PROJECT MAC
Computer Systems Research Division

January 12, 1976
Request for Comments No. 102

# EVENTCOUNTS: A NEW MODEL FOR PROCESS SYNCHRONIZATION

by Raj Kanodia and David P. Reed

This paper is a first attempt to present a new model of synchronization that we have developed over the past couple of years. In the near future we intend to publish a more extensive version as a Project MAC Technical Report, and a summary paper in some journal. We would appreciate your comments on both technical content as well as style of presentation.

---

Table of Contents

# Chapter 1

## Introduction

In this paper, we present an alternative model of synchronization. The most common existing models of synchronization are based upon the principle of mutual exclusion and shared data to achieve synchronization. We shall argue that both these characteristics of the synchronization models (1) have undesirable effects upon security properties of solutions realized in these models, (2) limit their applicability to distributed systems, and (3) are dependent upon environment specific properties of systems (i.e., representation of data, semantics of operations on data, etc.) In addition, some simple synchronization problems are quite hard to solve in such models.

Our model of synchronization is an event oriented model of synchronization in which processes coordinate their activities by signalling and observing occurrences of events via synchronization variables, called eventcounts and sequencers. Thus, these synchronization variables become interfaces for all interaction among processes and a process normally does not need to know the names or locations of other processes. This has a major simplifying influence on proving correctness of synchronization programs.

There exists a very strong notion of explicit and implicit information flow paths between processes and synchronization variables in our model. This makes it possible to solve some problems that could not be solved in existing models in a manner consistent with their security requirements. The notion of information flow is directly related to dependencies among processes in the system that affects their recovery from failure. It is easier to study such issues in this model.

The basic synchronization objects, eventcounts, do not require mutual exclusion for their implementation. It makes it possible to solve many synchronization problems in which mutual exclusion is not an intrinsic requirement, without resorting to mutual exclusion. Our model encompasses two of the most widely studied models of synchronization - namely P-V [9] and block-wakeup [28]. Thus, while all programs constructed for these two models can be directly realized in our model, it provides some interesting new solutions, which are easier and have fewer dependencies among cooperating processes.

This model makes no assumptions about the environmental properties of systems and therefore it is directly applicable to distributed systems, i.e. those systems in which due to lack of a shared memory, all communication among processes must be via communication channels involving unpredictable time delays. We have also found a "robust" implementation in the distributed systems case where the communication channels themselves may be unreliable.

We have been inspired in this work by many others working in the field of synchronization, among whom are Brinch Hansen [1,2], Dijkstra [5,6,7,8,9], Lamport [17], Johnson and Thomas [16], Saltzer [28], Rappaport [27] Easton [10], Greif [11], Parnas [24], Lipton [21], and the ARPANET Host-Host Protocol Committee [22].

The overall organization of this paper is as follows. The second chapter describes the eventcount model of synchronization. In that chapter, we also demonstrate how to construct semaphores in this model. In the third chapter, we discuss security aspects of our model and present a solution of the hitherto unsolved problem in which a set of writers and readers sharing a data base must achieve all synchronization without introducing information flow paths from the readers to the writers. In the fourth and fifth chapters, we present solutions to some classical and some new synchronization problems. The sixth chapter is devoted to a comparison

of the new model with the existing models of synchronization. In the seventh chapter, we show how recoverable communication protocols between remote computers can be realized within our model. In the eighth chapter, we describe methods of implementing the eventcount model in shared memory as well as distributed systems. In the ninth chapter, we give an axiomatic description of the eventcount model and apply it to verification of synchronization conditions.

# Chapter 2

## A New Model of Synchronization

Our model of synchronization is somewhat different from the most commonly discussed model - namely semaphores. In the semaphore model, the approach is to maintain a complete description of the state of the system in a central store and allow individual processes mutually exclusive access to it, so that processes can obtain the state in a consistent form. In our model, a synchronization problem is expressed in terms of timing constraints [11] on occurrences of events. Events are divided into event classes and events of a given class are represented by an associated synchronization variable of the type "eventcount". There exist primitive operations that permit processes to signal and observe occurrences of events. The timing constraints of a synchronization problem are realized using these primitives.

Some synchronization problems require a special kind of timing constraint that can not be realized by eventcounts alone. These problems have the characteristic that the order of different activities is not specified in advance, but rather the synchronization system dynamically defines a total order among them. To realize this type of constraint, we use synchronization variables called "sequencers". There is only one primitive operation defined for a sequencer.

Next, we discuss eventcounts and sequencers in greater detail. Then we show how to realize semaphores (and associated operations) in our model.

## 2.1 Eventcounts

The main purpose of an eventcount is to keep a count of the number of events of a particular class that have occurred in the past. An eventcount can be thought of as an integer variable whose value never decreases (because events can not un-happen).

We define an advance primitive to signal the ocurrence of events and three primitives observe, await, and read to obtain values of eventcounts. As we shall see shortly, the observe primitive is not necessary unless some method of expressing busy waiting is desired.

### 2.1.1 Signalling Events

We define a primitive operation advance(E) 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. A process with permission to perform an advance operation on an eventcount is said to be a "signaller" or "manipulator" of that eventcount. An advance operation takes a finite amount of time to execute. While an advance is in progress, it is not specified as to when the integer value of the eventcount changes; it is only necessary to ensure that it change sometime before the operation is completed. The value of the eventcount equals the number of advance operations performed on it (i.e. the initial value of an eventcount is zero).

### 2.1.2 Observation of Events

An observer of E can obtain its integer value by the primitive operation observe(E). If "r" be the value returned, then the only information this primitive provides is that at least "r" events have been signalled via E; one can not assume that more have not been signalled. There may be an arbitrary delay between the time

an advance completes and the time an observe returns the value incorporating the effect of the advance. The reason for this is that we would like to capture the delay inherent in a distributed system. However, the effect of all advance operations will eventually be communicated to all observers and the successive values of observe(E) in the same process will be non-decreasing.

The single-producer, single-consumer system, in which a process, called the producer, is transmitting messages to another process, called the consumer, via an infinite length buffer buf[0:∞] can be realized in terms of advance and observe by utilizing a single eventcount IN (initial value zero) as follows:

```
producer:
   in := 0;
   do forever
      begin
         generate message;
         buf[in] := message;
         advance(IN);
         in := in+1;
      end

consumer:
   out := 0;
   do forever
      begin
         while observe(IN) ≤ out do nothing;
         message := buf[out];
         out := out +1;
      end
```

### 2.1.3 Awaiting Occurrence of an Event

In the consumer program, we see a "busy form" of waiting, which is wasteful of processor time. To eliminate it, we define an await primitive. If "v" be an integer value, then await(E,v) returns in a finite time after observe(E) ≥ v. Using await, the while loop in the consumer can be replaced by:

```
await(IN, out+1);
```

The busy form of await(E,v) is equivalent to the following program:

```
while observe(E) < v do nothing;
```

It may be helpful to the reader to have a visual image of an eventcount and interactions of various processes with it. An advance operation may be imagined to consist of transmitting one message to each of the observers. The value of observe(E) is the number of messages received for E. This particular image provides the intuitive reasons that suggest that the eventcount mechanism is appropriate to distributed systems.[1]

### 2.1.4 Awaiting Occurrence of One of Several Possible Events

Often a process is interested in one, any one, of a number of events in the system. One rather common example of this situation occurs when a process awaiting termination of some i/o activity has also set up a time out event. In a distributed system a receiver might want to wait for "arrival of a message or break down of the sender". To permit realization of such "relationships" between events, we generalize the await primitive to the following:

await($E_1$,$v_1$, ..., $E_n$, $v_n$)

where $E_1$, ..., $E_n$ are eventcount names and $v_1$, ..., $v_n$ are corresponding integer values. This primitive returns in a finite time after one (or more) of the eventcount values becomes greater than or equal to the corresponding integer value.[2]

---

[1] Indeed, this is one method of implementing eventcounts in the distributed systems, although not the recommended one. A better mechanism, which is robust with respect to the loss of messages, is described in the implementation section of this paper.

[2] Merlin and Farber [23] have shown that it is not possible to construct asynchronous recoverable protocols if some knowledge of execution time of the events is not provided. This directly relates to the notion of time-out and the generalized await

Even though we have defined the observe primitive for eventcounts, in practice it is not necessary to use the observe primitive unless some method of expressing busy waiting is required. Thus, we are left with only two primitives: advance and await.

### 2.1.5 Obtaining the Current Value of an Eventcount

The observe primitive is defined to return a value that is monotonically increasing but not necessarily up-to-date. The reason for this apparently relaxed definition will become clear later. Meanwhile, there are synchronization situations in which an eventcount may be used to represent the state of a shared resource or a data base. In such a situation we would like to be able to obtain a current value of an eventcount.

One method of using the current value of eventcounts follows. Consider a process that has just finished an event A. If this process now obtains a current value r of an eventcount B, then it is certain that no more than r events of type B had completed when A finished. Therefore, the process can be certain that A preceded the (r+1)th event of type B (indeed the (r+1)th event of type B may never occur). When is such a relationship useful? Consider a data base being shared between a reader and a writer. We wish that the writer be able to proceed any time it wishes without having to wait for a read to complete. At the same time, we want the reader to be able to determine if its read operation of the data base overlapped with any write operation so that it can determine whether the data base was in a consistent state or not during the read operation. The problem can be solved as follows: Before beginning to read the data base, the reader obtains a curent value (C) of an eventcount that denotes the number of times write operations have completed. Then

---

primitive is one method of realizing such a protocol. In a later section of this paper, we give an example of such a protocol.

after it has finished reading the data base, the reader obtains the current value (S) of an eventcount that denotes the number of times write operations were started. If S>C then it is possible that a write operation was started during the read operation and therefore the read operation might have obtained inconsistent results and must be repeated.[1]

For this reason, we define an additional primitive operation read(E) that returns the current value of an eventcount E. The essential difference between observe and read can be explained by describing the operations in terms of message transmissions. As before, each advance operation transmits a message to each of the observers. The observe operation simply counts the number of advance messages received. But the read operation waits until it has received all the messages that were transmitted at or before the time at which the read was initiated, and then it counts the number of advance messages received. Thus, there is an inherent delay in the execution of read, caused by transmission delays in the communication channels.

Let $t_s$ and $t_c$ denote the times at which a read operation starts and completes. Let S(E,t) and C(E,t) denote the number of advance operations started and completed at or before time t on eventcount E. Then the value "r" returned by the read(E) is defined by the following:

$$C(E,t_s) \leq r \leq S(E,t_c)$$

By using read, a process ensures that the effect of all advance operations that were completed at or before the time at which read started, has been communicated to it. However, the value returned by read may also include the effect of those advance operations that were initiated after the initiation of, but before completion of, the read.

---

[1] This technique is a variation of the version number technique of data base synchronization [10]. A general version of this problem is discussed later on in the section titled "Security".

Since there are no delays in the communication paths in shared memory systems, read and observe for a shared memory, single word eventcount are identical. In distributed systems, the distinction between observe and read is critical. Possible implementation strategies for shared memory and distributed systems are described in the implementations section of this paper.

### 2.1.6 Some Comments on Eventcounts

Before proceeding further, we make the following comments regarding eventcounts:

1-Most synchronization problems involve cooperation among cyclic processes. Thus an eventcount whose value is increased by one in each cycle, is a natural method of representing the history, and therefore the state, of a cyclic process.

2-A primitive operation on the eventcount never interferes with other primitive operations on the same eventcount. This makes it practical to implement eventcounts in distributed systems.

3-One consequence of our definition of the await primitive is that upon return from await(E,v) the following assertion remains true forever:

$$V(E) \geq v$$

where V(E) is the value of the eventcount E. This property of the await primitive has a major simplifying effect on verification of synchronization properties of programs. This aspect of eventcounts is fully explored in a later section of this paper.

4-It is possible to implement eventcounts such that the only information transfer through an eventcount occurs from the signaller processes to the observer processes. When such an implementation is used in conjunction with appropriate protection mechanisms, it is possible to control the flow of information through eventcounts and thus provide solutions for problems that could not be solved within

the existing models. One section of this paper, titled "security", is devoted to a thorough discussion of the protection aspect of our model.

## 2.2 Sequencers

We define a new type of synchronization object, called a "sequencer", for which there exists only one operation - namely ticket. The ticket operation returns an integer value, called a "ticket". If a ticket operation does not overlap in time with other ticket operations on the same sequencer then its value equals the number of ticket operations performed on the sequencer in the past. Furthermore, no two ticket operations on the same clock ever return the same ticket. Essentially, a sequencer is a natural number generator and returns the sequence 0,1,2,..., etc. A ticket operation is guaranteed to terminate.

Obtaining tickets from a sequencer is similar to obtaining tickets from ticket-machines in large department stores. The ticket operation is a request for service of a shared resource. Its result, a ticket, is an authorization for the use of that shared resource. A ticket also contains information as to when the authorization becomes effective. If several processes request authorization at approximately the same time, the sequencer imposes an ordering on the authorizations it grants.

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 apriori 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 completion of all events previously requested through T. Each producer executes the following program:

```
producer_i:
   integer t;

   do forever;
      begin
         generate message;
         t:=ticket(T);
         await(IN, t);
         buf[t] := message;
         advance(IN);
      end
```

The consumer process executes the same program as before.

In terms of message transmissions, a "sequencer" may be viewed as a single message containing the value of the sequencer and continuously circulating in a ring through each process that might perform the ticket operation. A ticket operation consists of stopping the message, reading its value, and incrementing it by one before letting it proceed further.

Before proceeding further, we briefly summarize the machinery we have constructed for synchronization. We have defined eventcounts with the four operations: advance, observe, await, and read. Eventcounts are sufficient to synchronize activities where order can be specified in advance. In practice, the observe primitive may be eliminated unless it is necessary to express busy waiting. We also defined a "sequencer" for which there is only one operation - namely ticket. Sequencers are required where the order of activities is determined dynamically; that is, wherever mutual exclusion is called for.

## 2.3 Constructing semaphores

Because a ticket operation is guaranteed to terminate, we can define both fair and unfair semaphores[1] as well as a wide variety of scheduling disciplines. Let's examine a way to obtain a fair semaphore using these primitives.

A fair semaphore, s, will consist of a sequencer, s.T and an eventcount, s.E. The P and V operations can be defined as follows:

```
P(s): t := ticket(s.T);
      await(s.E, t);

V(s): advance(s.E);
```

Now if we enclose a critical section of code with these P and V operations, we can guarantee that a) at most one process can be executing the critical section at a time, and b) that any process that begins the ticket operation in P will eventually enter the critical section -- in other words, the semaphore is fair. The difference s.E-s.T corresponds to the value maintained by an ordinary semaphore implementation.

By changing the initial values of the eventcounts that comprise the semaphore, one can allow more than one process in the critical section. Thus, we really have defined a general semaphore.

The following observations may help develop some intuition about the relationship between semaphores and eventcounts. The semaphore s has been split into two parts, s.T and s.E. The part s.E is written into only by those processes that perform the V operation on s and s.T is written into only by those processes that perform the P operation. P and V never write in the same part. It is this separation that allows us to clearly identify and control the channels of information

---

[1] A fair semaphore is one in which a P operation by a process cannot be prevented forever from completing by the repeated execution of P operations in other processes(assuming sufficient V operations are occurring, of course).

flow. The ability to identify the intrinsic information flow paths of a synchronization problem also helps its implementation in distributed systems. Corresponding to the splitting of semaphore, the P operation is also split into two parts - the queueing operation and the waiting operation. Thus scheduling, which is implicit in the semaphore model, has been made explicit in the eventcount model.

## 2.4 On Storing Eventcounts

If eventcounts are used to synchronize cyclic processes that never terminate, one may be concerned that the value of the eventcount cannot be stored in a finite amount of memory. In practice, however, one can always bound the values that will be encountered, since no system can 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, one can first assume that eventcounts are unbounded, and when the problem has been solved, one can determine practical bounds for the values of the eventcounts and reserve enough storage for them. Proceeding in this manner is exactly analogous to dealing with the limitations on Algol integers.

# Chapter 3

## Security Properties

In order to discuss the properties of eventcount and sequencer objects with respect to the flow of information in the system, we must identify the ways in which the primitive operations of our model transmit information. First, we concentrate our discussion on eventcounts. There are two important ways in which the eventcount operations can transmit information from process to process. First of all, advance operations may occur, changing the value of the eventcount, and this change may be observed by some process. This we will call an explicit information path, since the purpose of the operations on eventcounts requires that the value of an eventcount be manipulated and observed. The other way in which an eventcount can transmit information is through the timing of its operations. For example, if one has two mutually exclusive operations that may be executed by two processes, those two processes can detect whether they were excluded by observing the time that the operation took to complete. We will call these paths implicit information paths, since they are an incidental side effect of the definition of the operations.

We were careful to provide a definition of the eventcount operations that does not require mutual exclusion of eventcount operations. In an implementation of eventcounts tha does not use mutual exclusion there are no possible implicit information channels, because any number of operations on an eventcount may be in progress simultaneously. At the level of abstraction at which we deal with a shared memory system it is sufficient to implement each eventcount primitive as a single hardware instruction relying on memory arbitrartion to achieve this security

property, since the mutual exclusion implicit in the memory arbitration is undetectable at that level. In the implementations chapter of this paper, we describe some implementations that preserve the security properties of eventcounts.

As we have defined the operations observe, await, and read, they are pure receivers of information. There is no way that any eventcount operation can detect the fact that another observe, await, or read operation has occurred, will occur, or is occurring. However, these operations can detect the fact that advance operations have occurred.

Similarly, the advance operation only transmits information. There is no way that an advance operation can tell if any other eventcount operation has occurred.

By controlling the ability of a particular process to execute particular operations on an eventcount, we can thus control its ability to use that eventcount as an information channel from or to another process. We may do this be defining two orthogonal access permissions that a process may have for an eventcount. The first permission entitles a process to receive information transmitted through the eventcount, and will be called receive-permission, while the second entitles a process to transmit information on the eventcount, and will be called transmit-permission. The observe, read, and await operations can be performed only if the process has receive-permission on the corresponding eventcount. Similarly, advance can be performed only if the process has transmit permission on the eventcount.

The ticket operation on a sequencer both receives and transmits information, since it influences the values returned by future ticket operations in other processes, and since its value is affected by ticket operations executed before it. Thus, if we use the receive and transmit-permissions to control a process's ability to use a sequencer, a process can perform a ticket operation only if it has both

permissions for the corresponding sequencer.

These permission requirements are summarized in the following table:

| Permissions Required | Operations: observe / read / await | advance | ticket |
|---|---|---|---|
| receive-permission | X | | X |
| transmit-permission | | X | X |

In order to show the effectiveness of these security controls in dealing with the confinement problem, we give the following example.

We wish to have several processes share a data base, of which there is one copy in the system. Any number of processes may read the data base simultaneously, but modifications to the data base must not interfere with each other, so processes that make modifications to the data base must be mutually exclusive. The readers should read a consistent version of the data base (that this requirement should be explicitly mentioned is a clue that we shall develop a solution in which a writer may proceed even while readers are reading). We wish it to be the case that there is no way for readers to transmit any information to other readers, or to any writer. That is, we wish to guarantee the confinement of any information held by a reader process within that process. We would particularly like to be able to accomplish this simply by restricting the access permissions granted to reader processes, rather than by proving the whole set of programs used by the reader process.

Assuming that actually reading and writing the data base only receive and transmit information respectively, and that confined processes only have the ability to read the shared data base, we can synchronize the processes accessing the data base without introducing any security problems in the following way. We use a

sequencer S, which is inaccessible to readers, to synchronize writers. We use two eventcounts $W_s$ and $W_c$ to allow readers to discover that a writer has started, and completed, respectively. We also ensure that the confined processes (i.e., readers) only have receive permission on both eventcounts. Thus the eventcounts and the sequencer can not be used to transmit information from the confined processes to any other processes. Then, the confined processes may read the data base, and obtain consistent values by executing the following code:

```
reader:  repeat
             w:=read(W_s);
             await(W_c,w);
             "read data base";
         until read(W_s)=w;
```

While unconfined processes can update the data base by this code:

```
writer:  advance(W_s);
         t:=ticket(S);
         await(W_c,t);
         "read and update data base";
         advance(W_c);
```

These algorithms are closely related to the ideas presented by Easton [10] for eliminating long term interlocks. The eventcounts used here correspond to his version numbers.

With this example, we have shown a way in which a confined process doing useful work can synchronize itself with other processes. Further, it is clear by looking at the synchronization primitives used by a particular process what information paths may be used by that process to transmit information to other processes.

We claim that it is impossible to solve this problem using mutual exclusion among readers and writers, because mutual exclusion primitive will always provide a potential channel by which readers can signal information to writers.

Another simple example of secure synchronization is provided by the earlier

producer-consumer example (with infinite length buffer), in which it is easily seen that, by appropriate use of receive- and transmit- permission, the consumer can be confined from transmitting information to the producer without affecting his ability to receive information from the producer.

# Chapter 4

## Some Classic Synchronization Problems

In this chapter, we show how two of the commonly discussed synchronization problems - resource allocation [6] and readers-writers [4] - can be solved in the eventcount model.

### 4.1 Deadlock-Free Resource Allocation

Suppose we have a large set of resources that may be used by a set of processes, but must be exclusively assigned to a particular process while it is in use (we have built in a kind of mutual exclusion here). Further suppose that processes occasionally require the use of more than one resource simultaneously. If we allow a process to ask for only one resource at a time, we introduce the possibility that deadlock may occur. We would like to avoid deadlock, if possible, and further we would like to guarantee that once a process indicates its need for a set of resources, it will be guaranteed to eventually get them.

Dijkstra[6] discusses a solution to a simple case of this problem as the "five dining philosophers" problem. His solution using semaphores requires major reworking to make it work for different numbers of processes, which might require different numbers of resources. In fact, it is not clear that his solution generalizes to the case where the resource requirements of some processes using resources are unknown to other processes using those resources.

We have discovered an interesting solution to the general problem that works independently of the number of processes, and where each process needs no knowledge of the resource requirements of other processes.

Let's consider the problem of deadlock. Deadlock can arise when process A requests resource C, and then requests D, while process B requests resource D and then requests resource C. At this point, process B is waiting for A to finish, and process A is waiting for B to finish, resulting in deadlock. If we were to seize all resources simultaneously, in a critical section, there would be no problem. Solving this problem with semaphores, where the set of resource-requesting processes is not known in advance, requires an extended semaphore primitive, P-multiple, which can wait on several semaphores simultaneously.

Defining fair scheduling for a P-multiple operation is quite difficult, and in particular should not be left implicit, since the timing behavior of the solution to the problem depends crucially on the exact definition of scheduling in the P-multiple operation. Our solution has each process awaiting one eventcount at a time, and can be shown to be fair, in the sense that a process that begins a request for resources eventually gets them.

We do this by using the ticketing concept, first introduced by Lamport[17]. Each resource will have a sequencer and an eventcount associated with it; one will be used to provide tickets, and the other will be the grantor of resources. We then need only have the process requiring a set of resources take a set of tickets for those resources all at once, and then wait until its turn is reached. A program for obtaining a set of resources, R, follows. The set R consists of the individual resources A, B, C, etc. The sequencer associated with a particular resource, A, will be indicated by subscripting: $T_A$; and the eventcount associated with A will be $G_A$. There is a global semaphore, s, built out of a sequencer s.t and an eventcount s.e in

the way we previously described.

```
use(R):    t:=ticket(s.t)              get tickets for R
          await(s.e,t)                 in a critical section
             t_A:=ticket(T_A)            get tickets for each resource
             t_B:=ticket(T_B)
             ...
          advance(s.e)

          await(G_A,t_A)                wait turn for all resources
          await(G_B,t_B)                in any order
          ...

          "use resources in R"

          advance(G_A)                grant resources to next users
          advance(G_B)                in any order
          ...
```

The process executing this code is guaranteed to get through the "getting tickets" phase in finite time. Once it has gotten through the "getting tickets" phase, it has established its relative position in the queues of processes waiting for resources in R. Thus it will eventually get those resources, before processes that request some overlapping set of resources after its request get any of the resources in the overlapping set. Further, since it gets its tickets effectively simultaneously, deadlock can never occur.

One can imagine the working of this program as follows. Imagine the requesting processes as a group of people. Processes requiring resources line up in the order in which they make their requests. As resources become free, we start with the head of the queue, and ask each person whether they require that resource. The first person in the line to require a resource will be given it, and if he now has a complete set of resources he may proceed to use them.

The ticket values which a process gets are related to this image, since the ticket value a process holds for a resource is equal to the number of processes which have requested that resource before it. This value is just the sum of the number of processes which already have used the resource and the number of processes which have

requested the resource before this process but are in the queue ahead of this process. Consequently, when all of those processes have used the resource, it will be this process's turn.

It is our claim that this program is very easy to understand and prove. It is not our intention to solve the general resource allocation problem, but just to show the utility of the eventcount idea as a tool to describe and solve problems in that area.

The solution we have presented is rather simple, and needs to be improved if it is to be used in an operating system environment. In particular, we need not wait till both resources are available to the process to begin using the first -- the process can begin using resource A, for example, when resource A becomes available, and need only wait for resource B to be freed when resource B is required. Similarly, the process can do the advance operations at the end of the program as soon as the resource is no longer needed.

This is not the only program that can be used to fairly schedule resources -- in particular, we may define other programs that allow resources to be allocated to processes that request them in other than FIFO order, but are still fair. Such programs may achieve better overall resource utilization, in the sense that if another process requires resource A while the process executing the program above would be waiting for B, it may be allowed to proceed.

## 4.2 A Readers - Writers Problem

Another short example that shows the power of the synchronization method we describe is our version of Courtois, Heymans and Parnas'[4] second readers-writers problem. The problem is similar to our earlier problem of secure data-base synchronization. The statement of the problem, however, makes it impossible to confine the reader processes from transmitting information to the writers.

Briefly, the problem is this. There are a number of reader processes and a number of writer processes sharing a data base. Any number of readers may be reading the data base simultaneously, when there is no writer writing in the data base. Only one writer may write in the data base at a time, and may not do so until there are no readers actually reading the data base. Now, the crux of the specification is that writers have priority; that is, if a process requests that it be allowed to write, it should be enabled to do so as soon as possible.

An important requirement is that if a write is in progress, and some reads are waiting for that write to complete, and another write request comes along, it should be promoted ahead of the waiting reads. Furthermore, we add the requirement (not in the original problem) that writes be fairly scheduled, so that any write request is guaranteed to finish and not be locked out forever by reads or other writes. It is clear from the statement of the problem that reads can be locked out forever by a steady stream of writes.

Our solution uses a sequencer S to sequence write requests and four eventcounts,which have the following intuitive meanings:

$R_s$ -- reads started (including aborted attempts)
$R_c$ -- reads completed (including aborted attempts)
$W_r$ -- writes requested
$W_c$ -- writes completed

The programs for reading and writing follow.

```
writer:  advance(Wr);                 Let the readers know about write
         t:=ticket(S);                Obtain a ticket for this write
         if read(Wc)<t                If a previous write has not completed
         then await(Wc,t);               then await its completion;
         else repeat                  otherwise make sure all reads are done.
                r:=read(Rs);
              await(Rc,r);
              until read(Rs)=r;

         write;
         advance(Wc);                 Signal completion of write.

reader:  greenlight := false;
         while not greenlight do
         begin
            await(Wc, read(Wr));        Wait for writers to finish.
            w:=read(Wc);                Get number of writes completed
            if read(Wr)=w               If no pending writers ..
            then begin                    then attempt to read;
                    advance(Rs);            Signal attempt to read.
                    if read(Wr)=w           If still no writers
                    then greenlight := true    then it is OK
                    else advance(Rc)           Otherwise abort attempt
                 end;
         end;

         read;                          Read the data base
         advance(Rc);                   Signal completion of read.
```

In our solution, a writer advances the eventcount $W_r$ to let the readers know about itself, and then it obtains a ticket for its turn to write. At this point if a previous write request has not yet completed then the writer only need wait its completion without worrying about the behaviour of the reader processes. Otherwise it must ensure that no reads are in progress. A reader process first examines if there are any obstacles in its path (i.e., writes pending or in progress). If there are none, then it signals its intention to begin the read operation by advancing the eventcount $R_s$. However, it can not quite begin the read because some writer might have entered a request. Therefore, after signalling its intention to begin reading, the reader proceeds to read only if no writer has announced its intention to write in the meantime. This ensures that writers have priority over readers. If a reader

encounters obstacles in its progress, it simply waits until those obstacles are removed and then re-examines the situation.

The important thing to notice about these programs is their clarity. We expect that the reader will be able to understand how they work quickly, as opposed to the solution of Courtois, Heymans and Parnas, which is quite complex.

The solution we present here is different from the solutions published by Courtois, Heymans and Parnas using semaphores, and is also different from the solution published by Hoare using monitors, in the fact that the reader processes never execute a piece of code which excludes other readers, even for a brief period. For this reason, a delay that affects a particular reader process can never affect the progress of other readers (it may, however, delay a writer).

Some comment should be made about the definition of the time when a writer makes his request. It is never possible for the reader to tell if a writer has requested a write before he began operation, since in any system there is some time delay between the time the write is requested and any reader can know that the write was requested. Thus, the best we can say is that there is some delay between the time the write is requested and all other processes can know it was. This delay is built into the definition of the advance operation in the first step of the writer in our solution. Thus, in our solution, we can guarantee that no reads will be allowed to begin after the advance operation in the writer has been executed.

Another difference of our solution from Hoare's monitor solution and Courtois, Heymans and Parnas' solution is that the only communication between processes is done through the eventcount and sequencer mechanism. In the other two solutions, mutual exclusion is used to protect global variables which are shared among all processes. This can only add complication to a distributed system implementation, since mechanisms for sharing the values of global variables are quite complicated. Since

our implementation does not use shared global variables, the solution generalizes directly to a distributed system, without adding any new mechanisms.

# Chapter 5

## Some New Synchronization Problems

In this section, we discuss some synchronization problems that have received very little attention in the past, and show how to realize their solutions in the eventcount model. First, we show how to incorporate a real-time clock and associated time-out mechanisms into our model. Then we discuss a broad class of problems that are collectively referred to as monitoring problems. Finally, we show how the eventcount model can be used to construct self-stabilizing distributed systems.

### 5.1 A Real-Time Clock and Time-Out Mechanisms

In our model, a real time clock can be considered to be an eventcount and standard primitives (i.e., read and await) can be used to obtain the clock values and set up time-out mechanisms. We assume that there is a hardware process that is advancing the clock at regular intervals; a single interval being the unit of time represented by the clock. If we let "time" be the eventcount representing a real time clock, then read(time) obtains the time of the day. A process can use the following program to suspend itself for an interval of "interval" time units by executing the following program:

```
blocktill := read(time) + interval;
await(time, blocktill);
```

Similarly, a process awaiting occurrence of some event on an eventcount E (await(E,v)), can set up a time out mechanism as follows:

```
alarm := read(time) + interval;
await(E, v, time, alarm);
```

In many computer systems, there exist programs for generating unique identifiers from a real time clock for various purposes. The following program, which uses an eventcount E and a sequencer S (both initialized to zero), generates such unique identifiers from the clock "time" and uses only standard primitive operations of our model:

```
procedure unique_id(id);
begin
   t:=ticket(S);                          Get a ticket
   await(E,t);                            Wait for turn
   id:=read(time);                        obtain time value
   while read(time)=id do nothing;        make sure no one else gets the same
   advance(E);                            Enable the next process to enter
end
```

## 5.2 Monitoring

Most of the examples in this paper require some kind of exclusion. This is because most "synchronization problems" to be found in the literature have some kind of exclusion in their specification. We suspect that this bias toward thinking of synchronization as being exclusion has left untouched an important class of synchronization problems, where exclusion is not part of the specification of the problem. In this section of the paper, we discuss a number of relatively simple problems that do not require exclusion, but rather much simpler synchronization specifications.

Our earlier example of the producer-consumer problem with unbounded buffer is such a case. Here, each slot in the buffer starts out empty, then is filled by the producer, and then is used by the consumer. The synchronization specification for this problem is quite simple. The consumer must always access a slot after the producer accesses it. The simplicity of this specification results in the fact that we can solve the problem without the use of the ticket operation.

There is a particularly important class of non-exclusive synchronization problems which we call monitoring. Suppose we have an object whose state changes over time. An example of such an object might be a data base (where the state consists of its contents), a page of virtual memory (whose state might be either in primary memory or out of primary memory), or some processor on a distributed processor system (whose state might be down or up). If for some reason we are interested in observing the changes to the object as they progress, but cannot interfere with the agent making the changes (as might be the case with the down or up state of a processor which is unreliable), we cannot be sure of the state at any particular time. We must be satisfied with the fact that any state observed by a process may be out of date by the time we look at it. We can wait for the system to change to a particular state, but by the time we complete the wait, the system may have changed to another state. If a process is operating under these ground-rules, but is observing changes to the state of some object, we will say it is monitoring the object.

We can use the values of an eventcount to represent the state of a monitored object, if we always change the eventcount with advance or update[1] whenever the object's state is changed. An example of this concept can be found in the Multics operating system file storage backup facility. Associated with a file is a variable called "date/time modified", which holds the date and time of the last modification to the file. The backup procedure wants to write a copy of the file to tape after it

---

[1] Quite often, we need to use the following program construct for advancing single manipulator eventcounts (i.e. eventcounts that are advanced by only one process):

```
while s > read(E) do advance(E);
```

Its effect is to change the eventcount value to "s" from "r". For the sake of clarity and brevity, we will use the following equvivalent operation:

```
update(E,s);
```

has changed. It does not want to copy the file repeatedly when there are no changes. Further, it is desirable that the backup procedure not interfere with the process making the changes. What the backup procedure does, then, is wait until the "date/time modified" of a segment exceeds the value it had when it previously was copied. We can say that "date/time modified" is an eventcount. For each segment, backup logically has a process which does the following:

```
while true do
     begin
          time:=read(segment.dtm)
          "copy the segment"
          await(segment.dtm,time+delta)
     end
```

Another interesting example is to be found in a network of computers such as ARPA Network. In such a network, individual machines often go down either due to system crashes or planned shutdown in such a manner that all logical communication paths between machines and processes must be aborted and each communicating process or machine be informed of the shutdown. If a machine comes up shortly after an unplanned shutdown, it is possible that it will receive messages directed towards its earlier instance due to the possibility that information of it going down has not had a chance to reach all the machines in the network. For proper operation, it is essential that a machine be able to detect these "spurious" messages and discard them. The solution presently adopted in the ARPA Network Host-to-Host protocol [22] is that no communication between two machines is established until they have exchanged a pair of RESET messages. This technique normally works except when a machine goes down and comes up several times in quick succession. Coupled with this is the problem that a process faces when it tries to establish multiple logical connections with a single instance of another machine. The problem is that when a process attempts to establish several logical connections with another machine, it has no method of determining whether all the connections were made with one instance

of the remote machine or not. Suffice here to say that no straightforward solutions have been proposed.

Rather simple and elegant solutions can be obtained using eventcounts. First of all, how can we model the up/down status of a machine? Our model uses two eventcounts per machine, one called upcount and the other called downcount. When a machine comes up, it increases its upcount by one. When a machine goes down, its downcount is increased by one by some other machine that has determined the fact of the former machine going down. Now, processes can monitor the state of another machine merely by observing its eventcounts. If the downcount of a machine is greater than or equal to the upcount then the machine has definitely gone down since the time of obtaining the value of upcount.

Another useful feature of this representation of the system is that we can associate with each logical communication path the values obtained by read(upcount) of both machines. This enables a process to determine with certainty whether multiple logical connections with another machine belong to the same instance of other machine or not. And, it also enables a machine to determine whether a logical connection path exists or not. Another useful feature of this model is that the values of upcounts associated with each logical connection can be put into messages being transmitted over the logical connection. This allows each machine to determine with certainty whether it is to discard a particular message or not, thus eliminating the need for RESETs. It should be noticed that all this mechanism is based upon only two eventcounts per machine and nowhere do we need any machinery other than that provided by the basic eventcount implementation.

Maintaining the upcount requires that each machine have at least some non-volatile memory in which to store its upcount value, which may not be possible for some machines. However, we expect that each machine keeps track of the real time

in some consistent form. The real time as measured in number of time units elapsed since some fixed instant in the past is really an eventcount and will suffice for our purposes. Rather than maintain upcount and downcount, a machine maintains uptime and downtime. When a machine comes up, it updates its uptime by the time of the day. Notice that it is not necessary that all machines use the same unit or reference for their real time clock. The only requirement is that each machine use a consistent method of expressing time as a monotonically increasing number with reference to some fixed instant in the past.

## 5.3 Self-stabilizing Systems with Distributed Control

In a system with distributed control, total state information is ditsributed in various stores connected with individual processes comprising the system. For the sake of robustness, such a system can be constructed such that each process communicates with other processes by recording its state information in its store which is accessible to other processes for observation only [17]. In such systems, property of self-stabilization which states that if the system is in an illegitimate state, it will return to a legitimate state in a finite number of moves - for a more precise definition, see [8] - is quite useful. However, design of even simple systems with these properties is quite difficult. Let us consider a simple system consisting of two processes interested in alternating the use of a common resource. Each process can indicate its state with respect to the use of shared resource by a boolean variable indicating whether or not it is using the resource. We invite the reader to atempt to construct a solution which will bring the system into a legitimate state in a finite number of steps regardless of the starting state. It is difficult. Essentially, the problem is that while a process is taking some action to bring the system into the legitimate state, other process can take some action which

will counter-act first process's action, thus keeping the system in an illegitimate state indefinitely. This is analogous to the critical race problem encountered in switching circuits. We claim that it is easier to solve such problems using eventcounts. We present the following solution to the above problem using one eventcount for each process:

```
process0:
   update(C0, read(C1));
   if mod(C0,2) = 0 then begin
      use resource;
      advance(C0)
   end;
   else await(C1,read(C0)+1);
   goto process0;

process1:
   update(C1, read(C0));
   if mod(C1,2) ≠ 0 then begin
      use resource;
      advance(C1)
   end;
   else await(C0,read(C1)+1);
   goto process1;
```

Processes, numbered 0 and 1, signal changes in their state via eventcounts $C_0$ and $C_1$ respectively. Process number 0 is using the resource when $\underline{mod}(C_0,2)=0$ and process number 1 is using the resource when $\underline{mod}(C_1,2)=1$. The system will eventually move into the legitimate state (only one process using the resource at one time) from an illegitimate state (both processes using the resource simultaneously). In this example, the critical race is avoided because when the system is in an illegitimate state only one process can make the move. In general, it is easier to construct such solutions with eventcounts because each possible state of the system can be uniquely identified. (Eventcount values never repeat). This makes it possible to uniquely identify the "next legitimate" state from each illegitimate state thus making it possible for each process to determine its own move independent of other processes. It is clear that any distributed self-stabilizing system consisting of cyclic processes can be constructed using eventcounts.

Next, we present a solution for a somewhat more complicated self-stabilizing system consisting of several machines arranged in a ring. Each machine communicates only with its neighbors and in the stable state only one machine can be "enabled" at any given time and control passes from a machine to its neighbor in the clockwise direction. Each machine referes to its own eventcount as C and to that of its anticlockwise neighbor as C'. One of the machines has the following program:

```
LOOP:   await(C', read(C));
        "enabled";
        update(C, read(C')+1);
        goto LOOP;
```

and all other machines have the following program:

```
LOOP1:  await(C', read(C)+1);
        "enabled";
        update(C, read(C'));
        goto LOOP1;
```

Dijkstra gave this problem and its solution for a "central daemon"[1] in [8]. Our solution using eventcounts is applicable to truly concurrent machines. Only assumption in the above programs is that updates to eventcounts do eventually get communicated to their neighbors. In systems where this assumption does not hold, a timeout facility can be introduced in each machine such that it will periodically wakeup and examine its neighbor's state.

---

[1] With a central daemon, each machine can assume that while it examines the state of other machines and makes a move, other machines may not be changing the state information.

# Chapter 6

## A Comparison with Some Existing Models

In this chapter we shall compare some common models of synchronization with the eventcount model. In order to facilitate our discussion, we provide a list of desirable and undesirable properties for a synchronization system. For the sake of brevity, each property is labeled with a suitable name. A synchronization system should have the following properties:

1-Distributed systems ($D_s$): As more and more distributed systems come into existence, we must be able to deal with their synchronization problems. Issues of reliability and error recovery become even more important in distributed systems and any synchronization method should be able to deal with such issues.

2-Structuring of Solutions ($S_t$): Does a solution structure capture the structuring of problem.

We shall characterize synchronization systems in terms of the following undesirable properties:

1-Dependency on environment specific properties ($E_p$): dependency on language features, representation of data, or special protocols for accessing data are to be avoided.

2-Unnecessary explicit information flow paths ($I_e$): Any synchronization problem has certain intrinsic information transfer among some processes. If however, its realization in some synchronization system requires additional information flow paths then it is possible that the original security requirements would be violated. In addition, these information flow paths introduce dependencies that

are not intrinsic to the problem and have an adverse effect on the robustness of the solution.

3-Hidden information flow paths ($I_h$): Mutual exclusion coupled with access to a real time clock can be shown to provide a two way communication channel between processes [18]. Thus, if mutual exclusion is not an intrinsic requirement of the problem, it should be avoided especially where security is of great concern.

4-Unnecessary sequentiality ($S_q$): If a synchronization system forces sequentiality and prevents exploitation of parallelism of multi-processors and distributed systems, then the system performance would be affected. This can become serious if the number of processes involved is rather large.

## 6.1 Some existing models of synchronization

The two most common models of synchronization are: (1) the P-V model using semaphores [9] and (2) block and wakeup model [28]. In the P-V model, the approach to solving synchronization problems is to record the state of the system in shared data and allow each process mutually exclusive access to it. In the block-wakeup model, a process can transmit a signal to another process, which will awaken the recipient if already blocked. When the receiving process returns from block, it is expected to examine some shared data and determine the cause of the signal and other relevant information. Mutual exclusion on shared data is not specified but it is necessary for correct operation. To avoid a "critical race" between the transmitter and receiver, a wakeup waiting switch, one per receiving process, is provided and it is required that operations on the switch be mutually excluded in time. The mutual exclusion of operations on the synchronization variables (i.e., semaphores and wakeup waiting switch) is implicit in both models. Thus, shared data and mutual exclusion of operations on shared data (as well as synchronization variables) are inescapable consequences of both the models. A third consequence of the block and wakeup model

is that the solutions depend on identity of the processes involved. A wakeup must be sent to some specific process. In a system of cooperating processes, if some process is replaced by another process, its name must be made known to all other communicating processes. The P-V model also introduces similar process dependencies but in a more subtle way. Even though a semaphore's existence is independent of existence of processes, a signal transmitted through a semaphore (a V operation) can be received by only one process. The solution to this problem is the "private semaphore", i.e. a semaphore for each process on which no other process will ever perform the P operation [9]. Thus a third consequence of both models is process dependency. We mention this here only because it accentuates the need for shared data (other than synchronization variables) for communicating process identities.

There exists a third model of process synchronization based upon message passing [2]. A sender communicates information to the receiver by depositing a message in the receiver's queue from which receiver can read the message at its leisure. Since the message queues are the synchronization variables, this model eliminates the need for shared data. Mutual exclusion in operations on synchronization variables is required. One consequence of this model is that the structure of information being communicated (i.e. contents of messages) is not defined and has problems similar to those introduced by shared data. This model is also heavily dependent upon process identities because each message queue is associated with some process. As pointed out by Lauesen [19], in this model it is not possible to construct semaphores without introducing an administrator process.

There exist two other important models of synchronization: (1) conditional critical regions [1,15], and (2) monitors [2,14]. Both these models are based upon the principal of mutual exclusion of critical regions and utilize shared data for maintaining the state of the system. There exist several proposals to extend the basic concept of semaphores [25,30,32], but all the resulting models are based upon

the principle of mutual exclusion and suffer from the adverse effects of mutual exclusion.

Next, we discuss how the three consequences (shared data, mutual exclusion on synchronization variables and shared data, and dependence on process identities) affect the properties.

## 6.2 Effects of Shared Data

One of the implications of using shared data (other than synchronization variables) for synchronization purposes is that synchronization depends upon some agreed upon representation of data and semantics of operations on the shared data. Thus, programs executing in two different languages may not be able to synchronize their operations. This is especially true for processes located on different knids of machines. Unless processes are given selective access to various parts of the shared data, the system will have the undesirable property of unnecessary information flow paths ($I_e$). One assumption of the shared data approach is that it is possible to give a complete description of the global state of the system at all the times [11].

In the eventcount model, there is no need for "shared data" since synchronization variables themselves provide a convenient method of representing a system's state. Thus, to synchronize their operations, all that two different systems need to do is to provide a common implementation of our synchronization model. Furthermore, operations on eventcounts and sequencers correspond to the information flow paths that are intrinsic to the problem. Therefore, this model does not introduce any unnecessary information flow paths. Thus, while use of shared data has the undesirable properties $E_p$, $I_e$, (and indirectly) $I_h$, and $S_q$, in our model, there is no need for shared data and consequently we can avoid these.

## 6.3 Effects of Mutual Exclusion

It can be shown that mutual exclusion coupled with access to a real time clock provides a hidden communication channel among processes [18]. Thus mutual exclusion on shared data and synchronization variables introduces hidden information paths where none might be necessary. This makes it impossible to construct systems in which some processes must be totally confined. In distributed systems, mutual exclusion must be avoided whenever possible, since time taken to achieve a consensus among processes may be rather long and therefore these models of synchronization can not be applied to distributed systems ($D_s$). Furthermore, unnecessary mutual exclusion implies unnecessary sequentiality ($S_q$). The block and wakeup model provides no simple solution to mutual exclusion problems ($S_{me}$), which can be easily solved in the P-V model.

In our model, it is possible to avoid both kinds of mutual exclusion unless it is an intrinsic requirement of the problem. We have shown methods of implementing operations on eventcount without mutual exclusion. A sequencer is used only when it is an intrinsic requirement of the problem. Since it is not necessary to use shared data other than the synchronization variables, we can also avoid mutual exclusion on shared data. It is precisely because the eventcount model does not rely upon shared data and mutual exclusion that it is applicable to distributed systems.

## 6.4 Effects of process dependence

In the P-V model, a monitoring problem[1] can be solved by using the so called private semaphores [9]. The state of the shared resource is maintained in a shared data base and each process is given mutually exclusive access to it. When in the critical region, a process examines the state variables and determines whether it can proceed or not. If it can proceed, then it performs a V operation on its private semaphore, otherwise it alters the state variables such that when the state, for which it is waiting arrives, some other process will perform the V operation on its private semaphore. Upon exit from the critical section a process performs a P operation on its private semaphore. We might consider this solution satisfactory if the set of processes involved is fixed and small. However, under the dynamic conditions of time sharing systems where old processes must be removed and new ones introduced, this solution becomes quite complex because of the necessity of introducing the names of private semaphores and associated state information into the shared data. Furthermore, as the number of processes in the system grows, contention for access to shared data increases thus delaying the real time progress of processes. This is perhaps not a serious problem for single processor systems, but in multi-processor systems this can become a potential bottleneck especially if the number of processes is large. By and large, these remarks apply to the block and wakeup model also.

It is clear that solution of a simple problem, which intrinsically requires nothing more than pure, non-interfering observation of the activities of a system is impossible in the existing models unless some means of communicating information from observers to the system are introduced, thus making the system dependent upon the

---

[1] As discussed elsewhere in this paper, a monitoring problem is one in which one or more processes are interested in observing the changes in the state of some shared entity. In principle, it should be possible to solve such a problem without resorting to mutual exclusion.

behavior of observers. This communication requires shared data and possibly mutual exclusion and suffers from all the bad effects of the two discussed above.

In the eventcount model, signals transmitted via eventcounts have a permanent existence and are available to all observer processes because observation of events does not alter eventcount values. This implies that from signaller's view point signals are transmitted to some abstract entity which may consist of one, two, or any number of processes. How these signals are utilized by the observers (i.e., one signal per process or one signal to all processes) can be negotiated among the observer processes themselves and is of no concern to the signaller processes. This eliminates the need for signalling processes to know the names of the observer processes.

The independence of signaller processes from observers has profound influence on structuring and proving correctness synchronization processes. It permits us to examine each process's behaviour in relation with only those eventcounts that it interacts with without being concerned with the details of other processes.

# Chapter 7

## A Message Communication Protocol

The producer-consumer problem discussed before, extends to the network environment where the "producer" and "consumer" processes are located in remote machines and communicate over message transmission channel rather than a shared buffer [31, 22]. Since the communication channels are often unreliable, complex protocols have been developed to achieve reliable communication [3]. In this chapter we shall demonstrate, by an example, that the problem of constructing such protocols is really a synchronization problem no different in nature than those encountered in shared memory systems and that it can be easily described and solved using the eventcount synchronization model.

### 7.1 A Communication Protocol for a Reliable Channel

Consider two remote processes: (1) a message generator located in the "sender" machine and (2) a mesage consumer located in a "destination" machine. Assuming that there exists an unreliable comunication channel and a reliable eventcount synchronization system between the two machines, we are interested in constructing a reliable message transmission path from the message generator process to the mesage consumer process. We are interested in two kinds of unreliability introduced by the message transmission channel: (1) some messages may get lost in the transmission path and (2) messages may not arrive in the order of transmission. In addition, there needs to be flow control mechanism that would prevent the sender from overflooding the buffer space allocated by the destination.

For the sender machine, we define the following processes in addition to the message generator: (a) a space allocator which allocates buffer space for use by the message generator, (b) a transmitter which reads messages from the buffer space and transmits them to the destination machine, and (c) a space freer process which frees the buffer space occupied by the messages that have already been transmitted. Similarly for the receiver machine, there is a process to allocate buffer space, a receiver process to receive messages from the sender and deposit them into the buffer space, and a buffer freer process to free the buffer space occupied by messages that have already been consumed by the mesage consumer.

Next, we define a few eventcounts to be used for synchronization among these processes. Notice that each eventcount is manipulated by only one process whose name is given in parentheses following the description of each eventcount. These eventcounts are as follows:

- G: Each message in the potentially infinite message stream is assigned a unique number (starting with 1). At any given time, its value indicates the number of messages generated. (Message Generator)
- T: number of messages transmitted to the destination. (transmitter)
- $A_s$: total amount of buffer space allocated to the message generator in the unit of messages. (Space Allocator in sender)
- $F_s$: total amount of buffer space from which messages have been transmitted and which has been freed. (Space Freer in sender)
- C: number of messages consumed by the mesage consumer process. (Message Consumer)
- R: total number of messages received by the receiver process. In the message stream it points to the message received last. (Receiver)
- $A_d$: total amount of buffer space allocated to the receiver. (Space Allocator in destination)
- $F_d$: total amount of buffer space from which messages have been consumed and which has been freed. (Space Freer in destination)

For correct operation, the following conditions[1] must hold:

Sender machine: $A_s \geq G \geq T \geq F_s$
Destination machine: $A_d \geq R \geq C \geq F_d$
Transmission control: $R \leq T \leq A_d$

---

[1] Derivation of these relationships is inspired by Pouzin's work [26].

The above relationship for the sender machine states that the message generator will never generate more messages than the space allocated ($A_s \geq G$), the transmitter will never transmit more messages than generated ($G \geq T$), and the space freer will never free the space occupied by the messages not yet transmitted ($T \geq F_s$). Similarly, the requirements for the destination machine are that the receiver will never receive more messages than the space allocated for them ($A_d \geq R$), the consumer will never consume more messages than received ($R \geq C$), and the space freer will never free space occupied by messages that have not been consumed ($C \geq F_d$). The first of these requirements ($A_d \geq R$) is enforced by the transmission control requirement that the transmitter will never transmit more messages than the space allocated by the destination ($T \leq A_d$) and that the receiver will never receive more messages than transmitted ($R \leq T$).

To build a system that satisfies above requirements is straightforward and simple in the eventcount model. For the sender machine, message generator observes $A_d$, transmitter observes G, and space freer observes T. We have an essentially similar situation for the destination machine. The transmission control requirement, $T \leq A_d$, can be met by the transmitter observing the eventcount $A_d$ and making sure that it never transmits more than the allocated space. The requirement $R \leq T$ is imposed by the physics of the situation, the receiver can not receive more messages than transmitted. In this system, each process works completely asynchronously and has a minimum amount of interaction with other processes which is dictated by the requirements of the system.

The overall structure of the system is depicted in figure 1. We show the essential synchronization relationship by putting each process's eventcount(s) next to it. Arrows going out from an eventcount point to those processes that observe it. The arrows going into eventcounts come from manipulator processes.

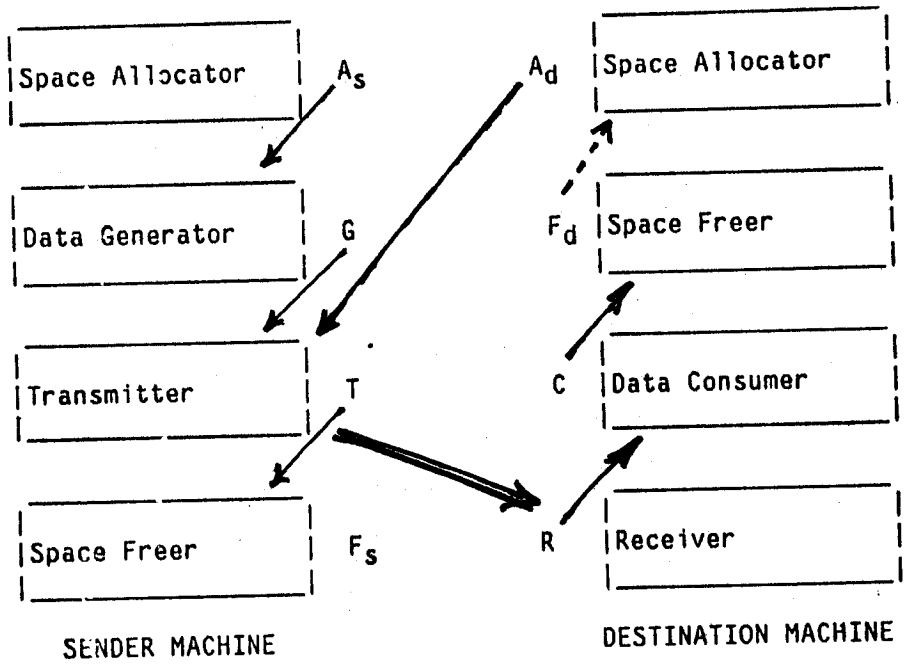



**Figure 1 - Communication Protocol for a reliable channel**

## 7.2 A Recoverable Protocol for an Unreliable Channel

The system works satisfactorily for a reliable transmission channel. Extension to the unreliable case is simple and straightforward. To enable the receiver to detect loss of messages we need some method of identifying messages. This can be readily accomplished by depositing the sequence number of each message in it. We introduce a new process called "retransmitter" in the sender machine and the following eventcounts:

$E_d$: It points to the rightedge of the retransmission window (assuming that stream proceeds from left to right). (Receiver)
$E_s$: This clock indicates the right edge of the stream which has been retransmitted. (Retransmitter)

Whenever the receiver detects a hole in the message stream, it updates $E_d$ to the right edge of the hole. If however, the receiver does not have reassembly capability, it can discard all messages past the hole and set $E_d$ to the highest sequence number encountered or to the value of T. The retransmitter observes eventcount $E_d$ and initiates retransmission whenever $E_d > E_s$. The retransmission,

however, need not start at $E_s$, rather it starts at R. Thus, retransmitter observes two eventcounts. This method takes care of both kinds of unreliability in the communication channel mentioned previously. To embellish things further, we can add a timeout facility in the retransmitter that will cause it to retransmit, if messages in a previous retransmission have not already been received successfully. It is now essential that the buffer space occupied by the messages in sender be freed only after the messages have been succesfully received ($F_d$<R). This requirement can be easily met by the sender's buffer freeing process, which now observes the eventcount R rather than T. The overall structure of this system is depicted in figure 2. By way of examples two possible programs for transmitter and retransmitter processes are

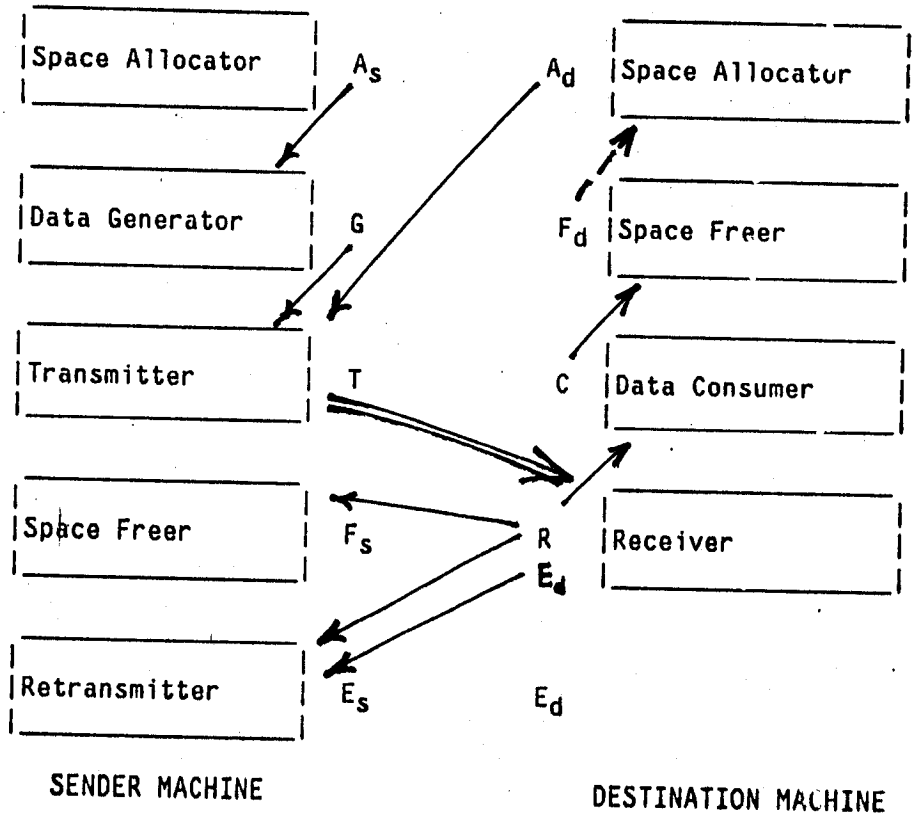



**Figure 2 - Communication Protocol for an unreliable channel**

also provided.

```
process transmitter;

integer length, t;

t := 1;

while true do
begin
     await(G, t);
     await(A_d, t);
     send message(t);
     advance(T);
     t:=t+1
end
end


process: retransmitter;

comments "∞" is meant to represent some very large value of the real time clock not
         expected to be reached during the life of this process. Similarly interval
         is assigned some meaningful value in the units of the real time clock
         "time". G, T, A_d, E_d, E_s, and R are eventcounts. "send message(s)" sends
         the message with the sequence number s to the destination machine.;

integer leftedge, rightedge, length, alarm;

     alarm := ∞;
     interval := 50;

while true do begin
     await(E_d, read(E_s), time, alarm);
     leftedge := read(R)+1;
     rightedge := read(E_d);
     if rightedge ≥ leftedge
     then begin
             while leftedge ≤ rightedge do
             begin
                  send message(leftedge);
                  advance(E_s);
                  leftedge := leftedge +1
             end;
             alarm := read(time) + interval
          end
     else alarm := ∞;
     end
end
```

## 7.3 Information Flow Paths and Security Issues

Figures 1 and 2 provide a graphic illustration of information transfer paths introduced into the system by the synchronization mechanism. Assuming that the message transmission channel from the sender to the destination machine does not provide a reverse communication path, the only information path from the destination machine to the sender machine is that between the space allocator (destination) and the transmitter process via eventcount $A_d$. Under normal situation, due to buffer space limitations, there would exist in each machine a communication path from space freer process to the space allocator. This path completes an indirect information transfer path from the mesage consumer process to the message generator process. If the security requirements of the system were to demand so, this information path could be eliminated by eliminating the path from $A_d$ to the transmitter process. This of course has the effect of eliminating all paths from the destination machine to the sender machine. One implication of eliminating this path is that there must be some predetermined buffer allocation strategy that can be followed by the transmitter process. If a machine could guarantee to be able to supply buffer at a rate faster than the maximum rate of message transmissions permitted by the channel, then one can assume an effective allocation of infinity. Another method of eliminating this information path would be to eliminate some path within a machine; for example, the path from eventcount C to the space freer in the destination machine. However, among processes sharing resources (as they do if they are located on one machine) there exist hidden or implicit information paths which are virtually impossible to eliminate. [18]. If the security requirements of the system require that even implicit paths be eliminated from the mesage consumer to the message generator then nothing short of eliminating all information paths from destination to the sender machine would satisfy the security needs. It is clear from figure 2 that a reliable channel can not be constructed from an unreliable channel without some form of

communication from the destination to the sender machine. The only alternative is that for a given channel, reliability of transmission can be improved to some desired degree by transmitting multiple copies of each message.

## 7.4 Some Comments

Besides illustrating to the reader that a complex network protocol problem of a distributed system is really a synchronization problem and that the eventcount model can be used to solve it, we hope to have demonstrated the ease of application of this model. The power of the eventcount model results from the fact that it permits decomposition of a complex system into asynchronously executing, more or less independent components that need to know about the total system state no more than absolute minimum required for their proper operation. Furhermore, the size of each component of the system can be kept small so that it can be easily understood and proved to work correctly.

# Chapter 8

## Implementation Issues

In this chapter we discuss a number of ways to implement eventcounts. We will first consider how to do so in a system in which all of the processors share memory. After this, we will discuss some of the issues in implementing eventcounts on a distributed system, where memory is not shared. In the course of our discussion, we will present the idea of a "single-manipulator" eventcount, which will be somewhat simpler to implement than a fully general multiple-manipulator eventcount. We will show how this "single-manipulator" eventcount can be used very elegantly in the construction of a general eventcount.

For the distributed system (and shared-memory, multi-word) implementations of eventcounts, it is useful to partition eventcounts into two classes: (1) eventcounts on which no read operation will be performed and (2) eventcounts on which read operation can be performed. The former are called asynchronous eventcounts and the latter are called time-synchronized (or synchronous) eventcounts. As we will see soon, asynchronous eventcounts are easier to implement than synchronous eventcounts.

Finally, we will discuss how to implement a sequencer in both the shared-memory case, and the distributed system case.

## 8.1 Shared Memory Eventcount Implementation

Let us assume that we have a number of processes sharing memory. Memory will be composed of words which are long enough to hold any reasonable eventcount values (say 64 bits). The simplest possible implementation in this case requires that the process be able to execute a machine instruction (aos) which adds one to the contents of a storage location in one memory cycle. The arbitration ability of the memory interface will prevent two simultaneous additions from causing incorrect results.

If this is the case, then we can implement the advance operation as an aos instruction to the memory cell which holds the contents of the eventcount. The observe operation can be implemented as load instruction from the memory cell which holds the eventcount value. The read operation is identical to observe. Finally, the await operation can be implemented as a loop which repeatedly obtains the observe value of the eventcount, and compares that value with the desired value.

If we wish to avoid busy-waiting, we can modify the await operation so that if the comparison fails the first time, the process executing the await is blocked. The address of the eventcount, and the value waited for will be stored with the blocked process. The advance operation will have to be modified to search the blocked process queue for any processes waiting for the advanced eventcount's new value. However, key to implementation of await operation is the observe operation, whether or not it is made available to the users. In the remainder of this chapter we will not discuss implementation of await.

The implementation that we have just described is perhaps most likely to occur in most systems. However, this implementation does not apply to machines which do not have an "aos" instruction and neither does it apply to distributed systems. Another problem in a computer system might be that a single memory word may not be large enough to hold an eventcount. In the remainder of this section we discuss

alternate metnods of implementing eventcounts for these situations. A reader not interested in the details may skip it without loss of continuity.

### 8.1.1 Single- vs Multiple-Manipulator Eventcounts

A rather surprising result of our definition of eventcounts, which we state here without proof, is that the sum of several eventcounts is also an eventcount. Therefore, we can construct a multiple-manipulator eventcount from the sum of several single-manipulator eventcounts. This is a powerful result and key to implementation of multi-word and/or distributed system eventcounts.

Let there be m manipulator processes S1, S2, ..., Sm for an eventcount E. To construct E, we associate a single-manipulator eventcount E[i] with each manipulator Si. Now, the operation advance(E) for process Si, represented by $\text{advance}_i$, can be defined as follows:

```
procedure advance_i(E);
begin
   advance(E[i]);
end;
```

Operations observe(E) and read(E) can be constructed as follows:

```
procedure observe(E);
begin
   return(observe(E[1])+observe(E[2])+ ...+observe(E[m]));
end;

procedure read(E);
begin
   return(read(E[1])+read(E[2])+...+read(E[n]));
end;
```

The operation await(E,v) can be implemented using observe. To eliminate busy waiting, it is necessary to use the generalized form of await waiting on each of the component eventcounts simultaneously.

The above construction for a multiple-manipulator eventcount is independent of any particular representation of single-manipulator eventcounts. It holds for shared memory as well as distributed systems, and for single word as well as multiple word eventcounts. it holds for asynchronous as well as synchronous eventcounts. Furthermore, it preserves the information flow properties for eventcounts. As a consequence, all we need do is show how to implement single-manipulator eventcounts.

In the remainder of this section we shall restrict our discussion to implementation of single manipulator eventcounts only. First, we describe the implementation of multiple-word asynchronous and synchronous eventcounts for shared memory and then for the distributed systems.

### 8.1.2 Shared-memory, multi-word, asynchronous eventcount

If the memory word is not long enough to hold an eventcount, we can implement this type of eventcount using several memory words. We provide a brief description of the algorithm. A n+1 word long eventcount $E_{n+1}$[1] can be constructed from an n word long eventcount $E_n$ and one word of memory M. The advance operation on $E_{n+1}$ consists of adding 1 to the contents of M, and then, if there is a carry over from the addition to M, an advance operation on $E_n$. To perform the observe operation, each process must now maintain the last observed value of $E_{n+1}$, called $E_o$. Notice that $E_o$ is unique to each observer process. To obtain the value of observe($E_{n+1}$), a process first obtains observe($E_n$) (called x) and then reads the content of M (called y). The value of observe($E_{n+1}$) is given by the following:

$$\max( (x*2^l+y), E_o)$$

where l is the length of M in number of bits.

---

[1] Remember that we are dealing only with shared-memory, asynchronous, single-manipulator eventcounts.

Even though we have defined the eventcount recursively, in practice it need not be so. An n word long eventcount can be implemented as a sequence of n words, where the advance operation is performed from the right to the left (i.e., least significant word to the most significant word) and the observe operation is done from left to right (i.e., most significant word to the least significant word). We remind the reader that we are talking about asynchronous eventcounts only, that is the read operation is not defined.

### 8.1.3 Shared-memory, multi-word, synchronous eventcounts

We use the same technique as used above to describe the multi-word eventcount of this type, i.e., we shall define advance, observe, and read operations recursively. A n+1 word long synchronous, single manipulator, shared memory eventcount can be constructed from two n word long eventcounts $E_n$ and $E_n'$ of similar type and two memory words M and M'. The operation advance($E_{n+1}$) consists of first adding 1 to M, and then if there is a carry over, advancing $E_n$. This procedure is repeated with $E_n'$ and M'. The operation read($E_{n+1}$) can be best described by the following algorithm:

```
procedure read(E(n+1));
begin
   y:=read(En');
   m:= M';
   x:=read(En);
   if x > y
      then return(x*2^1)
      else return(y*2^1+m);
end;
```

The reason why this read operation gives correct result is as follows: if the manipulator is advancing the pair ($E_n'$, M') when the reader is obtaining y and m, then the manipulator must have already advanced the pair ($E_n$, M). If x is not greater than y then this advance operation could not have caused an overflow on M (or M'), hence ($E_n'$, Mn') returns correct value. If on the other hand x > y then the

eventcount value ($E_{n+1}$) must have crossed the value $x*2_1$ since read was initiated and hence this is the correct value. The observe operation can be performed on any of the pair ($E_n$,M) or ($E_n$', M') and is similar to that for the asynchronous eventcount.

A 64 bit long synchronous eventcount on a 16 bit word machine will require 22 words of shared memory.

## 8.2 Eventcounts in Distributed Systems

### 8.2.1 Asynchronous Eventcounts on Distributed Systems

We are now ready to discuss the implementation of eventcounts on a distributed system. The basic problem here is that changes to an eventcount (via advance) take time to propagate down communication lines to other systems. We have two options -- we can implement all eventcounts on one system, and require that other systems request that system to do operations, or we can distribute the eventcount values, so that each system maintains a local copy. We have chosen the latter alternative.

We have already shown how to add together single-manipulator eventcounts to obtain multi-manipulator eventcounts, so we will concentrate on single-manipulator eventcounts. First, let's see how to construct a single-manipulator asynchronous eventcount.

This is rather simply done, by using two shared-memory single-manipulator eventcounts, one local to the manipulator system, and the other local to the observer system. In addition, we require two processes, one on each system. These things are connected as in figure 3. Manipulations to the eventcount are accomplished by manipulating $E_m$. Process $M_t$ waits for changes to $E_m$ using await, and then reads the current value and transmits it over the communications line. Process $O_r$ waits for

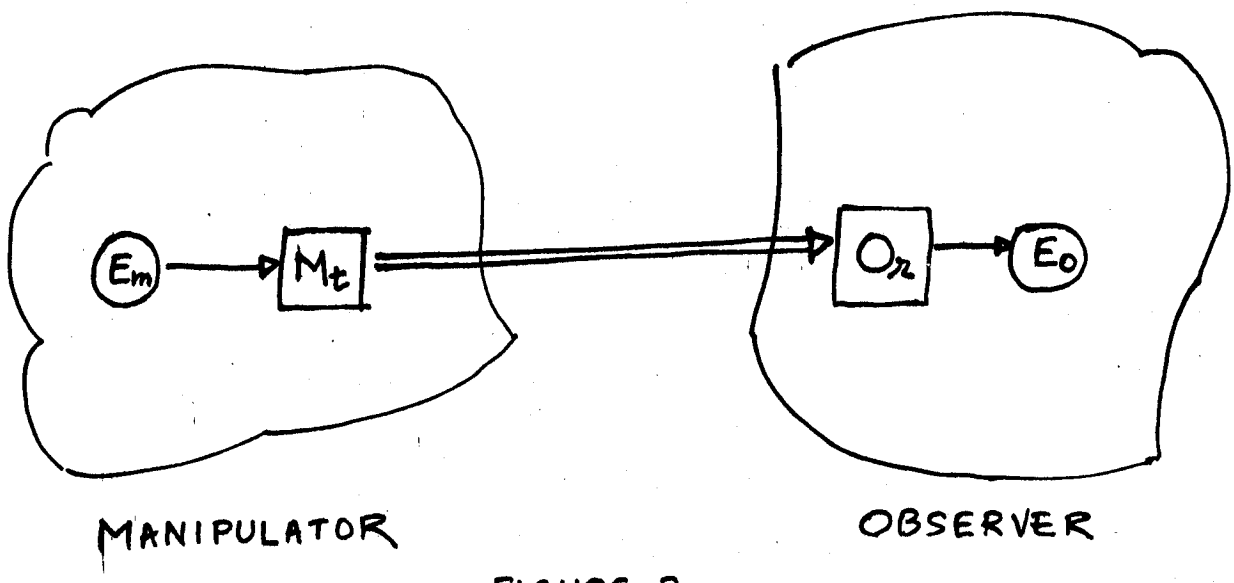



FIGURE-3-

messages over the communications line, and then updates the observer systems local copy, $E_o$, to the received value, v, using the following loop:

```
while read(Eo) < v do advance(Eo)
```

Processes on the observer system can then use the eventcount $E_o$ in await and observe operations as if it were the real eventcount. $E_o$ has the required property that it follows closely behind the eventcount $E_m$.

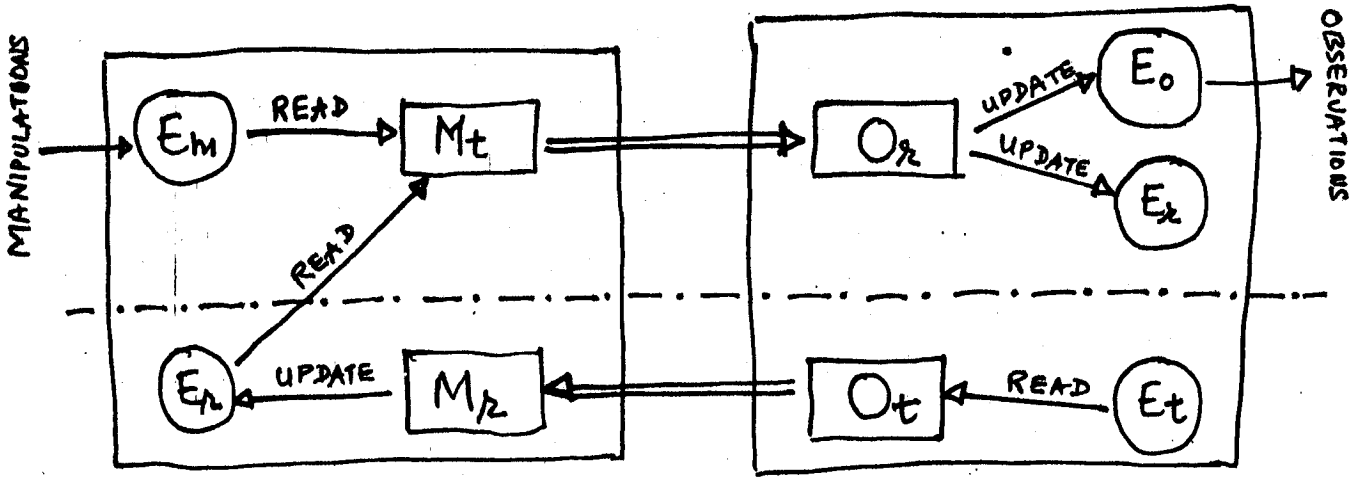
A feature of this implementation is that it will work even if the messages on the communications line are received in a different order than they were transmitted. In fact, if we make process $M_t$ transmit extra messages at fixed time intervals, we can make the eventcount more robust with respect to loss of messages. This is a rather useful result, which will be true also of the single-manipulator synchronous eventcount. We can achieve this robustness because of the non-decreasing nature of the eventcount value.

## 8.2.2 Synchronous Eventcounts in Distributed Systems

In order to implement a synchronous eventcount, we need only the capability to do a read operation on the distributed eventcount just described. What we must do to implement the read operation is to guarantee that the process on the observer system doing the read does not get a value from $E_o$ which is less than the value of $E_m$ was when the read was started. We can do this by a time-stamp. Along with every message sent by the process $M_t$, there will be a value which represents the time when $M_t$ read the value of $E_m$. Process $O_r$ will be changed to update $E_o$ and then a new eventcount, $E_r$. from the transmitted value and its time stamp, respectively. Then when a process on the observer system does a read, it will wait until $E_r$ exceeds the time when the read was begun.

In order to make the time-stamps work correctly, it is necessary to make sure that the "time" obtained on the observer system when it starts a read never lags behind the "time" obtained by the process $M_t$ when it transmits a message. This can be ensured by implementing the "time" as an asynchronous single-manipulator

FIGURE #4




eventcount, where the roles of the observer and manipulator are reversed. Figure 4 depicts the construction of a working single-manipulator synchronous distributed

eventcount. The portion of the figure below the dotted line creates timestamps, and the portion above transmits values with timestamps.

In order to guarantee that a read will complete, we change $M_t$ to transmit a message whenever either $E_r$ or $E_m$ are changed. A read operation is just the following sequence.

```
t:=read(E_t)+1
advance(E_t)
await(E_r, t)
v:=read(E_0)
```

## 8.3 Implementation of Sequencers

In a shared memory system, a sequencer can be implemented as a single word in shared memory and ticket operation consists of reading the value of the word and increasing it by one, all in an indivisible operation mutually excluded from other operations on the same word. If the "aos" instruction used previously were augmented to save the value of the memory word before increasing it, then this single instruction can implement shared memory, single word eventcounts as well as sequencers.

If, however, there is no such instruction or if a single word is not large enough to hold all possible values, then some form of mutual exclusion is necessary to perform the ticket operation. The implicit communication channel provided by mutual exclusion can not, however, cause security violations since there exist explicit communication channel in the ticket operation.

In distributed systems, implementation of sequencers is more difficult. One method would be to associate a process with each sequencer whose job is to receive requests for ticket operations from all the processes and send them the ticket values. The requests for ticket operation and replies can be accomplished much in

the same way as read operation for synchronized eventcounts.

Another method of implenting sequencers is to circulate a message among all processors, such that two processors never have possession of the message simultaneously. Then, whenever a processor has the message, it can execute a ticket operation, and then pass the message along. This is not particularly robust, since if the message is lost, the fact is not detected, and can result in ticket operations never terminating.

While we have not found a perfectly robust solution to this problem, the problem is analogous to Dijkstra's problem of Self-Stabilizing Systems [8] discussed earlier.

# Chapter 9

## Axioms and Verification Techniques

In this chapter we present an axiomatic description of eventcounts and sequencers and demonstrate how to apply these axioms to verify synchronization properties of solutions realized in the eventcount model by way of an example.

### 9.1 An Axiomatic Description of the Eventcount Model

We would like to be able to define the effect of primitive operations in our model precisely, in order to be able to make and prove statements about the correct operation of our programs. In particular, we would like our model to be able to define the effect of eventcount primitives executed on the same eventcount concurrently in time.

In order to do this we introduce a notion of time into the model. This time is purely a formal variable, which serves to specify the relative timings of operations executed in a set of concurrent processes. Any program operation A will have a starting time, $t_s(A)$, and a completion time, $t_c(A)$, such that $t_s<t_c$. We can then say that two primitive operations A and B may execute simultaneously if it is possible that $t_s(A)\leq t_s(B)<t_c(A)$ or $t_s(B)\leq t_s(A)<t_c(B)$. In a sequential process, operations are ordered so that if A is followed by B, $t_c(A)\leq t_s(B)$.

We would like to be able to show the interaction of any possible timings of a set of processes, and to constrain the possible timings. We will do this by means of a technique using hidden functions of time, by which we mean a set of functions whose values vary over time, but which cannot be directly observed by programs in the

system. By this technique, we may hope to describe timing properties which cannot be dealt with by traditional program-proving techniques involving only the state of the whole system as observable by a program. In particular, in a distributed system, or one in which certain security restrictions are enforced, programs cannot in general observe the state of the whole system at a particular time. Also, programs cannot observe the state of the whole system while they are executing a synchronization primitive, so traditional program-proving techniques cannot make a statement about the behavior of the system while synchronization primitives are executing. However, we may nonetheless care about the performance of the system as seen from an omniscient external viewpoint, in order to be able to discuss such ideas as "fair scheduling" and so forth.[1] Time is also an important system value when the system has its boundaries outside the computer system itself, as is the case in most process-control systems or air-traffic-control systems, for example.

### 9.1.1 Axioms for Eventcounts

For an eventcount E, we define two hidden functions S(E,t) and C(E,t) such that at time t, S(E,t) and C(E,t) equal the number of advance operations started and completed on E at or before time t. They are both non-decreasing functions of time (see fig. 5). Finally, we define the function O(E,p,t) to be the maximum value v returned by observe or read operations or input to await operations on E in process P whose $t_c$ is less than or equal to t. We use this function to guarantee that actually observed values of eventcounts are non-decreasing from the point of view of a particular process.

---

[1] It should be noted that not all possible external behaviors can be generated by programs using our synchronization primitives, but that was not our intention. We do however want to be able to discuss the limitations of our primitives from this viewpoint.

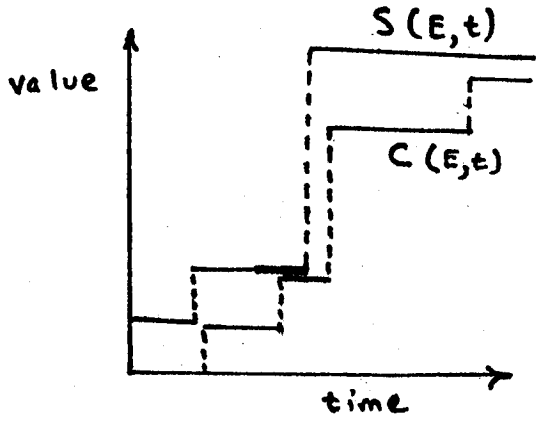



Figure 5

We will now give the defining constraints for the values returned by observe and read as well as the effect of executing await, by the following six axioms. In these axioms, we say that $B(E,p,t_1,t_2)$ is the value that would be returned by an observe operation X executed in process p, such that $t_s(X)=t_1$ and $t_c(X)=t_2$. Similarly, $R(E,p,t_1,t_2)$ is the value that would be returned by a read operation X in process p, such that $t_s(X)=t_1$ and $t_c(X)=t_2$. The axioms are as follows:

1. All operations except await are guaranteed to terminate in finite time. [termination]

2. For an operation await(E,v), if there is a t such that $v \le S(E,t)$, then the await operation will terminate at some $t_c$ such that $v \le S(E,t_c)$. [termination of await]

3. For all t,p $(\exists t_1)(\forall t_2 \ge t_1)$ $C(E,t) \le B(E,p,t_1,t_2)$. By this statement, we formalize the idea that eventually the effect of advance operations will be observed by observe operations in other processes. [eventual propagation of changes to observe]

4. $0 \le B(E,p,t_1,t_2) \le S(E,t_2)$. [bounds on observe]

5. $C(E,t_1) \le R(E,p,t_1,t_2) \le S(E,t_2)$. [bounds on read]

6. $O(E,p,t) \le R(E,p,t,t')$,
   $O(E,p,t) \le B(E,p,t,t')$.
   These statements just mean that the observed values of the eventcount in a process don't decrease, that is, a read operation followed by an observe operation in the same process will return values in a non-decreasing order. [consistent observation]

Fig. 6 shows how axiom 5 defines the result of a read operation, which starts at time $t_1$ and ends at time $t_2$. If we consider the read operation to have happened at some time between these two times, the value returned must be in the shaded region.

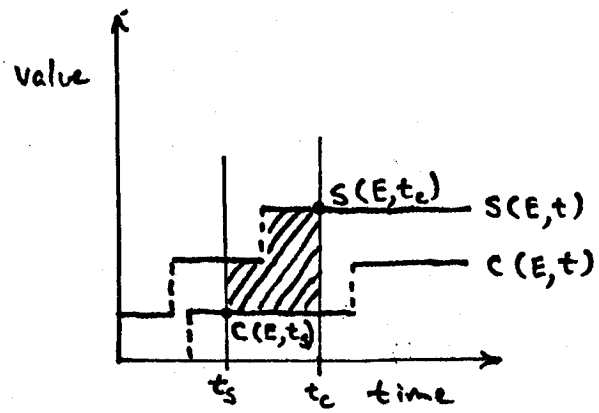



Figure 6

We guarantee no more than this; therefore, $C(E,t_1) \leq R(E,p,t_1,t_2) \leq S(E,t_2)$, which is just axiom 4. The other axioms result from similar reasoning.

### 9.1.2 Axioms for Sequencers

For a sequencer S, we define two hidden functions $S_T(S,t)$ and $C_T(S,t)$ such that at time t their values equal to the number of ticket operations started and completed, respectively, on S at or before time t. Also let $T(S,p,t_1, t_2)$ be the value returned by ticket operation on the sequencer S in process p, which started at time $t_1$ and completed at time $t_2$. The two axioms for the sequencers are as follows:

1. $C_T(S,t_1) \leq T(S,p,t_1,t_2) < C_T(S,t_2)$. [bounds on ticket values]
2. Let X(S,p,t) be the set of values returned by ticket operations in process p on S which have completed on or before t. Then for all p,q such that $p \neq q$, $X(S,p,t) \cap X(S,q,t) = \phi$. [ticket uniqueness]

## 9.2 Verification of synchronization processes

In this section, we show how the axioms related to our model can be used to verify whether or not certain required synchronization conditions hold for a given system of parallel programs. Techniques for proving correctness of parallel programs are still in their infancy and it is not our intention to present a formal methodology for proving correctness of programs using eventcounts.

Typically verification of assertions would proceed as follows. The assertions for a problem are stated as relationships between problem variables that must hold for the duration of some activity. Each activity corresponds to execution of some sequence of code in some program. Typically, problem variables are counts of occurrences of events; for example, number of writes. A problem variable could also be the number of processes engaged in a particular activity at any given time. Some of these relations could be invariant over the life of the system under consideration. The first step in our verification method is to establish correspondence between the problem variables and synchronization variables of the program which, in general, are eventcounts and sequencers. This would enable us to restate relations in terms of program variables. To accomplish this, we examine the program structure and derive C and S functions for each eventcount in terms of execution sequences of activities associated with each event. This provides us with relationships between problem variables and synchronization variables. The next step is to chose some relationship and attempt to verify it by examining execution of the program containing the code sequence associated with the assertion.

One of the most important properties of an eventcount is that its value (or its C and S functions) never decreases in time. Thus, if we place an assertion of the type $S(E,t) > x$, or $S(E,t) \geq x$, at some time in the execution of some program, where $x$ is an integer value, then the truth of this assertion can not be altered by

execution of any other program. If x is a local variable, then the assertion holds until the program changes value of x, but parallel execution of other programs still can not cause this assertion not to hold. This result has considerable impact on simplifying verification of assertions. If values of x are related to values of some other eventcount then we can derive a relationship between the eventcounts merely by eliminating x.

Since proofs of correctness for even short, sequential programs can be rather long, we chose a rather simple and short problem, which is already familiar to us, namely the producer, consumer problem, as our test case. It has the additional advantage that its synchronization conditions have already been discussed in the literature [12]. Let us consider a single-producer, single-consumer system communicating through a finite length buffer buf[0:N-1]. We present the following solution for this problem:

```
procedure deposit(d);
     begin
     D1:  in:=read(IN);
     D2:  await(OUT, in-N+1);
     D3:  buf[mod(in,N)]:=d;
     D4:  advance(IN)
     end

procedure accept(r);
     begin
     A1:  out:=read(OUT);
     A2:  await(IN, out+1);
     A3:  r:=buf[mod(out,N)];
     A4:  advance(OUT)
     end
```

A synchronization condition for this system is that the following relation must be true at all times:

(1) $\quad 0 \leq N_d(t) - N_r(t) \leq N$

where $N_d(t)$ and $N_r(t)$ are the number of messages deposited and received respectively. The left inequality of this relation states that the number of messages received can

not exceed the number of messages deposited and the right inequality states that number of messages in the buffer (= number deposited - number received) can not exceed the length of the buffer.

The first step in our verification method is to establish correspondence between the variables describing the relationship (1) and the synchronization variables of the programs (i.e. eventcounts IN and OUT). Since we are assuming that there is only one producer process, executions of procedure deposit can never overlap with each other and we can treat it as a sequential program. A similar argument applies to procedure receive. It is easy to see from the above programs that the eventcount IN is advanced by 1 each time a message is deposited and OUT is advanced by 1 each time a message is removed. Therefore, at any time t, the following holds:

$$C(IN,t) \leq N_d(t) = S(IN,t) \text{ and } C(OUT,t) \leq N_r(t) = S(OUT,t)$$

and (1) can be restated as follows:

(2) $\quad 0 \leq S(IN,t) - S(OUT,t) \leq N$

Next, we shall prove the left inequality of this relation:

(3) $\quad 0 \leq S(IN,t) - S(OUT,t)$ or $S(IN,t) \geq S(OUT,t)$ for $t \geq 0$

We shall use the following notation in the remainder of this section: $An_i$ refers to the $i$th execution of the statement $An$. $t_s(An_i)$ is the time at which $i$th execution of statement $An$ begins and $t_c(An_i)$ is the time of completion of that statement. We focus our attention on the procedure "accept". Let us assume that at the time "accept" is invoked for the $i$th time, the relation (3) holds. That is:

(4) $\quad S(IN,t) \geq S(OUT,t)$ at $t = t_s(A1_i)$

Since A1 and A2 do not alter the value of OUT, it follows that:

(5) $\quad S(IN,t) \geq S(OUT,t)$ for $t_s(A1_i) \leq t_c(A2_i)$

From the definition of S and C functions and the sequential nature of successive executions of "accept", we can deduce the following:

(6) $C(OUT,t) = S(OUT,t) = OUT_i$ for $t_c(A4_i) > t \geq t_s(A1_i)$
(7) $S(OUT,t) = OUT_i + 1$ for $t_s(A4_i) \leq t \leq t_c(A1_i)$

where $OUT_i$ denotes $S(OUT, t_s(A1_i))$. From statement A1 and the axiom relating to read, we have:

$$C(OUT,t_s(A1_i)) \leq out \leq S(OUT,t_c(A1_i))$$

which when combined with (6) reduces to:

(8) $out = OUT_i$

From A2 and the axiom relating to await:

$S(IN,t_c(A2_i)) \geq out+1$
or $S(IN,t_c(A2_i)) > out$
or $S(IN,t_c(A2_i)) > OUT_i$ from (8)
or $S(IN, t) > OUT_i$ for $t \geq t_c(A2_i)$

When combined with (6) and (7), this yields:

(9) $S(IN,t) > S(OUT,t)$ for $t_c(A2_i) \leq t \leq t_s(A4_i)$
(10) $S(IN,t) \geq S(OUT,t)$ for $t_s(A4_i) \leq t \leq t_c(A4_i)$

Combining (5), (9) and (10) will yield:

(11) $S(IN,t) \geq S(OUT,t)$ for $t_s(A1_i) \leq t \leq t_c(A4_i)$[1]

This shows that assertion $S(IN,t) \geq S(OUT,t)$ is invariant under execution of "accept". Since it is initially true, we conclude that (3) holds. Proof of the right hand inequality of relation (2) follows almost identically. Notice the symmetry between the procedures "deposit" and "accept". The only difference is in the await statement where -N+1 is added to the integer value instead of 1.

It can be easily shown that the system will not dead lock. A dead lock occurs

---

[1] Note that $A > B$ implies $A \geq B$.

if at any time t both producer and consumer have started execution of awaits which shall never terminate. From (6), (8), and axioms relating to await, we can easily show that deadlock at time t implies:

(12) $S(IN,t) \leq S(OUT,t)$

and similarly for the producer deadlock implies:

(13) $S(OUT,t) \leq S(IN,t)-N$

Inequalities (12) and (13) reduce to:

$$S(IN,t) \leq S(IN,t) - N \quad \text{or} \quad 0 \leq -N$$

which leads to a contradiction for positive values of N. Therefore for buffer lengths greater than zero, there can be no deadlock. It is to be expected that if there is no communication buffer (N=0), the system will be deadlocked.

## References


[1] Brinch Hansen, P. Structured Multiprogramming. Comm. ACM 15, 7 (July 1972), 574-578

[2] Brinch Hansen, P. The Nucleus of a Multiprogramming System. Comm. ACM 13, 4 (April 1970), 238-241

[3] Cerf, V. G., Kahn, R. E. A Protocol for Packet Network Intercommunication. IEEE Trans. Commun. 22, 5 (May 1974), 637-648.

[4] Courtois, P.J., Heymans, F., Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (Oct 1971), 667-668

[5] Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control. Comm. ACM 8, 9 (Sept 1965), 569

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

[7] Dijkstra, E. W. Co-operating Sequential Processes. In Prgoramming Languages (Ed. F. Genuys), Academic Press, New York, 1968.

[8] Dijkstra, E. W. Self-stabilizing Systems in Spite of Distributed Control. Comm. ACM 17, 8 (Aug 1974), 453-455

[9] Dijkstra, E. W. The Structure of the "THE"-Multiprogramming System. Comm. ACM 11, 5 (May 1968), 341-346

[10] Easton, W. B. Process Synchronization without Long-term Interlocks. Proceedings of the third symposium on operating systems principles, Operating Systems Review 6, 1-2, (Jun 1972)

[11] Greif, I. Semantics of Communicating Parallel Processes. MAC-TR-154, MIT Proj. MAC, Cambridge, Mass., 1975

[12] Habermann, A. N. Synchronization of Communicating Processes. Comm. ACM 15, 3 (March 1972), 171-176.

[13] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. Comm. ACM 12, 10 (Oct 1969), 576-583

[14] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. Comm. ACM 17, 10 (Oct 1974), 549-557

[15] Hoare, C.A.R. Towards a theory of parallel programming. In Operating Systems Techniques, Academic Press, New York 1972.

[16] Johnson, P.R. and Thomas, R.H. The Maintenance of Duplicate Databases. (ARPA) Network Working Group, RFC-677, (Jan 1975).

[17] Lamport, L. A. New Solution of Dijkstra's Concurrent Programming Problem. Comm. ACM 17, 8 (Aug 1974), 453-455

[18] Lampson, B. W. A Note on the Confinement Problem. Comm. ACM 16, 10 (Oct 1973), 613-615.

[19] Lauesen, S. A Large Semaphore Based Operating System. Comm. ACM 18, 7 (July 1975), 377-389.

[20] Levitt, K. N. The application of program-proving techniques to the verification of synchronization processes. FJCC 1972, 33-47.

[21] Lipton, R. J. On Synchronization Primitive Systems. Ph.D. Thesis, Carnegie-Mellon University, Dept. of Computer Science, (Jun 1973)

[22] McKenzie, A. Host/Host Protocol for the ARPA Network. in Current Network Protocols, Network Information Center, Menlo Park, Calif., NIC 8246, (Jan 1972).

[23] Merlin, P. M., Farber, D. J. Recoverability of Communication Protocols - Implications of a Theoretical Study. IBM Research Document, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

[24] Parnas, D.L. On a Solution to the Cigarette Somker's Problem (without conditional statements). Comm. ACM 18, 3 (March 1975), 181-183.

[25] Patil, S. S. Limitations and capabilities of Dijkstra's semaphore Primitives for coordination among processes. MIT Proj. MAC, Computational Structures Group Memo 57, (Feb 1971)

[26] Pouzin, L. Network Architectures and Components. 1st European Workshop on Computer Networks, Arles (France), April-May 1973.

[27] Rappaport, R. L. Implementing Multi-Process Primitives in a Multiplexed Computer System, MAC-TR-55, MIT Proj. MAC, Cambridge, Mass., 1968

[28] Saltzer, J. H. Traffic Control in a Multiplexed Computer System. MAC-TR-30, MIT Proj. MAC, Cambridge, Mass., 1966.

[29] Sorenson,P. G. Interprocess Communication in Real-time Systems. Fourth Symposium on Operating System Principles, ACM Operating Systems Review 7, 4 (Oct 1973).

[30] Vantilborgh, H., Lamsweerde, A. On an Extension of Dijkstra's Semaphore Primitives. Information Processing Letters 1, (1972), 181-186

[31] Walden, D. C. A System for Interprocess Communication in a Resource Sharing Computer Network. Comm. ACM 15, 4 (April 1972), 221-230

[32] Wodon, P. Still another tool for synchronizing cooperating processes. Department of Computer Science Report, Carnegie-Mellon U., Pittsburgh, Pa., (1972)