M.I.T. Laboratory for Computer Science
Computer Systems Research Division

March 29, 1977
Request for Comments No. 138

SYNCHRONIZATION WITH EVENTCOUNTS AND SEQUENCERS

David P. Reed
Rajendra K. Kanodia *
M.I.T. Laboratory for Computer Science (formerly Project MAC)
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, Massachusetts 02139

The attached paper is submitted to the Sixth ACM Symposium on Operating Systems Principles.

The work presented in this paper was sponsored in part by Honeywell Information Systems, Inc., and in part by the Air Force Information Systems Technology Applications Office (ISTAO), and by the Advanced Research Projects Agency (ARPA) of the Department of Defense under ARPA order No. 2641 which was monitored by ISTAO under contract No. F19628-74-C-0193.

* Present affiliation:
Bell Laboratories
Holmdel, N.J. 07733

Synchronization with Eventcounts and Sequencers

ABSTRACT

Synchronization of concurrent processes requires controlling the relative ordering of events in the processes. We propose a new synchronization mechanism, using abstract objects called eventcounts and sequencers, that allows processes to control the ordering of events directly, rather than indirectly through mutual exclusion. Direct control of ordering seems to simplify correctness arguments and also simplifies implementation in distributed systems. The mechanism is defined formally, and then several examples of its use are given. The relationship of the mechanism to protection mechanisms in the system is explained; in particular, eventcounts are shown to be applicable to situations where confinement of information matters. An implementation of eventcounts and sequencers in a system with shared memory is described.

Keywords and phrases: process synchronization, interprocess communication, distributed systems, security models, mutual exclusion, semaphores.

CR Categories: 4.30, 4.32.

Introduction

The design of computer systems to be concurrently used by multiple, independent users requires a mechanism that allows programs to synchronize their use of shared resources. Many such mechanisms have been developed and used in practical applications. Most of the currently favored mechanisms, such as semaphores [Dijkstra,68] and monitors [Hoare,74] are based on the concept of _mutual exclusion_.

In this paper, we describe an alternative synchronization mechanism that is not based on the concept of mutual exclusion, but rather on observing the sequencing of significant events in the course of an asynchronous computation. Two kinds of objects are described, an eventcount, which is a communication path for signalling and observing the progress of concurrent computations, and a sequencer, which assigns an order to events occurring in the system.

Eventcounts and sequencers are a more natural mechanism for controlling the sequence of execution of processes that do not need mutual exclusion. Examples of these applications are monitoring state changes of operating system variables, and broadcasting the occurrence of an event to any number of interested processes.

In applications where mutual exclusion mechanisms are explicitly prohibited, such as physically distributed systems and systems that need to solve the confinement problem, eventcounts and sequencers can be used to solve synchronization problems in a very natural way.

Eventcounts

An eventcount is an object that keeps a count of the number of events in a particular class that have occurred so far in the execution of the system. Events in this context are changes in the state of some part of the system, for example, a change in the value of some variable, the arrival of input from an input device, etc. A class of events is a set of events that are related to one another, such as all changes to the variables X and Y, or all inputs from teletype number 377.

An eventcount can be thought of as a non-decreasing integer variable. We define an _advance_ primitive to signal the occurrence of an event associated with a particular eventcount and two primitives, _await_ and _read_, that obtain the "value" of an eventcount.

The primitive _advance_(E) is used to signal the occurrence of an event in the class associated with the eventcount E. The effect of this operation is to increase the integer "value" of E by 1. The "value" of an eventcount equals the number of _advance_ operations that have been performed on it (i.e. the initial value of an eventcount is zero). In the rest of this paper, a process that executes _advance_ operations on an eventcount E will be referred to as a signaller of that eventcount.

A process can observe the value of an eventcount in one of two ways. It can either read the value directly, using the primitive _read_(E), or it can block itself until the eventcount reaches a specific value using the _await_(E,v) primitive.

The _read_(E) primitive returns the "current" value of the eventcount E. As we shall discuss later, the value returned is some value that E had after the _read_ primitive began execution. Thus, although the value may increase during the execution of the _read_ primitive, the result of _read_(E) is a lower bound on the current value of E after the read, and an upper bound on the value of E before the read.

Quite often, a process may not wish to execute until some event in a class that it has interest in has happened. Although that effect can be achieved by looping around an execution of a read(E) primitive until a specified value of the eventcount, E, is reached, it is more useful to provide a primitive that incorporates the waiting. Such a primitive would also provide the opportunity to avoid the busy form of waiting. Consequently, we define a primitive await(E,v), that suspends the calling process until the "value" of the eventcount E is at least v. (1) Of course, as in all mechanisms designed for asynchronous processes, the await primitive may not return immediately once the vth advance on E is executed; the only guarantee is that at least v advances have been performed by the time await(E,v) returns.

Timing Relationships

In a single-processor system with processes communicating through shared memory, the definitions of advance, await, and read given above are adequate. In a distributed system, particularly one that consists of several processors interconnected by communication lines, the definitions are not adequate, because there may be a communications delay between the time an advance operation is finished and the time that another processor can determine that an advance has occurred.

(1) We require that v be a private variable that cannot change while the process is suspended (effectively v is a call-by-value parameter).

Lamport has defined time in a distributed system as a partial ordering of the events in the system [Lamport,76a]. We have modified somewhat his formalization of time as a partial ordering to allow definition of eventcounts in a precise way. Events are the executions of primitive operations of a process, and particularly the executions of advance, await and read primitives constitute events. (1) Time-ordering of events in the system is specified by a partial-order relation on events, denoted by -> (read "precedes"). The relation -> must satisfy the following: (2)

D1) If A and B are events in the same process, and A is executed before B, then A->B.

D2) If A is the transmission of information by one process, and B is the receipt of that information, then A->B. (Examples of transmission of information are sending of messages, storing into a shared variable that is later loaded by another process, etc.)

D3) If W_E is the completion of an await operation of the form await(E,t), then there are at least t members of the set $\{A_E \mid A_E$ is the execution of advance(E) and $A_E \rightarrow W_E\}$.

D4) If R_E is the execution of a read operation of the form v:=read(E), then there are exactly v members of the set $\{A_E \mid A_E$ is the execution of advance(E) and $A_E \rightarrow R_E\}$.

D5) If A->B and B->C, then A->C.

As in Lamport's formalism, it is possible that for two events in the system, A and B, neither A->B nor B->A holds. Such a pair of events will be called concurrent, since they correspond to events that have happened closer together in time that can be resolved within the system, in terms of communication delays. Events that happen at different times (from the point of view of an

(1) If advance, read, or await is implemented as a sequence of primitive steps in a process, the event corresponding to the operation occurs sometime during the execution.

(2) Our formalism differs from Lamport's in that: (a) Conditions D3 and D4 are new in our formalism and (b) Condition D2 is generalized to include information transmission other than by passing messages.

omniscient observer) on two different processors may be concurrent under our definition. For example, if A and B are events on two different physical processors, and A's processor does not detect the occurrence of B until after A has occurred, while B's processor does not detect the occurrence of A until after B has occurred, we will call A and B concurrent.

We require that for any event A, A->A does not hold. This requirement is necessary for physical realizability, for otherwise an event could precede itself.

The definition just given allows for the possibility that <u>advance</u>, <u>await</u>, and <u>read</u> operations on the same eventcount may be concurrent. In a later section, we will show how concurrent operations on the same eventcount can be implemented correctly without using mutual exclusion as an underlying mechanism.

<u>Single-Producer, Single-Consumer Example</u>

As an example of eventcounts used for synchronization, we show how a producer and a consumer process can synchronize their use of a shared N-cell ring buffer. The ring buffer is implemented as an array in shared memory, called <u>buffer</u>[0:N-1]. Two eventcounts, IN and OUT (initially zero), are used to synchronize the producer and consumer. The producer generates a sequence of values by calls on a subroutine "produce," and stores the <u>i</u>th value in buffer[i <u>mod</u> N]. The consumer reads these values out of the buffer in order. The two eventcounts, IN and OUT, coordinate the use of the buffer so that:

1) the consumer doesn't read the <u>i</u>th value from the buffer until it has been stored by the producer, and

2) the producer doesn't store the (i+N)th value into the buffer until the <u>i</u>th value has been read by the consumer.

```
procedure producer()
  begin integer i;
    for i:=1 to infinity do
    begin
        await(OUT,i-N);
        buffer[i mod N] := produce();
        advance(IN);
    end
  end

procedure consumer()
  begin integer i;
    for i:=1 to infinity do
    begin
        await(IN,i);
        consume(buffer[i mod N]);
        advance(OUT);
    end
  end
```

Let the event P_i be the store of the _i_th value by the producer routine, and the event C_i be the reading of the _i_th value. The two synchronization conditions are equivalent to saying that $P_i\text{->}C_i$ and $C_i\text{->}P_{i+N}$.

We can show the synchronization conditions fairly simply. Let $A_{IN,i}$ be the _i_th execution of _advance_(IN) by the producer, $A_{OUT,i}$ be the _i_th execution of _advance_(OUT) by the consumer, $W_{IN,i}$ be the _i_th execution of _await_(IN,i) by the consumer, and $W_{OUT,i}$ be the _i_th execution of _await_(OUT,i-N) by the producer. Because all the _advances_ on IN are done in one process, they are totally ordered, such that if j<k, $A_{IN,j}\text{->}A_{IN,k}$. Similarly, if j<k, $A_{OUT,j}\text{->}A_{OUT,k}$.

We also know that $W_{OUT,i}\text{->}P_i\text{->}A_{IN,i}$, and $W_{IN,i}\text{->}C_i\text{->}A_{OUT,i}$, from the sequencing of the processes.

Now, condition D3 above states that must be at least i _advances_ on IN preceding $W_{IN,i}$, so $A_{IN,i}\text{->}W_{IN,i}$. Similarly, $A_{OUT,i}\text{->}W_{OUT,i+N}$. Putting these relationships together,

$$P_i\text{->}A_{IN,i}\text{->}W_{IN,i}\text{->}C_i\text{->}A_{OUT,i}\text{->}W_{OUT,i+N}\text{->}P_{i+N}$$

or, showing the ordering of P′s and C′s,

$$P_i \text{->} C_i \text{->} P_{i+N}.$$

A similar proof of such a producer-consumer system was given by Habermann[Habermann,72]. Our use of eventcounts corresponds directly to his use of the auxiliary proof variable ns, which counts the number of times a semaphore is V′ed.

Note that in our producer-consumer example, each eventcount has exactly one writer, in contrast to the usual semaphore solution in which both processes modify the same synchronization variable. This reduction in write competition seems often to occur in eventcount solutions, making correctness proofs easier (as in the example), making confinement feasible (see later section on information flow), and making the problem of synchronizing physically distributed processes easier to accomplish.

The synchronization of the producer and consumer is obtained from the ability of the eventcount primitives to maintain relative orderings of events, rather than by mutual exclusion. A sort of exclusion between the producer and consumer arises out of the ordering constraints, but this exclusion does not require run-time arbitration among events, and so is not mutual exclusion.

Sequencers

Some synchronization problems require arbitration: a decision based on which of several events happens first. Eventcounts alone do not have this ability to discriminate between two events that happen. Consequently, we provide another kind of object, called a sequencer, that can be used to totally order the events in a given class.

A sequencer, like an eventcount, can be thought of as a non-decreasing integer variable that is initially zero. The only operation on a sequencer is an operation called ticket(S), which is applied to a sequencer, and which returns a non-negative integer value as its result. Two uses of the ticket(S) operation will always give different values. (1) The ordering of the values returned corresponds to the time-ordering of the execution of the ticket operations.

The inspiration behind the ticket operation is the automatic ticket machine that is used to control the order of service in bakeries or other busy stores. The ticket machine gives out ascending numbers to people as they enter the store, and by comparing the numbers on the tickets one can determine who arrived at the ticket machine first. Furthermore, the person at the counter can serve the customers in order by calling for the customer whose number is one greater than the one previously served, when he is ready to serve a new customer.

Unlike eventcounts, sequencers do use a form of mutual exclusion. The events corresponding to two ticket operations on the same eventcount may not be concurrent. The precise definition of sequencers in terms of the partial ordering -> is:

1) if T and T′ are events corresponding to ticket operations on the same sequencer, S, then either T->T′ or T′->T.

2) If T is an execution of t:=ticket(S) then the value assigned to t is the number of elements of the set {X | X is execution of a ticket operation on S and X->T}.

(1) And therefore arbitration is required in the underlying implementation.

The use of sequencers can be best illustrated by an example. Let us introduce multiple producers in our producer-consumer example. We want all deposit operations to be mutually exclusive, but are unwilling to place an _a priori_ sequence constraint on the several producers. We use a sequencer called T, which is used by each producer to obtain a "ticket" for depositing its message into the buffer. Having obtained a ticket, a process merely waits for the completion of all producers that obtained prior tickets. Each producer thus executes the following program:

```
procedure producer()
  begin integer t;
    do forever
    begin
         comment synchronize with producers;
         t := ticket(T);
         await(IN,t);

         comment synchronize with consumer;
         await(OUT,t-N+1);
         buffer[(t+1) mod N] := produce();

         advance(IN);
    end
  end
```

The consumer process executes the same program as before.

This program works by using the total ordering among the _ticket_(T) operations to totally order the stores into the buffer array. The _await_(IN,t) operation does not terminate until the _advance_(IN) operation of the producer that got the value t-1 from its _ticket_ operation is executed. This _advance_(IN) operation also enables the consumer process to read the value just stored, so reading of cell t _mod_ N can proceed concurrently with storing into cell (t+1) _mod_ N. While we could have used an eventcount other than IN to synchronize the producers (making the program more "structured" perhaps), we chose not to do so to illustrate the "broadcast" nature of _advance_. Each execution of _advance_(IN) wakes up two processes in this example.

Building Semaphores

Eventcounts and sequencers can be viewed as primitives at a lower level than semaphores. We can build semaphores out of eventcounts and sequencers, and in addition, can build some more powerful operations on these semaphores.

A semaphore can be built out of an eventcount and a sequencer. Call the semaphore S, and the eventcount component S.E and the sequencer component S.T. The P and V operations on S can be written as follows:

```
procedure P(S)
   begin integer t;
      t:=ticket(S.T);
      await(S.E,t)
   end

procedure V(S)
   begin
      advance(S.E)
   end
```

The P and V operations work in a way analogous to queueing in a bakery or barbershop. The ticket operation assigns numbers to each process attempting a P-operation. The process then waits until the corresponding number is announced by the advance operation in V. The difference between the current "values" of S.T and S.E corresponds to the value of the semaphore.

Another operation one can do on semaphores built with eventcounts and sequencers is a simultaneous-P. This operation, which has been proposed by several authors [Patil,71], involves suspending the invoking process until it can successfully lock several resources simultaneously. Such a primitive could be used to solve Dijkstra's problem of the "Five Dining Philosophers" [Dijkstra,71] very simply, by associating a semaphore with each fork, and having each philosopher use a simultaneous-P operation to seize the two desired forks at the same time.

The simultaneous-P operation on 2 semaphores, R and S, could be coded as follows. (1) We require a global semaphore (implemented with a sequencer and an eventcount), G, that is used to synchronize a part of the simultaneous-P operation.

```
procedure Pboth(R,S)
  begin integer g, r, s;

     comment first lock coordinated ticket generator;
     g:=ticket(G.T);
     await(G.E,g);

       comment get a coordinated set of tickets;
        r:=ticket(R.T);
        s:=ticket(S.T);
     advance(G.E);

     comment now wait for each semaphore in turn;
     await(R.E,r);
     await(S.E,s);
  end
```

Several things should be noted about the simultaneous-P operation. First, the G semaphore is used as a global lock on getting tickets from the sequencers of R and S, but not on the waiting. Consequently, there is no waiting going on under a lock. Second, the <u>await</u> operations on R.E and S.E may be deferred until later in the program.

The ability of eventcounts and sequencers to implement the simultaneous-P operation arises directly from breaking the semaphore P operations into two parts, one "enqueueing" the process for use of the semaphore, and the other waiting for the semaphore to become free.

(1) The more general operation for any number of semaphores can be easily seen once the operation of the 2 semaphore simultaneous-P operation is shown.

Flow of information in Eventcounts and Sequencers

An eventcount is an abstraction that allows signalling and observation of events in a particular class. As such it is an information channel among processes. The eventcount operations, read, await, and advance, are defined so that each is either a pure observer of events in the class, or a pure signaller.

We can contrast this purity with the semaphore abstraction. Like an eventcount, a semaphore is a channel that allows the signalling and observation of events in a class. A V operation, like the eventcount advance, is a pure signaller. On the other hand, the P operation on a semaphore is not a pure observer -- it modifies the semaphore, an event that can be observed through other P operations.

These observations about the flow of information make the eventcount synchronization primitives attractive for secure systems that attempt (as far as possible) to solve the confinement problem defined by Lampson [Lampson,73]. For each eventcount, one can provide two kinds of access permission to each process. The first kind, signaller-permission, allows a process to use the advance operation on the eventcount, and thus transmit information to observers of the eventcount. The second kind, observer-permission, allows a process to use the read and await operations on the eventcount, and thus observe information transmitted by signallers of the eventcount. By appropriately assigning access permissions to processes, one can ensure that no information is transmitted from secure processes to processes that do not have the right to directly access secure information.

Because semaphores do not have pure observation primitives, the use of access permissions is not sufficient to solve the confinement problem in this form, and requires more extensive examination of the way the processes use the

P and V operations to ensure that the exclusion provided by P and V operations does not implicitly carry secure information.

Unlike eventcounts, sequencers do not have pure observation and pure signalling operations. Consequently, information is transmitted through a sequencer and received from a sequencer each time the ticket operation is used. Thus, permission to use a sequencer is like having both signaller- and observer-permission.

Secure Readers-Writers Problem

As an example of using access permissions on eventcounts to control the transmission of information, consider a special readers-writers problem. We will call this problem the secure readers-writers problem, to distinguish it from other such problems [Courtois,71]. There are two groups of processes, called the readers and the writers. These processes all share access to the same data base. It is the job of the writers to perform transactions on the data base that take the data base from one consistent state to another. For this reason, the writers need to both read and update the data base, and thus inherently have the ability for two-way communication with all other writers.

The job of the readers is to extract information from the data base. We presume that extracting the desired information may require several operations, so there is a problem of ensuring that the information obtained by the reader is self-consistent, i.e. all accesses referred to the same data base state.

We wish to ensure that readers cannot use the data base or any associated synchronization mechanisms to transmit information to writers or to other readers. (1) In addition, we wish to ensure this without having to know in advance the programs that the readers and writers will execute. Without knowing the program, we cannot ensure correct operation, but we can ensure secure operation as long as we can be sure that the access controls work right.

The first step in the solution to this problem is to require that readers have permission only to read from the data base, and have only observer-permission to any eventcounts associated with the data base. Then this condition guarantees the security requirement, by forbidding readers to use advance and ticket operations. We may then be concerned that the access constraints may have also eliminated the possibility of correctly synchronizing the readers with the writers.

The only difficulty is that the readers are not allowed to exclude writers in order to ensure a consistent set of accesses to the data base. Thus the readers must perform all the accesses, then abort the reader operation and retry the accesses if a writer modified the data base while the reader was reading. The following programs will work correctly in this way (S and C are eventcounts initialized to zero, and T is a sequencer):

```
        procedure reader()
          begin integer w;
abort:       w:=read(S);
             await(C,w);
             "read data base";
             if read(S)≠w then goto abort;
          end
```

(1) Such a constraint arises in information flow control models such as the MITRE security model [Bell,73].

```
procedure writer()
  begin integer t;
     advance(S);
     t:=ticket(T);
     await(C,t);
     "read and update data base";
     advance(C);
  end
```

This solution to the secure readers-writers problem is closely related to the proposal of Easton [Easton,72] for eliminating long term interlocks. The eventcounts here correspond to Easton's version numbers. Another solution using version numbers to the secure readers-writers problem is given by Schaefer [Schaefer,74].

Implementation of Eventcounts and Sequencers

In this section we primarily discuss those issues of implementation that arise in a system that allows processes to communicate through shared memory. Another paper in preparation will discuss in detail the issues of implementing eventcounts in a physically distributed system [Kanodia,77].

If eventcounts are used to synchronize cyclic processes that never terminate, one may be concerned that the values of eventcounts cannot be stored in a finite amount of memory. In practice, however, one can always bound the values that will be encountered, since no real system will operate forever.

In this sense, eventcounts are a convenient abstraction, just as the datatype integer is a convenient problem solving abstraction for Algol programmers. In solving a problem by using eventcounts and sequencers, one can first assume that eventcounts are unbounded. When the problem is solved under that assumption, then one can use practical constraints to bound the values of eventcounts and reserve storage for them. Proceeding in this manner is exactly analogous to dealing with the limitations on Algol integers.

In discussing implementation of eventcounts, we will concentrate on the implementation of <u>advance</u> and <u>read</u>, since an implementation of <u>await</u> is highly dependent on the structure of the operating system (implementation of <u>await</u> in an operating system for Multics is described by Reed in his S.M. thesis [Reed,76]).

The major difficulty with implementing the <u>read</u> and <u>advance</u> operations in a shared memory, multiprocessor system is to allow <u>read</u> and <u>advance</u> operations in any number to proceed simultaneously, without some more basic form of hardware interlock. If an eventcount could fit in a single storage word, and the memory provided an access command that allowed a memory word to be incremented in one memory cycle, while excluding other such commands, implementing eventcounts would be simple. However, this would also be taking advantage of a lower-level mutual exclusion mechanism in the memory.

We will implement eventcounts in two stages. First, we define the concept of a single manipulator eventcount, and show how a full eventcount can be implemented as the sum of single manipulator eventcounts. Then we will show how a single manipulator eventcount can be built using just load and store operations on memory words without mutual exclusion.

A single manipulator eventcount is an eventcount that works correctly as long as concurrent execution of <u>advance</u> operations is avoided. One can assure this avoidance by restricting the eventcount to be manipulated by only one process or by only one physical processor (if there is no process-switching during <u>advance</u>). In a moment we will demonstrate an implementation that allows any number of <u>read</u> operations to execute concurrently with the <u>advance</u>, but for now let us assume that such an implementation is feasible.

One can then construct a multiple manipulator eventcount that can be the subject of concurrent advances by any of N different processes (or processors) simply by implementing the eventcount as N single manipulator eventcounts. advance on the multiple manipulator eventcount will advance the appropriate single manipulator eventcount. read on the multiple manipulator eventcount is done by reading the component single manipulator eventcounts, in any order, and summing the resulting values. Assuming single manipulator eventcounts work as advertised, any number of processes can then advance the multiple manipulator eventcount concurrently. In distributed systems, where a number of systems with local shared memory are connected with communications lines, an eventcount that can be manipulated at multiple sites can be built out of eventcounts that can be manipulated only at their "home" site in the same way. An advance is performed at the home site, and read is done by summing the component eventcounts at all sites.

For the single manipulator eventcount implementation, let us assume the least imaginable form of hardware support, namely that each bit of the eventcount must be read and written separately. (1) We can still insure that the advance operation is atomic by representing the eventcount in Gray code [Kohavi,70], (2) so that incrementing an eventcount involves writing a single bit.

(1) We describe an implementation under these conditions primarily as an existence proof; the practical application is to convince ourselves that eventcounts can be implemented in a minicomputer whose wordsize is too small to contain a whole eventcount.

(2) The suggestion to use Gray code, which considerably simplified our original single manipulator eventcount implementation, is due to Lamport [Lamport,PC], who credits C.S. Scholten with the idea.

The _read_ algorithm can only fetch each bit separately. It must take care that the value it obtains is one that corresponds to some value reached by the eventcount _during_ the execution of the _read_ operation. For this reason, the _read_ algorithm fetches each bit twice, first fetching each bit in the eventcount from high-order bit (least rapidly varying) to low-order bit, and then starting with the low-order bit and going towards the high-order bit. If a bit that has changed during the _read_ is encountered on the second pass, one knows immediately two things: first, at least one _advance_ has happened, and second, at least one value the eventcount must have had during the time the _read_ was in progress. The following algorithm _reads_ an eventcount stored in array E, where the high-order bit is in E[1] and the low-order bit is in E[L], putting the result in array v[1:L].

```
procedure read(E)
  begin integer i, fence;
     for i:=1 to L do v[i]:=E[i];

     fence:=0;
     for i:=L-1 to 1 step -1 do
          if v[i]≠E[i] then fence:=i;

     if fence≠0 then
          begin
            v[fence+1]:=1;
            for i:=fence+2 to L do v[i]:=0;
          end;
  end
```

The variable fence is used to point to the highest-order (lowest subscripted) bit for which a change is detected between successive accesses. A proof of this algorithm would take too much space in this paper, but it is fairly easy to justify its correctness. By induction, one can show that if bits i down to 1 of E and v compare equal in the second loop, then bit i of E was never modified between the access in the first loop and the access in the second loop. Thus, fence points at the highest-order bit that changed during the read. At the time it changed, the low order bits must have been in the configuration of a 1 in the highest-order position, and 0 in any remaining positions. Thus, if any bits have been changed, we just reconstruct the value the eventcount had at the time it was changed, by filling in a 1 at position fence+1, and zeros in the rest.

Implementation of ticket on a sequencer requires some form of mutual exclusion. Lamport has shown one method of obtaining mutual exclusion under our assumption that each bit is read separately [Lamport,76b] [Lamport,76c]. Such an algorithm could be used in the absence of a memory interlock feature. We would like to discover a simpler algorithm that is specialized to the problem of getting a value from a sequencer, rather than providing general mutual exclusion, since we can obtain mutual exclusion at a higher level.

Conclusions

The basic result presented in this paper is a new mechanism for synchronizing concurrent processes that is not based on mutual exclusion. Instead, a process controls its synchrony with respect to other processes by observing and signalling the occurrence of events in classes associated with objects called eventcounts. Since our mechanism directly controls the sequencing of events, it seems to be very natural for implementing parallel computations specified in terms of the ordering of events, as suggested by Greif [Greif,75].

Like semaphores, eventcounts serve as well-defined and namable interfaces between processes. Such an intermediate level of naming enhances modularity, since one need not name processes directly. In contrast to semaphores, however, eventcounts provide a kind of broadcast mechanism that allows the signaller of an event not to have to know how many processes to wake up when an event happens.

Eventcount solutions to problems of synchronization also seem to explicitly identify the information flow paths between processes that are inherent in the synchronization. This identification seems to help in simplifying proof of correct synchronization, as well as enabling more effective confinement of information in a secure system.

A final benefit arising from the avoidance of mutual exclusion where possible is that unnecessary serialization of processes can be avoided -- especially important, since serialization can be a bottleneck in multi-processor systems or physically distributed systems.

References

[Bell,73] Bell, D.E., and LaPadula, L.J., Secure Computer Systems: A Mathematical Model. The MITRE Corporation, MTR-2547, Volume II (November 1973).

[Courtois,71] Courtois, P.J., Heymans, F., and Parnas, D.L., Concurrent control with "readers" and "writers". CACM 14, 10 (October 1971), pp. 667-668.

[Dijkstra,68] Dijkstra, E.W., Cooperating Sequential Processes. in Programming Languages (Ed. F. Genuys), Academic Press, New York, 1968.

[Dijkstra,71] Dijkstra, E.W., Hierarchical Ordering of Sequential Processes. Acta Informatica 1, (1971), pp. 115-138.

[Easton,72] Easton, W.B., Process Synchronization without Long-Term Interlock. Proceedings of the Third ACM Symposium on Operating System Principles, (Operating Systems Review 6, 1 and 2) (June 1972), pp. 95-100.

[Greif,75] Greif, Irene, Semantics of Communicating Parallel Processes. M.I.T. Project MAC TR-154 (September 1975).

[Habermann,72] Habermann, A. Nico, Synchronization of Communicating Processes. CACM 15, 3, (March 1972), pp. 171-176.

[Hoare,74] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. CACM 17, 10 (October 1974), pp. 549-557.

[Kanodia,77] Kanodia, R.K., and Reed, David P., Synchronization in Distributed Systems. paper in preparation.

[Kohavi,70] Kohavi, Z., Switching and Finite Automata Theory, New York, McGraw-Hill, 1970, pp. 12-14.

[Lamport,76a] Lamport, Leslie, Time, Clocks, and the Ordering of Events in a Distributed System. Massachusetts Computer Associates Technical Report CA-7603-2911 (March 29, 1976).

[Lamport,76b] Lamport, Leslie, Synchronization of Independent Processes. Acta Informatica 7, 1 (1976), pp. 15-34.

[Lamport,76c] Lamport, Leslie, On Concurrent Reading and Writing. submitted to CACM.

[Lamport,PC] Lamport, Leslie, private communication with the authors.

[Lampson,73] Lampson, Butler W., A Note on the Confinement Problem. CACM 16, 10 (October 1973), pp. 613-515.

[Patil,71] Patil, S.S., Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes. M.I.T. Project MAC Computational Structures Group Memo 57 (February 1971).

[Reed,76] Reed, David P., Processor Multiplexing in a Layered Operating System. S.M. and E.E. thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, (June 1976). Also available as M.I.T. Laboratory for Computer Science Technical Report TR-164.

[Schaefer,74] Schaefer, M., Quasi-synchronization of readers and writers in a secure multi-level environment. System Development Corporation TM-5407/003, (September 1974).