers;
     readers := 0;
     loop
          wait until non-empty(read) v non-empty(write);
          if non-empty(write)
               then
                    wait until readers = 0;
                    execute write[1]
               fi;
          if non-empty(read)
               then
                    detach execute read[1] count(readers)
               fi
          pool
     end reader-writers-3;
```

We now summarize the advantages of the sentinel concept:

1. Sentinels are just processes, so their understanding does not entail any substantially new concept. Only the enqueued statements which accompany processes served by sentinels require any extension of standard semantics.

2. Sentinels are easy to understand. The code for a sentinel is usually sequentially executed. Waiting occurs at well-defined points, with clear semantics.

3. Sentinels have no "hidden" or unspecified scheduling discipline. It is usually obvious from inspection what queue is served next.

4. Sentinels can be constructed without the single queue "bottleneck". Multiple queues are used for this purpose. This eliminates some of the anomalies cited in [Lipton 73].

5. Multiple queues available with sentinels allow the communication of information by queue selection and also the sorting of processes into classes when order of arrival is irrelevant.

6. Sentinels provide a way of avoiding the possibility of "unmatched brackets" in synchronizing operations (cf. [Greif 75]). For example, it is unnecessary to have separate entries for start-write and end-write.

7. Sentinels with simple waits can be implemented efficiently through compiler optimization, yet do not prohibit their user from constructing more-exotic but perhaps less-efficient waits.

8. The sequential combination of waits, such as allowed in sentinels, is often more efficient and easier to design than a single combinatorial condition.

9. For sentinels with sequential programs and simple waits, correctness can often be proved using only sequential program proof techniques.

10. Dynamically created sentinels offer no unique implementation problems.

11. Sentinels provide customized selection of processes from queues (rather than a fixed discipline) and for priority execution as desired by the programmer.

12. A library of "standard" sentinels is easily maintained.

13. The issues of resource protection and synchronization mechanism can be separated through use of sentinels.

With regard to property number 13, it should be a simple matter for a protection mechanism to force access to certain objects through certain procedures (cf. [Wulf, et al. 74]). Thus, the mechanism could easily be extended to force the use of a certain tag, which causes coordination by a sentinel.

Comparison with Other Synchronizing Constructs

Although sentinels have ideas in common with many proposals, it appears that they have the most in common with serializers [Atkinson and Hewitt 77]. Although it may be due to our lack of intimate familiarity with the actor model on which serializers are based (sentinels are based on sequential programs), it appears that serializers lack at least properties 1, 2, and 11 above. Unlike serializers, processes do not "possess" control of sentinels. The latter are independent processes in their own right. Also, sentinels need not explicitly *relay* messages to processes using resources, enhancing the ability of the system to enforce protection.

Monitors [Hoare 74] are also closely related, but appear to lack properties 1, 3, 4, 5, 6. Regarding property 3, [Howard 76] describes numerous possible interpretations for the underlying scheduling in monitors. Although Hoare gave a specific interpretation, it appears that it may be less than transparent to the programmer. Although monitors as described in [Hoare 74] are not dynamically creatable, extension to allow this presents no real problem.

Path expressions [Campbell and Habermann 74] bear a certain similarity to sentinels. However, as proposed in the cited reference, they are apt to leave aspects of implementation arbitrary, which sentinels avoid doing. Also, the "completeness" of path expressions seems more subject to question than the completeness of sentinels, the former being based on regular expressions. We conjecture that there is an algorithm for producing from any path expression a sentinel implementation, and that this is true for path expressions which lack many of the restrictions imposed in the reference cited. Also, unlike monitors and path expressions,

sentinels need not *encase* their resources. Hence the same sentinel procedure can be used for any number of different resources of different types.

Conditional critical regions [Brinch Hansen 73] appear to lack properties 1, 3, 5, 8, 9, and 11. Conditional critical regions are apt to be rather opaque to the programmer without knowledge of the underlying scheduling disciplines, particularly when the change of a variable causes more than one awaited condition to become true simultaneously.

Atomic actions [Lomet 77] form another type of coordination construct. Like the others being compared here, they also have the property that the processes being coordinated are responsible for carrying out the actions inside the primitive. Consequently, the method of dealing with conflicting calls to the primitive is opaque to the programmer. This is in constrast to the sentinel's use of explicit polling of requests to make the treatment of conflicting calls transparent.

The supervisory computer concept [Gaines 72] does use the notion of an active process which coordinates other processes. It does not give a language construct *per se*, nor does it put the control of single statements under control of the synchronizing primitive. The programs are written on a lower level than sentinels and no built-in queuing is provided.

Proofs

At this stage, we have not had much experience in proving properties of sentinels. It is clear, however, that sequential program proof techniques can be used for proving invariants to a large extent, thanks to the sequential nature of most sentinels. Although there are concurrent transitions to be considered, namely processes joining queues and detached processes completing, these can be kept under control by careful programming.

It is too early to attempt a set of formal proof rules. We can make some observations however. For any queue q if P is a predicate not referring to q, the following is a valid inference rule:

$$\left\{\begin{array}{l} \text{now } P \\ \text{wait until non-empty}(q) \\ \text{now } P \wedge \text{non-empty}(q) \end{array}\right\}$$

Here *now* indicates an invariant assertion for that point in the program. This rule is valid because a process cannot leave the queue once it joins it, other than through an *execute* instruction in the sentinel.

On the other hand, the following would *not* be valid:

$$\left\{\begin{array}{l} \text{now } P \\ \text{wait until empty}(q) \\ \text{now } P \wedge \text{empty}(q) \end{array}\right\}$$

because a process may join the queue after the *wait* is satisfied, without any action on the part of the sentinel. We can summarize this distinction by saying that *non-empty* is a *monotone* predicate whereas *empty* is not, where monotonicity of a predicate within a sentinel means its invariance within a sentinel, relative to the behavior of enqueuing processes.

Similarly, if we use a variable *count* to count the number of detached processes in a certain category, then for any N not occurring in P, we have a rule

$$\left\{\begin{array}{l}\text{now } P\\ \text{wait until } count \leq N\\ \text{now } P \wedge count \leq N\end{array}\right\}$$

Based on such considerations, we have annotated the *semaphore* sentinel program with invariant assertions, as shown on the following page.

```
comment:  body of procedure semaphore annotated with assertions;

comment:  now indicates an invariant at that execution point in the program;

comment:  henceforth indicate an invariant for each execution point to follow;

comment:  P_ex and V_ex are the sequences of P and V tokens executed, respectively;

comment:  A(x) abbreviates |P_ex| + x = |V_ex| + initial-sem-val;

henceforth initial-sem-val > 0;
sem-val := initial-sem-val;
henceforth sem-val ≥ 0;
loop
    now A(sem-val);
    wait until non-empty(P) ∨ non-empty(V);
    now(non-empty(P) ∨ non-empty(V)) ∧ A(sem-val);
    if non-empty(P)
        then
            now non-empty(P) ∧ A(sem-val);
            if sem-val = 0
                then
                    now non-empty(P) ∧ A(sem-val) ∧ sem-val = 0;
                    wait until non-empty(V);
                    now non-empty(V) ∧ non-empty(P) ∧ A(sem-val);
                    detach execute V[1];
                    now non-empty(P) ∧ A(sem-val + 1);
                else
                    now non-empty(P) ∧ A(sem-val);
                    sem-val := sem-val - 1;
                    now non-empty(P) ∧ A(sem-val + 1);
                fi;
            now non-empty(P) ∧ A(sem-val + 1);
            detach execute P[1];
            now A(sem-val);
        else
            now non-empty(V) ∧ A(sem-val);
            detach execute V[1];
            now A(sem-val + 1);
            sem-val := sem-val + 1;
            now A(sem-val);
        fi
    now A(sem-val);
    pool
```

In general, we would like a high-level scheme for stating correctness of sentinels (i.e. a *denotational semantics*), and a method for proof of such correctness. The internal invariants are likely to form only one part of such a proof. The kind of scheme we seek has not yet been developed. However, we give an example to show what form such a scheme might take, with an accompanying informal proof.

Claim letting P_{ex} and V_{ex} denote the sequence of statements from queues P and V, respectively, which are executed, and P_{in} and V_{in} denote the sequence of statements which enter the queues, we have correct operation of the semaphore sentinel, as defined by the equations

$$V_{ex} = V_{in}$$

$$P_{ex} = \langle P_{in} \rangle_{|V_{in}| + initial\text{-}sem\text{-}val}$$

The notation is that $|X|$ represents the length of sequence X and $\langle X \rangle_n$ denotes the first n components of X, or all of X if there are fewer than n components.

These equations give a *denotational* semantics for the long-range behavior of the semaphore, in the spirit of the equations in [Keller 78]. They hold whether the sequences P_{in} and V_{in} are finite or infinite. The reader will also note a similarity to the "semaphore invariant" in [Habermann 72].

Proof Since all *executes* are done on the first queue element, we immediately have the inequalities (where $\leq$ denotes *is a prefix of*)

$$V_{ex} \leq V_{in}$$

$$P_{ex} \leq P_{in}$$

We are thus left with showing

$$|V_{ex}| = |V_{in}|$$

$$|P_{ex}| = \min(|P_{in}|, |V_{in}| + initial\text{-}sem\text{-}val)$$

To prove the first equality, suppose to the contrary that $|V_{ex}| \neq |V_{in}|$. Since $V_{ex} \leq V_{in}$, we have that $|V_{ex}| < |V_{in}|$. Notice that each iteration of the loop must execute a P or a V. From the invariant assertions in the

annotated version of the procedure, we see that at any time only finitely-many P's can be executed before either a V must be executed or waiting occurs. At this point, if $|V_{ex}| < |V_{in}|$, then another V can be executed. Hence the long-range behavior cannot have $|V_{ex}| < |V_{in}|$.

For proof of the second inequality, we examine two cases:

(i) $|P_{in}| \leq |V_{in}| + \textit{initial-sem-val}$.

(ii) $|P_{in}| > |V_{in}| + \textit{initial-sem-val}$.

In case (i), it suffices to show that the following gives a contradiction:

(iii) $|P_{ex}| < |P_{in}|$

Since every loop iteration executes a P or V and the P queue is polled first on each iteration, this implies that the sentinel must stop, waiting at the statement

wait until non-empty(V)

The invariant which precedes this statement gives us

$$A(0): \quad |P_{ex}| = |V_{ex}| + \textit{initial-sem-val}$$

But we already proved that $V_{ex} = V_{in}$, so

$$|P_{ex}| = |V_{in}| + \textit{initial-sem-val}$$

and combining this with (i), we get

$$|P_{in}| \leq |P_{ex}|$$

which does indeed contradict (iii).

Similarly, in case (ii), it suffices to show that the following gives a contradiction:

(iv) $|P_{ex}| \neq |V_{in}| + \textit{initial-sem-val}$

We infer from the assertions in the program that $A(sem\text{-}val) \vee A(sem\text{-}val + 1)$ is invariant, and since $sem\text{-}val \geq 0$ is also invariant and $V_{ex} = V_{in}$ has been proved, we have

$$|P_{ex}| \leq |V_{in}| + initial\text{-}sem\text{-}val$$

With (iv), this gives

(v) $$|P_{ex}| < |V_{in}| + initial\text{-}sem\text{-}val$$

So from (ii), and (v), we have

$$|P_{ex}| < |P_{in}|$$

Once again, this implies that the sentinel stops at

wait until non-empty(V)

where the invariant $A(sem\text{-}val) \wedge sem\text{-}val = 0$ gives

$$|P_{ex}| = |V_{in}| + initial\text{-}sem\text{-}val$$

which contradicts (v), as desired.

Conclusions and Future Research

We have introduced the sentinel construct as a means of achieving tailored communication disciplines between processes. As pointed out, this construct has features in common with other proposals for synchronizing constructs. We feel that the sentinel retains the most desirable features of each of these. It also adds new elements. In particular, it allows the programmer to specify scheduling which cannot be specified in some other schemes, without imposing undue complications. It separates scheduling actions from the processes being scheduled, in contrast to other approaches in which the synchronizing construct is passive, wherein the processes being synchronized are required to do any necessary bookkeeping. Finally, it adds the feature of having statements be a component of enqueued token, which we feel is useful in "separation of powers" when protection is at issue.

We have left unexplored many variations, e.g. restricting queue lengths (say, to 1). Although an example of a correctness proof was presented, much remains to be explored in this area, both formal and informal.

Acknowledgement

The comments of Professors John Smith and Gary Lindstrom and the typing of Karen Evans are greatly appreciated.

References

[Atkinson and Hewitt 77] R. Atkinson and C. Hewitt. Synchronization in actor systems. Proc. 4th ACM Conference on Principles of Programming Languages, 267-280 (Jan. 1977).

[Brinch Hansen 73] P. Brinch Hansen. Operating system principles. Prentice-Hall (1973).

[Campbell and Habermann 74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In Gelenbe and Kaiser (eds.), Operating Systems, Springer Lecture Notes in Computer Science, 16, 89-102 (1974).

[Dijkstra 68] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys (ed.), Programming Languages, Academic Press (1968).

[Gaines 72] R. S. Gaines. An operating system based on the concept of a supervisory computer. CACM 15, 3, 150-156 (March 1972).

[Greif 75] I. Greif. Semantics of communicating parallel processes. MIT Project MAC TR-154 (Sept. 1975).

[Habermann 72] A. N. Habermann. Synchronization of communicating processes. Comm. ACM 15, 3, 177-184 (March 1972).

[Hoare 73] C. A. R. Hoare. A structured paging system. Computer J., 16, 3, 209-215 (1973).

[Hoare 74] C. A. R. Hoare. Monitors: an operating system structuring concept. CACM 17, 10, 54-557 (Oct. 1974).

[Holt 71] R. C. Holt. On deadlock in computer systems. Tech. Rep. CSRG-6, Computer Systems Research Group, University of Toronto (April 1971).

[Howard 76] J. H. Howard. Signalling in monitors. Proc. Second International Conference on Software Engineering, 47-52, IEEE 76CH1125-4C (Oct. 1976).

[IBM 68] PL/I reference manual. IBM form C28-8201-1 (March 1968).

[Keller 68] R. M. Keller. Analysis of implementation errors in digital computing systems. Washington University Computer Systems Laboratory TR 6 (MS Thesis), (March 1968). U.S. Government RD Report AD 669-812.

[Keller 74] R. M. Keller. Towards a theory of universal speed-independent modules. IEEE Trans. on Computers, C-23, 1, 21-33 (Jan. 1974).

[Keller 76] R. M. Keller. Formal verification of parallel programs. CACM 19, 7, 371-384 (July 1976).

[Keller 78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In E. J. Neuhold (ed.), Formal description of programming concepts, 337-366, North-Holland (1978).

[Lipton 73] R. J. Lipton. On synchronization primitive systems. Ph.D. Thesis, Carnegie-Mellon University, Department of Computer Science (1973).

[Lomet 77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. Sigplan Notices, 12, 3, 128-137 (March 1977).

[Schmid 76] H. A. Schmid. On the efficient implementation of conditional critical regions and the construction of monitors. Acta Informatica, 6, 227-249 (1976).

[Witt 66] B. I. Witt. The functional structure of OS/360, Part II: Job and task management. IBM Systems J., 5, 1, 12-29 (1966).

[Wulf, et al. 74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. CACM 17, 6, 337-345 (June 1974).