Acta Informatica 7, 15—34 (1976)
© by Springer-Verlag 1976

# The Synchronization of Independent Processes

Leslie Lamport

Received January 22, 1975

*Summary*. This paper considers the problem of programming a multiple process system so that it continues to operate despite the failure of individual processes. A powerful synchronizing primitive is defined, and it is used to solve some sample problems. An algorithm is then given which implements this primitive under very weak assumptions about the nature of interprocess communication, and a careful informal proof of its correctness is given.

## Introduction

As computers developed the ability to execute several *I/O* operations concurrently, sophisticated operating systems were introduced to take advantage of this. It was discovered that an operating system can be more easily designed and implemented if it is considered to be a collection of separate asynchronous processes [3]. Simple primitives can be used to synchronize the processes, while the complexities of the actual hardware interrupt system are hidden in the implementation of these primitives. Once the primitives have been implemented, everything else can be done in a reasonably hardware independent fashion.

Recently, there has been considerable interest in systems composed of several independent computers [10]. The synchronizing procedures proposed for single computer operating systems have relied upon a central synchronizing process, either explicitly with monitors [7] or secretaries [5], or implicitly by relying upon a central process to maintain semaphores [3] or other shared variables. Although satisfactory for a single computer system, such approaches may be undesirable for a multiple computer system because they imply reliance upon the single hardware component which contains this central process.

This paper considers the problem of process synchronization without a central synchronizing process. A powerful synchronizing primitive is defined which generalizes the "conditional critical region" statement described in [1]. This primitive provides simple solutions to many synchronization problems. An algorithm to implement it is given which allows the system to continue normal operation despite the failure of any individual process, and a careful informal proof of its correctness is provided. It is shown that a reasonably efficient implementation of the algorithm is possible.

The main purpose of the paper is to show that synchronization problems can be solved without relying upon any central hardware component. This is done by implementing the synchronizing primitive under minimal assumptions about the nature of interprocess communication. The problem of language design is not considered, and we do not mean to imply that this primitive should be used in a general programming language. However, we hope that future languages for multiprocess systems make use of the ideas presented here.

This paper generalizes the results of a previous paper [8] which solved a specific synchronization problem: the mutual exclusion problem. This problem was also solved independently by Dijkstra [6] with no central synchronizing process, under weaker assumptions about interprocess communication. His solution recovers automatically from intermittent process errors, but unlike our solution, his does not work if any process fails completely. A similar error recovery feature can be added to our solution by making certain redundant computations, but space does not permit us to discuss this.

## 1. The Problem

### *An Example*

To motivate the discussion, we begin by describing a sample system. It is not meant to be a general model, but just a simple example. The system consists of a collection of independent computers and disk drives. Each computer has its own memory and central processor, and operates completely asynchronously with the other computers. It is connected to some subset of the other computers by two-way communication channels. A channel can transmit interrupt signals and data words.

Each computer is also connected by *I/O* channels to some subset of the disk drives. An *I/O* channel can handle a single operation to read or write a continuous block of data in the usual manner. A disk drive can perform only one operation at a time, but it will automatically sequence concurrent *I/O* requests from different computers. Note that a computer need not be connected to all other computers and disk drives. It is connected to another computer or to a disk drive by at most one channel.

Each computer maintains its own private data files on the disks. Since a disk can handle concurrent operations from different computers, to implement private files it is only necessary to ensure that space on the disks is properly allocated to individual computers. Efficient utilization of disk space requires that allocation be done dynamically. We therefore have the following *allocation problem:* each computer must be able to execute the operations of acquiring and releasing disk tracks. This means that we must synchronize concurrent *acquire* operations by two different computers so that the same track is not allocated to both of them. If there are not enough free disk tracks, then a computer executing an *acquire* operation will have to wait until the necessary number of tracks are released by other computers. Note that the "dining philosophers" problem of [5] is a special case of the allocation problem, since a "fork" can be represented by a disk drive connected to two computers.

Several computers may also want to share a common data file. This causes no problem if they just want to read from it. However, no other operations to the file may be allowed while one process is updating it. This is called the *readers/writers problem,* and was introduced in [2]. Since reading or updating the file may require several individual *I/O* operations, the disk hardware does not solve the problem for us. We must devise an algorithm to ensure that no other operation to the file can take place while a computer is updating it.

It is easy to solve these two problems if we let the necessary scheduling be done by a central computer which communicates with all other computers. However, we rule out such a solution because the failure of that computer would halt the entire system. We must find solutions which allow the system to continue despite the failure of any single component.

## *The Abstract System*

We now formulate the problem in a more general, abstract fashion. We assume a system of independent communicating processes. We will not bother to define precisely what a "process" is, but will use its customary informal definition as an algorithm executed by a processor. We assume a fixed collection of processes, numbered from 1 through $N$. Each process has its own local memory. Interprocess communication is achieved by allowing one process to read from another process' memory. We assume that reading or writing a single memory word is an indivisible operation[1]. A word might consist of just one bit. Later on, to permit more efficient implementations we will introduce a mechanism by which processes can send interrupt signals to one another. However, such a mechanism is not essential. Note that any data item must reside in some process' memory.

We allow processes to fail at any time by halting. Since we want one process to be able to continue despite the failure of other processes, it must be able to read from a failed process' memory. If a failed process' memory could retain the contents it had at the time of failure, then other processes could be blocked indefinitely. For example, if a writer failed while its memory contents indicated that it was writing, then it would block all other readers and writers. When a process fails, we allow it to enter a *malfunctioning period* during which reading its memory may yield any arbitrary values—even fluctuating, incorrect ones. However, we must assume that the process then reaches a *quiescent state* in which each word of memory assumes some default value. For convenience, a program variable is assumed to be encoded so that its default value is *zero* if it is a number, *false* if it is a boolean, and $\phi$ if it is a set. A failed process may be restarted at some predefined place in its algorithm.

In our sample system, a process is a program running on a single computer. It reads a word from another process' memory by sending a message to the appropriate computer and waiting for a reply. A computer is presumed to have some error detection mechanism to shut itself down if it fails. If a computer does not receive a response to a *read memory* request within some fixed length of time, then it assumes that the other computer has failed and uses the default value. In this example, we are assuming that there is no communication failure. The possibility of such a failure presents additional difficulties and will be discussed in a future paper.

## *The Synchronization Primitive*

Our goal is to show that a class of synchronization problems for our abstract system can be solved so that the system continues to function correctly despite

1 We could actually use a weaker assumption, but it would complicate the proofs.

the failure of one or more processes. This will be done by defining a powerful synchronizing primitive which permits simple solutions to many synchronization problems, including the ones described above, and then showing that this primitive can be implemented by our system of abstract processes.

The primitive we will use is a generalization of the "conditional critical region" statement described in [1]. It is expressed by a statement of the following form:

**region** *mode* **when** *condition* **do** *critical section* **od**

where *mode* assumes values from some arbitrary finite set $M$, and *condition* is some boolean function of the contents of processes' memories. We assume a symmetric function *conflict*: $M \times M \to \{true, false\}$. If two processes are concurrently executing **region** operations with values of *mode* equal to $mode_1$ and $mode_2$ respectively, then these processes are said to *conflict* if $conflict(mode_1, mode_2) = true$.

When a process executes a **region** statement, it first waits until it can enter its critical section. In order to enter the critical section, the following two conditions must hold:

C1. No conflicting process is in its critical section.

C2. The **when** condition equals *true*.

Entering the critical section is considered to be an instantaneous event, so C1 means that two conflicting processes cannot both be in their critical sections at the same time.

Sensible use of the **region** statement requires that some restrictions be made on it. We will make the following restrictions.

R1. The *mode* value cannot change during execution of the **region** statement.

R2. If the **when** condition depends upon a word in another process' memory, then that process can modify the word only within the critical section of a conflicting **region** statement.

R3. The critical section cannot contain another **region** statement.

Note that R2 will prevent race conditions between operations that test and modify the **when** condition. For convenience, we will also assume that a process' *mode* value can be encoded in a single memory word. (This is trivially true if a process always executes **region** statements with the same *mode*.)

The **region** statement permits a simple solution to the readers/writers problem, in which reading and writing a specific file are done with the following operations:

```
READ
region (file, "reader") when true do read od
WRITE
region (file, "writer") when true do write od
```

where $conflict((file_1, m_1), (file_2, m_2))$ is true iff (if and only if )$file_1 = file_2$ and $m_1$ or $m_2$ equals "*writer*". Rule C1 then guarantees that no other process can access a file while a process is writing it.

To illustrate the use of the **region** statement to solve the allocation problem, we consider the simple case in which all processes have access to all disk drives, and a process releases all tracks before acquiring any new ones. We let $T$ denote the set of all disk tracks, *acquired* $[i]$ be the set of all tracks currently being used by process $i$, and $|S|$ denote the cardinality of a set $S$. Below are the operations which process $i$ executes to acquire $n$ tracks and to release all acquired tracks. Note that *acquired* $[i]$ is stored in the memory of process $i$, $T$ is a constant, and *conflict* (*"allocate"*, *"allocate"*) $=$ *true*.

```
ACQUIRE
region "allocate" when $\sum_{j=1}^{N} |acquired[j]| + n \leq |T|$ do
  $acquired[i] := any\ n\ elements\ of\ \left(T - \bigcup_j acquired[j]\right)$ od
RELEASE
region "allocate" when true do
  $acquired[i] := \phi$ od
```

Conditions C1 and C2 imply that the required number of tracks are free when the *acquire* operation's critical section is executed, provided that no process is malfunctioning. Our assumption about the default values of variables implies that a failed process automatically releases its tracks. No damage is done if *acquired* $[j]$ assumes arbitrary values while process $j$ is malfunctioning. However, process $i$'s *acquire* operation may not be able to choose its tracks until $j$ reaches its quiescent state and *acquired* $[j]$ assumes its default value $\phi$. (Of course, serious errors may occur if a malfunctioning process writes onto tracks not allocated to it.) The failed process may be restarted anywhere outside its **region** statements.

The reader should have no trouble generalizing this solution to the case in which a process has access only to some subset of the disks. Two processes will conflict only if they can both access the same disk. Allowing the incremental acquisition of tracks requires a more complicated **when** condition in order to avoid deadlock. We refer the interested reader to the "bankers algorithm" of [4].

## *Scheduling*

In order to complete our specification of the **region** statement, we must say more about when waiting processes are to enter their critical sections. Often, the most difficult aspect of a synchronization problem is the scheduling of different processes' actions. To perform this scheduling, we now introduce a mechanism which allows one to specify the order in which conflicting processes should enter their critical sections.

When a process begins execution of the **region** statement, it enters at the tail end of a queue of processes waiting to enter their critical sections. For each pair of distinct processes $i$, $j$, we assume an arbitrary boolean function *should·precede* $(i, j)$ whose value may depend upon the contents of processes' memories, the *mode* value of waiting processes, and possibly other quantities external to our abstractions—e.g., the length of time processes have been in the waiting queue. We add the following condition which must be met for process $i$ to leave the waiting queue and enter its critical section.

C3. For every conflicting process $j$ on the waiting queue:

(a) if $j$ is before $i$ on the queue, then *should·precede* $(i, j) = true$.

(b) if $j$ is after $i$ on the queue, then *should·precede* $(j, i) = false$.

We complete our specification with the following requirement.

C4. Any process $i$ for which conditions C1–C3 remain satisfied will eventually enter its critical section.

Note that if *should·precede* $(i, j)$ is always false for all $i$ and $j$, then conflicting processes enter their critical sections in the same order in which they began executing their **region** statements. The *should·precede* function specifies when processes should enter out of turn. For example, in the readers/writers solution let us define *should·precede* $(i, j)$ to be true iff process $i$ wants to write and process $j$ wants to read the same file. Then a waiting writer prevents any waiting process from reading the file, so this gives writers a higher priority than readers. Conflicting writers enter their critical sections on a first come, first served bases.

As another example, in the allocation solution let us define *should·precede* $(i, j)$ to be true iff process $i$'s **when** conditon is true and process $j$'s **when** condition is false. Then other operations are not blocked by a process which is waiting to acquire more tracks than are currently available. In particular, any *release* operation is eventually executed.

This modified first-come-first-served scheduling procedure should solve the scheduling aspect of most synchronization problems. Note that it is the user's responsibility to make sure that his *should·precede* function eliminates the possibility of deadlock.

We will not specify any syntax for defining the *conflict* and *should·precede* functions. We are not concerned with language design, and are not proposing that this **region** statement be used in any real programming language.

## 2. The Implementation

### *Correctness*

We require that our implementation of the **region** statement function correctly regardless of the relative execution speeds of the different processes. When designing such a multiprocess algorithm, a careful proof of correctness is necessary if subtle, time-dependent errors are to be avoided. However, we cannot give a careful proof without defining more precisely what "correctness" means. The subtlety of the problem is indicated by the following example. Suppose process 1 maintains program variables $a$ and $b$ in its memory, with $a=1$ and $b=0$ initially. It then executes the following sequence of operations: $a := 0$; $b := 1$. Meanwhile, suppose process 2 evaluates the function $a*b$ by fist reading the value of $a$ then the value of $b$, obtaining the value 1 for both variables. Process 2 thus decides that $a*b$ equals 1, despite the fact that $a*b$ always remains equal to 0. As this example indicates, it can be impossible for a process to determine the correct value of a function which depends upon values stored in other processes' memories. Hence, it might be impossible for any implementation to satisfy conditions C1–C4.

Let us consider more closely the concept of correctness in our abstract system. At any instant, the state of a process is specified by the contents of its memory and the value of its "program counter", the latter specifying where it is in the execution of its algorithm. Executing the process produces a sequence of events. The only events we will consider are those which test or modify a word of (some process') memory. Such an event will usually change the process' state by changing the program counter, and may also change the contents of its memory. For convenience, we assume that a process never stops generating events. (Halting can be effected by a loop.)

Since we have assumed that reading or writing a single word of memory is an indivisible event, it can be shown that there is a total temporal ordering of all the events in the system. We denote this ordering by $\rightarrow$, so $e \rightarrow f$ means that event $e$ preceded event $f$. The choice of the relation $\rightarrow$ is somewhat arbitrary. If $e$ and $f$ are events in different processes which cannot causally effect one another because of the time needed by a signal to propagate from one process to the other, then we may arbitrarily define the ordering so that either $e \rightarrow f$ or $f \rightarrow e$.

The state of the system at any instant consists of the contents of all processes' memories and the values of their program counters. An *execution* of the system consists of some valid initial state and a sequence of process events. This sequence must be consistent with the initial state and the processes' algorithms. The system state is defined between any two consecutive events of the execution. Formalizing these concepts is a straightforward but tiresome task which we will not attempt.

A correctness property of an algorithm is expressed as a theorem about executions. The theorem can involve conditions on possible system states and sequences of events, and it must be true for all possible executions of the system. We have to formulate properties C1–C4 as theorems of this type which must be true for an implementation of the **region** primitive.

The statement "process $i$ is in its critical section" is an assertion about the values of process $i$'s program counter. Entering the critical section is an event in the process which occurs before it executes any operation in the critical section. For convenience, we define a failed or malfunctioning process to be outside its **region** statement, so it does not conflict with any other process. This is purely a matter of convention to simplify the statement of the correctness properties. There is obviously no way to prevent a malfunctioning process from executing its critical section at any time, and the value of a quiescent process' program counter is irrelevant.

Whether or not two processes conflict is a function of their program counters and, if they are both currently executing **region** statements, of their mode values. We can thus restate conditions C1 and C2 as follows.

D1. Two conflicting processes cannot both be in their critical sections at the same time.

D2. When a process enters the critical section, its **when** condition must be true.

Before restating C3, we have to define the waiting queue more precisely. There must be some part of a process' algorithm which represents the waiting queue, so whether a process is in the queue is a function of its program counter. We assume that a process which does not fail can leave the waiting queue only by entering its critical section.

One process must be able to decide if another is ahead of it in the queue. Hence, for each $i$ and $j$ there must be some function of the contents of process memories which specifies if process $i$ precedes process $j$ in the queue. We let $\#(i) < \#(j)$ denote this function, so its value is *true* if $i$ is before $j$ in the waiting queue.

The boolean function $\#(i) < \#(j)$ must be specified for any particular implementation. We thus need some condition to guarantee that this function has the desired properties. We would like to require that $\#(i) < \#(j)$ be true if process $i$ entered the waiting queue before process $j$ did. However, we also want to require that a process should not have to wait to enter the waiting queue, and one can show that both requirements cannot in general be satisfied.

Let the *doorway* denote the section of the algorithm from the beginning of the **region** statement until the process enters the waiting queue. We will make the following requirement.

D0. (a) There is a fixed bound on the number of process events in the execution of the doorway.

(b) For any conflicting processes $i$ and $j$ on the waiting queue: if $i$ entered the queue before $j$ entered the doorway, then $\#(i) < \#(j)$.

Condition D0 is a reasonable requirement to make on the implementation of the waiting queue. We cannot expect to determine which process entered the queue first unless they entered at "measurably different" times. The time needed to execute the doorway, which by D0 (a) is bounded, determines what "measurably different" means. Note that D0 (b) mentions only conflicting processes. This is because the relative position on the queue of non-conflicting processes is irrelevant.

We can now consider condition C3. It is possible for an implementation to satisfy D1 and D2 only because of restrictions R1 and R2. E.g., R2 means that a waiting process will be able to correctly evaluate its **when** condition while no conflicting process is in its critical section. Without some similar restriction on the *should·precede* function, it is impossible to satisfy C3. Indeed, the example given above of evaluating $a*b$ shows that it may be impossible for a process ever to obtain the correct value of *should·precede* $(i, j)$.

Let us consider the two parts of C3 separately. In our readers/writers solution, C3 (a) states that a writer in the waiting queue will enter its critical section before any conflicting process which is behind it in the queue. This is an important condition, because it ensures that every *write* operation is eventually executed. Condition C3 (b) states that a reader will not enter its critical section if there is a conflicting writer behind it in the queue. However, it doesn't matter if a reader enters its critical section even though a conflicting writer has just entered the waiting queue. In fact, there is no way to prevent such a possibility, since entering the critical section is effected by a single event. We will therefore not make C3 (b) a formal requirement, but will merely expect that an implementation "try" to satisfy it by having process $i$ evaluate *should·precede* $(j, i)$ before entering its critical section. We modify C3 (a) to obtain the following requirement.

D3. Let $i$ and $j$ be conflicting processes on the waiting queue, and assume that the value of *should·precede* $(i, j)$ does not depend upon any quantity which can change while $i$ and $j$ are in the doorway or on the waiting queue. Then process $i$ cannot enter its critical section if $\#(j) < \#(i)$ and *should·precede* $(i, j) = false$.

Condition C4 is different from C1–C3 because it asserts that something *must* happen, whereas the others assert that something must *not* happen. To state it precisely, we need some more definitions. A *time interval* is a finite sequence of consecutive events in an execution. It represents the operation of the system between the first and last of those events. For any positive integer $P$, a *$P$-interval* is a time interval containing at least $P$ events from each process. Our assumption that processes never stop generating events means that every sufficiently long time interval is a $P$-interval.

A function is said to be *strongly constant* on a time interval if it has a constant value during that interval, and every process which evaluates it during the interval obtains that value. In the example given above, the function $a*b$ has the constant value 0 during a time interval, but it is not strongly constant because a process evaluated it an obtained the value 1.

We can now restate C4 more precisely as follows.[2]

D4. There exist integers $M$ and $P$ such that at least one of the following conditions must be false at some point during any time interval consisting of $M$ consecutive $P$-intervals.

(a) Process $i$ is in the waiting queue.

(b) No process is malfunctioning.

(c) No process which conflicts with process $i$ is in its critical section.

(d) Process $i$'s **when** condition is true.

(e) For every process $j$ on the waiting queue which conflicts with process $i$:

  (i) *should·precede* $(i, j)$ and *should·precede* $(j, i)$ are strongly constant.

  (ii) if $\#(i) < \#(j)$ then *should·precede* $(j, i) = false$.

  (iii) if $\#(j) < \#(i)$ then *should·precede* $(i, j) = true$.

Conditions (c)–(e) imply that D1–D3 permit process $i$ to enter its critical section. If these conditions hold, then we want $i$ to go ahead and enter its critical section. Condition D4 asserts that if conditions (b)–(e) hold for a sufficiently long time, then (a) must become false, so process $i$ must leave the waiting queue and enter its critical section. Note that (b) becomes false when a process fails, and remains false until it reaches its quiescent state. Hence, we allow a sequence of failures and restarts by other processes to keep process $i$ indefinitely from entering its critical section.

Finally, we observe that if two processes never conflict with one another, then there is no reason why they should have to communicate with each other. For example, in the general allocation problem, two computers need never exchange messages if they cannot both access the same disk. For each process $i$, let $Con(i)$ denote the set of all other processes which can conflict with it. Then we require that process $i$ read only from its own memory and the memories of processes in $Con(i)$. Note that $j \in Con(i)$ iff $i \in Con(j)$.

2 Note that two consecutive $P$-intervals from a single $2P$-interval, but the converse is not true.

## *Waiting*

Synchronization requires that one process be able to wait for another to complete an operation. This waiting will be expressed with the following statement:

```
**wait until** *condition*
```

where *condition* is some boolean function of process variables. It is logically equivalent to the following loop:

```
label: if condition = false then goto label fi.
```

This loop implements the **wait until** statement with busy waiting. Such an implementation is inefficient because it means that a physical processor is kept idling. For example, a computer waiting to acquire disk tracks would like to perform other tasks instead of just executing this waiting loop. The synchronization of asynchronous processes always requires busy waiting. However, modern computers use interrupts to allow other operations to be performed while waiting. (The busy waiting occupies that part of the machine cycle which tests if an interrupt bit has been set.)

The interrupt hardware of a computer can be used to implement **sleep** and **awake** operations, defined as follows. Assume a special type of boolean variable called an *alarm*. An alarm may be set true by any of several different processes, using the operation

```
awake (alarm).
```

It may be tested or set false only by the process to which it belongs. It is tested by the operation

```
sleep (alarm)
```

which is equivalent to the statement

```
wait until alarm.
```

Since *alarm* cannot be reset by any other process, this is easily implemented with interrupts.

The general **wait until** *condition* operation can be implemented by assigning an alarm variable to *condition*. Any process which performs an operation that might set *condition* true must also execute a subsequent **awake**(*alarm*). The **wait until** statement is then implemented as follows.

```
label: sleep (alarm);
       alarm := false;
       if condition = false then goto label.
```

Note that a delay is allowed between the execution of successive statements.

This method of implementing waiting may seem somewhat inefficient because a condition is always tested after a process is awakened. However, it is foolproof because no error can occur if (a) two separate **awake** operations occur before *alarm* is reset, thus awakening the process only once, or (b) a single **awake** operation sets *alarm* again after it has been reset by the newly awakened process, thus awakening the process twice. No physical implementation of truly asynchronous processes seems capable of eliminating both of these possibilities.

The only requirement for this implementation of the **wait until** statement is that a process always execute an **awake** operation when it changes a shared variable to a value which might make a **wait until** condition true. (This applies to the initialization, so *alarm* must initially be true if *condition* is true.) The failure of a process might also make a condition true. We must therefore assume that a process failure generates the appropriate **awake** operations after it reaches its quiescent state. In practise, a process might awaken itself at regular intervals with a clock interrupt in order to test the condition, so a failed process need not actually awaken any other process[3].

This discussion shows that interrupts can be used to eliminate most of the busy waiting from a **wait until** operation. Elimination of additional unnecessary testing is a matter of program optimization. We cannot make any general statements about it, since it will depend upon the details of the particular system's hardware.

An obvious generalization of **wait until** is the statement

**parallel wait** $(condition_1, label_1; \ldots; condition_n, label_n)$;

which is equivalent to the following waiting loop:

```
label: if $condition_1$ then goto $label_1$ fi;
                 ⋮
       if $condition_n$ then goto $label_n$ fi;
       goto label.
```

It can be shown that any busy waiting loop can be eliminated by using **parallel wait** operations. Implementation of **parallel wait** by interrupts should be obvious and will not be discussed.

## *A Simple Algorithm*

We first present an implementation of the **region** statement for the special case in which the **when** condition is always true and *should·precede* $(i, j)$ is always false for all $i$ and $j$. Thus, a process enters its critical section when no conflicting process is either in its critical section or ahead of it on the waiting queue.

The algorithm is a simple extension of the one described in [8]. Before a process enters the waiting queue, it chooses a number greater than that of any other potentially conflicting process in the queue. The processes in the waiting queue are ordered by the numbers they have chosen, the one with the lowest number being at the head of the queue. If two processes $i$ and $j$ choose the same number, then $i$ is before $j$ iff $i < j$.

The shared variables are: a boolean array *choosing* $[1:N]$; an array *number* $[1:N]$ of non-negative integers; and an array *mode* $[1:N]$ of mode values. The $i$-th element of each array is in the memory of process $i$. The element *number* $[i]$ may be stored in several individual memory words, but *mode* $[i]$ must be stored in a single word. All variables are assumed to be initialized to *zero* or *false*, except *mode* $[i]$ which may have any initial value.

3 In many cases, the values of $M$ and $P$ in D4 and the execution speeds of the processes can be used to determine how long a process should wait before awakening itself.

Let $\#(i)$ denote the ordered pair $(number[i], i)$. The relation $<$ on ordered pairs of non-negative integers is defined to be the usual lexicographical ordering, except that zero is taken to represent an infinitely large integer. In other words, $\#(i) < \#(j)$ is true iff either

(i) $0 \neq number[i] < number[j]$, or

(ii) $0 = number[j] < number[i]$, or

(iii) $number[i] = number[j]$ and $i < j$.

This defines a total ordering of the $N$ elements $\#(1), \ldots, \#(N)$. Note that the boolean function $\#(i) < \#(j)$ is always defined, but its value defines the order of $i$ and $j$ on the waiting queue only when they are both on the queue.

We introduce a new type of **for** statement of the form

**for all** $j \in S$ **do** ... **od**

where $S$ is a set of integers. It is similar to the usual **for** loop in that the **do** clause is executed once for each value of $j$. However, the values of $j$ used are the elements of $S$, and the executions for the different values can be done in any order.

Below is the algorithm to implement the statement

**region** $mode{\cdot}value_i$ **when true do** *critical section*$_i$ **od**

in process $i$. For the sake of brevity, we write $conflict(i, j)$ as an abbreviation for $conflict(mode[i], mode[j])$. The symbol $\sim$ denotes negation[4].

```
begin integer $j$;
  doorway
    $mode[i] := mode{\cdot}value_i$;
    $choosing[i] :=$ true;
    $number[i] :=$ any integer $> maximum\{number[j] : j \in Con(i)\}$;
    $choosing[i] :=$ false;
  waiting queue
    for all $j \in Con(i)$ do
        wait until $\sim choosing[j]$ or $\sim conflict(i, j)$;
        wait until $\#(i) < \#(j)$ or $\sim conflict(i, j)$ od;
  critical section$_i$;
  $number[i] := 0$
end
```

*Correctness of the Simple Algorithm*

Correctness properties D0–D3 for this solution are deduced from the following two assertions. Their proofs are essentially the same as those of the corresponding

4 Where the program does not indicate the order of execution of memory references—for example, in evaluating the *maximum* function—the order is arbitrary. Recovery from transient errors, in the spirit of [6], can be accomplished by recomputing $mode[i]$ and making sure that $number[i] > 0$ in the waiting loops. A similar modification works for the more general algorithm described later.

assertions in [8], and they are omitted. To conform to the notation of [8], we define the *bakery* to consist of the waiting queue and the critical section.

*Assertion 1.* If processes $i$ and $j$ are in the bakery, $j \in Con(i)$, and $i$ entered the bakery before $j$ entered the doorway, then $\#(i) < \#(j)$.

*Assertion 2.* If process $i$ is in its critical section, process $j$ is in the bakery, and $j \in Con(i)$, then $\#(i) < \#(j)$ *or* $\sim conflict(i, j)$.

Condition D0 (a) is evident, and D0 (b) is implied by Assertion 1. Assertion 2 implies D1, since $\#(i) < \#(j)$ and $\#(j) < \#(i)$ cannot both be true. Condition D2 is trivially true for this special case. Condition D3 is implied by Assertion 2, since the truth value of $\#(j) < \#(i)$ does not change when $i$ enters its critical section.

We now prove D4. To do this, we will show that if (b)–(e) remain true while process $i$ is in the waiting queue, then $i$ must complete its **for all** loop and enter its critical section within $M$ $P$-intervals, for some $M$ and $P$. Let the *epilogue* be the part of the **region** statement's algorithm which follows the critical section. Let us choose $P$ large enough so that the execution of the doorway or of the epilogue takes at most $P$ events. We also choose $P$ large enough so that process $i$ will complete a single iteration of its **for all** loop in one $P$-interval if its **wait until** conditions are strongly constant and equal to *true* during that interval.

Assume that process $i$ is executing the $j$-th iteration of its **for all** loop. Conditions (c) and (e) imply that $conflict(i, j) =$ **false**, or $\#(i) < \#(j)$, or process $j$ is not in the bakery. To avoid extra terminology, we assume that process $j$ enters the **region** statement when it sets the value of *mode* [$j$]. Then exactly one of the following five conditions must be true.

(1) $conflict(i, j) = true$ and $j$ is executing the epilogue.
(2) $conflict(i, j) = true$ and $j$ is outside the **region** statement.
(3) $conflict(i, j) = false$.
(4) $conflict(i, j) = true$ and $j$ is in the doorway.
(5) $conflict(i, j) = true$ and $j$ is in the waiting queue and $\#(i) < \#(j)$.

While $i$ is in the bakery, each of these conditions can become false only when a higher numbered condition becomes true. (In particular, (5) cannot become false once it becomes true.) Conditions (1) and (4) must become false within one $P$-interval. If any of the other three conditions remains true for one $P$-interval, then process $i$ must complete the $j$-th iteration of its **for all** loop during that interval[5]. This shows that process $i$ must complete this iteration of its **for all** loop within six $P$-intervals. Hence, it must enter its critical section within $6 * |Con(i)|$ $P$-intervals. This completes the proof of D4.

## *Bounding number* [$i$]

One difficulty with this solution is that the values of *number* [$i$] could become arbitrarily large. For $N = 2$, a simple modification allows the non-zero values of

5 We are using the assumption that *mode* [$j$] is stored in a single memory word, since this implies that if $conflict(i, j)$ is constant then it is strongly constant. Without this assumption, it would be possible for a slowly executing process $i$ to remain indefinitely in its waiting queue while a fast process executes a sequence of nonconflicting **region** statements with differing mode values.

*number* [$i$] to be chosen from the set $\{1, 2, 3\}$. We merely define the ordering $<$ on this set by $1 < 2$, $2 < 3$, $3 < 1$, and our algorithm remains correct. (The reader can check that the above correctness proof is still valid.)

For $N > 2$, no such simple modification works. Hence, the general solution is formally correct only for processes with infinitely long memory words. However, finding a practical bound for *number* [$i$] is easy if we can be sure that each value of *number* [$i$] is chosen to be at most one greater than a previously chosen value of *number* [$j$], for some $j$. For example, if processes can enter the doorway at the rate of at most one per microsecond, then after a century of operation the value of *number* [$i$] would remain less than $2^{52}$. Since the algorithm is correct even though reading or writing *number* [$i$] may require several separate events, we can use several memory words to provide a sufficient range of values for this variable.

However, if *number* [$i$] is a multiple word variable, then it is a non-trivial problem to ensure that it is always at most one greater than some previously chosen *number* [$j$]. Suppose, for example, that the value is stored one decimal digit per word. If the value of *number* [$k$] increases from 99 to 100 while process $i$ is choosing the value of *number* [$i$], then $i$ could read a value of 199 for *number* [$k$] — thus choosing *number* [$i$] $\geqq 200$ although all previously chosen values of *number* [$j$] were $\leqq 100$. It is shown in [9] that this can be avoided by the following implementation rules.

*I1.* Values of *number* [$i$] are written from right to left (least significant word to most significant word) and are read from left to right.

*I2.* *number* [$i$] is chosen to be the maximum of (i) its previous non-zero value and (ii) $1 + \mathit{maximum}\{\mathit{number}[j] : j \in \mathit{Con}(i)\}$.

In the event of process failure, rule I2 cannot be met if a failed process may forget its previous value of *number* [$i$]. In that case, the failed process must not restart until every read of *number* [$i$] which was initiated before it reached its quiescent state has been completed. It may then pretend that it is starting initially. We must also assume that while process $i$ is malfunctioning, a read of *number* [$i$] does not obtain a value larger than the correct one.

## *A Sample Implementation*

To illustrate how the details of an actual system can be used to obtain a more efficient implementation, we now consider how the simple algorithm might be implemented on our sample system of interconnected computers. First, we will eliminate the *choosing* flags. Assume that one computer reads from another's memory by transmitting a read request. We can then assume that a process never reads a partially written value. The basic idea is for process $i$ to defer action on any read requests it receives while it is in the doorway. Then *choosing* [$i$] always appears false to any other process, so it becomes superfluous. However, this can cause a deadlock because a process must read from other processes' memories in order to leave the doorway. We must therefore have process $i$, while in its doorway, read the value of *number* [$j$] by a special ***urgent*** read request. If process $j$ receives an urgent read request for *number* [$j$] while in its doorway, then it simply returns the value zero. This is a valid implementation of the algorithm, because process $j$ is just acting as if the read request occurred before it began writing the new value of *number*$[j]$. This implies that a computer should never "swap out" a program while it is executing the doorway.

We assume that process $i$ reads both *mode*$[j]$ and *number*$[j]$ with a single read request, for $j \neq i$. Thus, it always obtains a "consistent" pair of values.

We implement the **wait until** statement with **sleep** and **awake** operations, using an array *alarm*$[1:N]$ of alarm flags. A simple approach would be to have a process awaken all connected processes upon leaving the bakery. However, we will attempt to eliminate superfluous **awake** operations. Whether or not this is a good idea will depend upon the actual hardware details.

To handle the problem of process failure, we simply assume that a failed computer $i$ performs an **awake** (*alarm*$[j]$) operation for each $j \in Con(i)$. (A more practical approach would be to have each sleeping process periodically awaken itself.)

The following program for computer $i$ gives an implementation of the above algorithm for this system of interconnected computers.

```
begin integer j, k;
   doorway:
      $mode[i] := mode{\cdot}value_i$
      $number[i] := 1 + maximum\{number[j] : j \in Con(i)\}$;
   waiting queue:
      for all $j \in Con(i)$ do
         while $\#(j) < \#(i)$ and $conflict(i, j)$ do
            sleep(alarm[i]) od;
   critical section$_i$;
   epilogue:
      $number[i] := 0$;
      for all $j \in \{j \in Con(i) : number[j] > 0$ and $conflict(i, j)\}$ do
         if $\{k \in Con(i) : \#(k) < \#(j)$ and $conflict(k, j)\} = \phi$
            then awake (alarm[j]) fi od
end
```

In the epilogue, process $i$ need only read the values of *number*$[j]$ and *mode*$[j]$ once for each $j \in Con(i)$. It can then save these values and use them as required within the **for all** loop. This read of *number*$[j]$ is a *non*-urgent one.

It is easy to verify that this implementation still satisfies Assertions 1 and 2, so D0–D3 hold. The proof of D4 is the same as before, once we show that conditions D4 (a)–(e) imply that process $i$ is not sleeping. To do this, we assume that $i$ is asleep and obtain a contradiction. Suppose that $i$ went to sleep during the $j$-th iteration of the waiting queue's **for all** loop. Let $e$ be the event of reading *number* $[j]$ and *mode* $[j]$ just before going to sleep, when process $i$ found $\#(j) < \#(i)$ and *conflict* $(i, j)$ both to be true. Since $j$ was in the bakery at event $e$, and no process in *Con* $(i)$ could have failed after event $e$ (otherwise, by our assumptions it would awaken process $i$), conditions D4 (c) and (e) imply that process $j$ left the bakery after event $e$[6]. Moreover, by our assumption that $i$ read *number* $[j]$ and *mode* $[j]$ with a single read request, $j$ left the bakery with *conflict* $(i, j) =$ *true*.

Now let $k$ be the process which most recently left the bakery with *conflict* $(i, k) =$ *true*. It must have left the bakery after event $e$. Since $k$ did not awaken process $i$, when executing the epilogue's **for all** loop it must have found some other process $l$ still in the bakery with $\#(l) < \#(i)$ and *conflict* $(l, i)$ both true. Process $l$ must then have left the bakery after process $k$ did, contradicting our choice of $k$. Hence, process $i$ cannot be asleep.

## *The General Algorithm*

We now describe an algorithm to implement the general statement

**region** $mode_i$ **when** $condition_i$ **do** *critical* $section_i$ **od**

in process $i$. For processes $i$ and $j$ with $j \in Con(i)$ we assume a **region** statement of the form

**region** $(i, j)$ **do** *critical* $section_{(i,j)}$ **od**

in process $i$ and

**region** $(j, i)$ **do** *critical* $section_{(j,i)}$ **od**

in process $j$. These have the property that process $i$ cannot be in *critical* $section_{(i,j)}$ while process $j$ is in *critical* $section_{(j,i)}$. These statements are implemented by the special case of the simple algorithm with $N = 2$ in which different processes always conflict. Note that a **region** $(i, j)$ statement and a **region** $(i, j')$ statement are implemented with disjoint sets of variables if $j \neq j'$.

Process $i$ maintains a boolean variable *precede* $[i, j]$ for $j \in Con(i)$. Its value is *true* only if $i$ may enter *critical* $section_i$ before $j$ enters *critical* $section_j$. Process $i$ only sets this variable to *true* from inside a **region** $(i, j)$ critical section. If $i$ and $j$ conflict, then $i$ will not set *precede* $[i, j]$ true unless *precede* $[j, i]$ is false. The use of the **region** $(i, j)$ statement thus prevents *precede* $[i, j]$ and *precede* $[j, i]$ from both being true at the same time.

The general algorithm for process $i$ is given below. For convenience, it is described with busy waiting. Practical implementation is discussed later. The boolean function *may·precede* $(i, j)$ is defined to equal

*number* $[j] = 0$ **or** $\sim$*conflict* $(i, j)$ **or**
$[\#(i) < \#(j)$ **and** $\sim$*should·precede* $(j, i)]$ **or**
$[\#(j) < \#(i)$ **and** *should·precede* $(i, j)]$.

If $S$ is a set of boolean values, then $\wedge S$ denotes the logical *and* of all the elements of $S$, so it equals *true* iff $S = \{true\}$. All booleans are initialized to *false*.

6 Our assumptions allow us to consider the setting of *number* $[j]$ to be a single event, so *number* $[j] > 0$ implies that $j$ is in the bakery.

```
begin integer j;
  doorway
    mode[i] := mode_i;
    number[i] := any integer > maximum{number[j] : j ∈ Con(i)};

  waiting queue
    while ~condition_i or ~∧{precede[i, j] : j ∈ Con(i)} do
      for all j ∈ Con(i) do
        if may·precede(i, j)
          then region (i, j) do
                 precede[i, j] := ~precede[j, i] or
                                  ~conflict(i, j) od
          else precede[i, j] := false
        fi od
    od;

  critical section_i
  number[i] := 0;
  for all j ∈ Con(i) do precede[i, j] := false od
end
```

Note that the use of the **region** $(i, j)$ statements eliminates the need for the *choosing* flag.

*Correctness of the General Algorithm*

The proof of correctness of the general algorithm is similar to that of the simple algorithm. First, observe that Assertion 1 holds for the general algorithm as well as the simple one, so D0 is satisfied. In place of Assertion 2, we have the following.

*Assertion 2'.* If $precede[i, j]$ *and* $precede[j, i] = true$ and $j \in Con(i)$, then $conflict(i, j) = false$.

*Proof.* We first define certain process events. Let $x$ denote either one of $i$ or $j$, and let $y$ denote the other one. Let $wp_x$ denote the most recent event in which process $x$ wrote $precede[x, y]$. Since $precede[x, y] = true$, $wp_x$ occurred while executing the critical section of the **region** $(x, y)$ statement. Let $rp_x$ and $rm_x$ denote the reads of $precede[y, x]$ and $mode[y]$, respectively, during that same execution of the **region** $(x, y)$ statement. Let $wm_x$ denote the event in which process $x$ wrote the current value of $mode[x]$.

For each $x$, we have the following $\dashrightarrow$ relations among these events.

$$\begin{tikzcd}
 & rp_x \arrow[dr] & \\
wm_x \arrow[ur] \arrow[dr] & & wp_x \\
 & rm_x \arrow[ur] &
\end{tikzcd}$$

(The temporal ordering of $rp_x$ and $rm_x$ is immaterial.) Because of the symmetry with respect to interchanging $i$ and $j$, we can assume that $wp_j \rightarrow wp_i$. The mutual exclusion property of the **region** $(i, j)$ statement then implies that $wp_j \rightarrow rp_i$ and $wp_j \rightarrow rm_i$. Since $wm_j \rightarrow wp_j$ we also have $wm_j \rightarrow rm_i$. Therefore, process $i$ set *precede* $[i, j]$ true in event $wp_i$ after obtaining the current values of *precede* $[j, i]$ and *conflict* $(i, j)$. Hence, the current value of *conflict* $(i, j)$ must be *false*. □

In order to enter its critical section, process $i$ must set *precede* $[i, j]$ true for each $j \in Con(i)$. Assertion 2′ therefore implies D1. Before entering its critical section, $i$ must evaluate $condition_i$ and find it true after first setting *precede* $[i, j]$ true for all $j \in Con(i)$. Hence, by Assertion 2′ there cannot be any conflicting process in its critical section while $i$ is evaluating $condition_i$. Restriction R2 then implies D2.

To prove D3, suppose $i$ and $j$ are in the waiting queue, $j \in Con(i)$ and $\#(j) < \#(i)$. Assertion 1 therefore implies that $j$ had entered the **region** statement before $i$ entered the waiting queue. Thus, $j$ was in the **region** statement before $i$ began executing its **while** loop. Condition D3 then follows from the definition of *may·precede* $(i, j)$.

Finally, we prove D4. We must show that for some choice of $M$ and $P$, if (b)–(e) remain true for $M$ $P$-intervals while process $i$ is in the waiting queue, then $i$ will enter its critical section. From the fact that our simple algorithm satisfies D0–D4, we can conclude that there exist $K$ and $P$ such that process $i$ will execute one iteration of its **while** loop within $K$ $P$-intervals. Let us choose $P$ large enough so that any process will execute its doorway or its epilogue within one $P$-interval.

The rest of the proof is now similar to that for the simple algorithm. For each $j \in Con(i)$, one of the same five conditions must hold, except that condition (5) is replaced by the following:

(5′) *conflict* $(i, j) =$ *true*, and $j$ is in the waiting queue, and *should·precede* $(i, j)$ and *should·precede* $(j, i)$ are strongly constant, and [$\#(i) < \#(j)$ and *should·precede* $(j, i) =$ *false*, or else $\#(j) < \#(i)$ and *should·precede* $(i, j) =$ *true*].

Similar reasoning to that used about then shows that process $i$ must enter its critical section within $|Con(i)| * (3 * K + 2) + 1$ $P$-intervals.

## *Implementation Considerations*

The algorithm given above uses busy waiting. In a practical implementation, upon entering the bakery a process would spawn individual subprocesses to evaluate $condition_i$ and to maintain each variable *precede* $[i, j]$. These subprocesses would be interrupt driven, and would sleep until something changed which required their attention. The main process would sleep until the **while** condition became false, at which time it would terminate the subprocesses and enter the critical section.

It is a complicated but straightforward programming task to translate the algorithm into such a form. Since it requires the definition of a mechanism for starting and aborting subprocesses, we will not bother to perform this translation. We hope that our sample implementation of the simple algorithm will convince the reader that a reasonably efficient implementation of the more complicated general algorithm is also possible.

## *Conclusion*

We have defined the general **region** primitive for synchronizing independent processes. This primitive seems to permit simple solutions to those synchronization problems which basically require the mutual exclusion of conflicting processes from certain critical sections. Problems requiring explicit communication among the processes will have more complicated solutions, and might benefit from another primitive for exchanging messages. However, initially synchronizing the communication will be a mutual exclusion problem. Once that has been solved, designing the dialogue between processes is a straightforward matter.

We saw that the **region** primitive could be used to implement systems which are insensitive to the failure of any individual process—assuming a reasonably well-behaved form of process failure. An algorithm was then described which showed that the primitive can be implemented with fairly minimal assumptions about the nature of interprocess communication. This algorithm is also insensitive to individual process failure, and can be implemented without too much busy waiting. No detailed analysis of its efficiency was made because any real system would probably allow stronger assumptions about interprocess communication, thereby permitting a more efficient implementation. Our main purpose was to show that a solution was possible even under weak assumptions.

If a system uses several independent hardware components, then one would like it to continue to operate correctly despite the failure of any component. Depending upon the application, "failure" might mean anything from physical destruction to turning off for maintenance. By introducing the **region** statement and describing several applications of it, we hoped to show that this problem can be approached from a general, high level language point of view. The algorithm for the **region** statement shows that this high level approach can actually be implemented.

It is more difficult to implement a true multicomputer system than a multiprocess system for a single computer. The standard technique of simply disabling interrupts at crucial times is no longer sufficient. (Observe that a monitor [7] is just an elegant abstraction of this technique.) Careful proofs of correctness are necessary if subtle, time-dependent errors are to be avoided. Writing such proofs enabled us to discover errors in earlier versions of these algorithms.

*Acknowledgement.* This work has been influenced by several enjoyable and informative discussions with Robert Keller.

## References

1. Brinch Hansen, P.: Concurrent programming concepts. Computing Surveys **5**, 223–245 (1973)
2. Courtois, P. J., Heymans, F., Parnas, D. L.: Concurrent control with "Readers" and "Writers". Comm. ACM **14**, 667–668 (1971)
3. Dijkstra, E. W.: The structure of the "THE" multiprogramming system. Comm. ACM **11**, 341–346 (1968)
4. Dijkstra, E. W.: Cooperating sequential processes. In: Genuys, F. (ed.): Programming Languages. New York: Academic Press 1968, p. 43–112
5. Dijkstra, E. W.: Hierarchical ordering of sequential processes. Acta Informatica **1**, 115–138 (1971)

6. Dijkstra, E. W.: Self-stabilizing systems in spite of distributed control. Comm. ACM **17**, 643–644 (1974)
7. Hoare, C. A. R.: A structured paging system. Computer J. **16**, 209–214 (1973)
8. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Comm. ACM **17**, 453–455 (1974)
9. Lamport, L.: On concurrent reading and writing. To appear in Comm. ACM
10. Nilsen, R. N. (ed.): Distributed function computer architectures. Computer **7**, 15–37 (1974)

Leslie Lamport
Massachusetts Computer
Associates, Inc.
26. Princess Street
Wakefield, Mass. 01880
USA