Science of Computer Programming 13 (1989/90) 1-21
North-Holland

A METHOD FOR SOLVING SYNCHRONIZATION PROBLEMS*

Gregory R. ANDREWS

Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA

Communicated by L. Lamport
Received August 1988
Revised June 1989

Abstract. This paper presents a systematic method for solving synchronization problems. The method is based on viewing processes as invariant maintainers. First, a problem is defined and the desired synchronization property is specified by an invariant predicate over program variables. Second, the variables are initialized to make the invariant true and processes are annotated with atomic assignments so the variables satisfy their definition. Then, atomic assignments are guarded as needed so they are not executed until the resulting state will satisfy the invariant. Finally, the resulting atomic actions are implemented using basic synchronization mechanisms. The method is illustrated by solving three problems using semaphores. The solutions also illustrate three general programming paradigms: changing variables, split binary semaphores, and passing the baton. Additional synchronization problems and synchronization mechanisms are also discussed.

1. Introduction

A concurrent program consists of processes (sequential programs) and shared objects. The processes execute in parallel, at least conceptually. They interact by using the shared objects to communicate with each other. Synchronization is the problem of controlling process interaction.

Two types of synchronization arise in concurrent programs [3]. *Mutual exclusion* involves grouping actions into critical sections that are never interleaved during execution, thereby ensuring that inconsistent states of a given process are not visible to other processes. *Condition synchronization* causes a process to be delayed until the system state satisfies some specified condition. For example, communication between a sender process and receiver process is often implemented using a shared buffer. The sender writes into the buffer; the receiver reads from the buffer. Mutual exclusion is used to ensure that a partially written message is not read. Condition synchronization is used to ensure that a message is not overwritten, and that a message is not read more than once.

Numerous synchronization mechanisms have been proposed, but to date there is no systematic method for using any one of them to solve a given synchronization

* This work was supported by NSF Grant CCR-8701516.

problem. Rather, the programmer typically proceeds in an ad hoc way until he or she finds a path through the maze of possible solutions. Having found a candidate solution, the programmer might then try to verify that it is correct. Such *a posteriori* reasoning does not, however, provide much help in arriving at a solution.

This paper presents a systematic method that guides the development of solutions to synchronization problems. The method is inspired by Dijkstra's seminal work on a calculus for deriving sequential programs [8] and by two Dijkstra notes on programming with semaphores [9, 10]. It is also inspired by the observation that synchronization can be viewed as the problem of maintaining a global invariant [19]. In particular, starting with an assertion that specifies the desired synchronization invariant, a sequence of steps is followed to derive a correct program that maintains this invariant.

Section 2 presents the derivation method. Section 3 illustrates the approach by deriving semaphore-based solutions to three familiar problems: critical sections, producers/consumers, and readers/writers. The section also describes three important programming techniques: changing variables, split binary semaphores, and passing the baton. Finally, Section 4 discusses related work and the applicability of the derivation method to other problems and other synchronization mechanisms.

2. Derivation method

The derivation method essentially reverses the steps employed in constructing an assertional proof of a given program. Therefore it is appropriate first to review relevant proof concepts. The focus is on *safety properties*, which assert that nothing bad happens during execution. Later *liveness properties* are considered; these are concerned with scheduling and assert that something good eventually happens.

A program state associates a value with each variable. Variables include those explicitly defined by the programmer and those—like the program counter—that are implicit. Execution of a sequential program results in a sequence of *atomic actions*, each of which indivisibly transforms the state. Execution of a concurrent program results in a sequence of atomic actions for each process. A particular execution of a concurrent program results in a *history*, which is an interleaving of the sequences of atomic actions produced by the processes.[1] Note that the number of possible histories is exponential in the number of atomic actions.

An abstract way to characterize the possible histories generated by a concurrent program is to construct a correctness proof using a programming logic. A compact way to present such a proof is by means of a *proof outline*, which consists of the program text interspersed with assertions. An *assertion* is a predicate that characterizes a set of states. In a proof outline, each atomic statement S is preceded and

[1] Because the state transformation caused by an atomic action is indivisible, executing a set of atomic actions in parallel is equivalent to executing them in some serial order.

followed by an assertion. The resulting *triples*, which are denoted

$$\{P\}\ S\ \{Q\}$$

have the interpretation that if execution of S is begun in a state satisfying P, and if S terminates, then the resulting state will satisfy Q. P is called the *precondition* of S and Q is called the *postcondition* of S. S is thus viewed as a *predicate transformer* since it transforms the state from one in which P is true to one in which Q is true [8].

A proof outline of a concurrent program must meet two requirements. First, the proof outline of each process must accurately characterize the effects of executing that process in isolation as a sequential program. Second, the proof outlines of the different processes must be *interference-free* [23]. In particular, for each pre- and postcondition P in one process and every atomic action S in another process, P must remain true if S is executed. Statement S will be executed only if its precondition, $pre(S)$, is true. Hence, S will not interfere with P if the triple

$$\{P \wedge pre(S)\}\ S\ \{P\}$$

is a theorem in the underlying programming logic.

The first requirement—a proof outline for each process—requires reasoning about sequential execution. The second requirement—noninterference—addresses the effects of concurrent execution. Thus, the key difference between concurrent programming and sequential programming is controlling process interaction so as to preclude interference.

In general, demonstrating noninterference involves checking each atomic action against every assertion in other processes. This requires work polynomial in the number of atomic actions. However, all commonly occurring safety properties can be specified by a predicate, BAD, that must not be true of any state [19]. For example, mutual exclusion is a safety property where a bad state would be one in which two or more processes are in their critical section. If BAD characterizes the states to be avoided, then a program satisfies the safety property specified by BAD if $\neg BAD$ is an *invariant*: an assertion that is true before and after every atomic action and hence is true of every program state. If an invariant is used to express all relations between shared variables, then the work required to show noninterference is linear in the number of atomic actions. In particular, it must only be shown that each atomic action preserves the invariant.

The derivation method is based on this view of processes as *invariant maintainers* with respect to synchronization. It consists of four steps:

Step 1. *Define the problem.* Identify the processes and synchronization property. Introduce variables as needed and write a predicate that specifies the invariant property that is to be maintained.

Step 2. *Outline a solution.* Annotate the processes with assignments to the shared variables and initialize them so that the invariant is true. Group assignments into atomic actions when they must be executed with mutual exclusion.

Step 3. *Ensure the invariant.* For each atomic assignment action, determine the precondition that will ensure that the state resulting from executing the action will satisfy the invariant. Where necessary, guard assignments with delay conditions to ensure that the postcondition of the assignment satisfies the invariant.

Step 4. *Implement the atomic actions.* Transform the atomic assignments into code that employs only sequential statements and the synchronization mechanism to be employed in the final solution.

The first three steps are essentially the same for various synchronization mechanisms. The last step involves using a specific synchronization mechanism to implement mutual exclusion and condition synchronization so atomic actions are indivisible and invariant-preserving. How this is done depends on the synchronization method employed.

The starting point in deriving a solution is coming up with an appropriate invariant. This can be done in one of two ways: either design a predicate BAD that characterizes bad states, then use $\neg BAD$ as the invariant; or directly specify a predicate $GOOD$ that characterizes good states and use $GOOD$ as the invariant. Specific examples are given in the next section.

In a solution outline, the notation

$$\langle S \rangle$$

will be used to indicate that statement list S is to be executed atomically. In the third derivation step, the notation

$$\langle \textbf{await } B \rightarrow S \rangle$$

will be used to indicate that the executing process is to be delayed until S can be executed atomically beginning in a state in which B is true. Here, B *guards* execution of S. The guard is chosen so that when the guarded statement terminates, the invariant will be true.

Dijkstra's weakest precondition function, *wp*, is used to compute the guards [8]. The weakest precondition $wp(S, Q)$ of statement S and predicate Q is the weakest predicate P such that if execution of S is begun in a state satisfying P, execution is guaranteed to terminate in a state satisfying Q. For an assignment $x := e$, wp is Q with e substituted for each free occurrence of x. For a sequence of assignments "$S1$; $S2$," wp("$S1$; $S2$," $Q) = wp(S1, wp(S2, Q))$; i.e., wp is the composition of the effects of the two assignments. These are the only applications of wp that are needed since all atomic actions resulting from Step 2 are sequences of one or more assignments.

Let K and L be predicates that do not reference variables changed by processes other than the one that will execute atomic action S. Also, assume that

$$\{K\}\ \langle S \rangle\ \{L\}$$

is a theorem. Finally, assume that assertion I is true before execution of S. If I is to be invariant, then it must be true after execution of S. This will be the case if S

is replaced by

$$\langle \textbf{await } B \rightarrow S \rangle,$$

where B is a predicate such that

$$K \wedge I \wedge B \Rightarrow wp(S, L \wedge I).$$

In short, B is chosen so that S is executed only if it will terminate and the resulting state will satisfy $L \wedge I$. To avoid unnecessarily delaying a process, B should be the weakest predicate for which the above formula is true. Often this will simply be the predicate *true*, in which case S need not be guarded.

3. Programming with semaphores

This section illustrates in detail how this derivation method can be used to derive semaphore-based solutions to synchronization problems. It also describes three important programming techniques that can be used with semaphores: changing variables, split binary semaphores, and passing the baton.

Semaphores are abstract data types each instance of which is manipulated by two operations: $\boldsymbol{P}$ and $\boldsymbol{V}$. These operations have the property that the number of completed $\boldsymbol{P}$ operations on a specific semaphore never exceeds the number of completed $\boldsymbol{V}$ operations. A semaphore s is commonly represented by an integer that records the difference between the number of $\boldsymbol{V}$ and $\boldsymbol{P}$ operations; in this case s must satisfy the semaphore invariant SEM: $s \geq 0$. A $\boldsymbol{P}$ operation decrements s; for SEM to be invariant, the decrement must be guarded since

$$wp(s := s-1, SEM) \;=\; s-1 \geq 0 \;=\; s > 0.$$

In contrast, a $\boldsymbol{V}$ operation increments s and need not be guarded since $SEM \Rightarrow wp(s := s+1, SEM)$. Using the notation introduced in the previous section, the semaphore operations are thus:

$$\begin{aligned} &\boldsymbol{P}(s)\text{:} \quad \langle \textbf{await } s > 0 \rightarrow s := s-1 \rangle, \\ &\boldsymbol{V}(s)\text{:} \quad \langle s := s+1 \rangle. \end{aligned}$$

Semaphores as defined above are *general semaphores*: the number of completed $\boldsymbol{V}$ operations can be arbitrarily greater than the number of completed $\boldsymbol{P}$ operations. A *binary semaphore* is a semaphore for which the number of completed $\boldsymbol{V}$ operations can be at most one more than the number of completed $\boldsymbol{P}$ operations. Such a semaphore is called a binary semaphore since its value, when represented as above, can be only 0 or 1. Thus, a binary semaphore b satisfies a stronger invariant $BSEM$: $0 \leq b \leq 1$. Maintaining this invariant requires guarding the $\boldsymbol{V}$ operation as well as the $\boldsymbol{P}$ operation. Using wp as above to compute the guards, the operations on a binary semaphore have the following definitions:

$$\begin{aligned} &\boldsymbol{P}(b)\text{:} \quad \langle \textbf{await } b > 0 \rightarrow b := b-1 \rangle, \\ &\boldsymbol{V}(b)\text{:} \quad \langle \textbf{await } b < 1 \rightarrow b := b+1 \rangle. \end{aligned}$$

As long as a binary semaphore is used in such a way that a V operation is executed only when b is 0, $V(b)$ will not cause delay.[2]

3.1. Critical sections: Changing variables

In the critical section problem, each of N processes $P[1:N]$ repeatedly executes a critical section of code in which it requires exclusive access to some shared resource, and a noncritical section, in which it computes using only local objects. Let $in[i]$ be 1 when $P[i]$ is in its critical section, and 0 otherwise. (It is assumed that *in* is not altered within any process' critical or noncritical section.) The required property is that at most one process at a time is within its critical section. This can be specified directly by the invariant

$$CS: \quad in[1]+\cdots+in[N]\le 1.$$

Alternatively, the bad state in which more than one process is in its critical section can be specified by

$$BAD: \quad in[1]+\cdots+in[N]>1.$$

Given that all $in[i]$ are 0 or 1, $CS=\neg BAD$ so both specifications yield the same invariant.

The second derivation step is to outline the solution. The processes share array $in[1:N]$, with each process setting and clearing its element of *in* before and after executing its critical section. Initially all elements of *in* are zero, so the invariant is initially true. Thus, a solution outline is:

```
var in[1:N] : integer := ([N] 0)   # Invariant CS
P[i: 1..N]:: do true → {in[i] = 0}
                       ⟨in[i] := 1⟩
                       {in[i] = 1}
                       Critical Section
                       ⟨in[i] := 0⟩
                       {in[i] = 0}
                       Noncritical Section
             od
```

The outline for each process contains assertions about the element of *in* manipulated by that process.[3] These assertions follow from the actions of the process, and are not interfered with. Thus, the solution outline is also a valid proof outline. However, it is not yet strong enough to conclude that execution of the critical sections is mutually exclusive. For this, invariant *CS* must be included in the proof outline.

[2] This is usually assumed and hence the V operation on a binary semaphore is often defined to be simply $\langle b := b+1\rangle$. However, if a V is incorrectly executed when b is 1, then b will no longer satisfy its definition *BSEM*.

[3] The notation $([N]\,0)$ in the initialization of *in* denotes a vector of N zeros. The notation $P[i\colon 1..N]$ denotes an array of N processes; within each process, index i has a unique value between 1 and N.

The third step is to guard assignments to *in* to ensure that *CS* is true after each atomic action. (Since *CS* is true initially, this will ensure that it is invariant.) Consider the first assignment, which sets $in[i]$ and thus establishes $in[i]=1$. Computing the weakest precondition as described in Section 2 yields:

$$\begin{aligned}&wp(in[i]:=1, in[i]=1 \wedge CS)\\ &=(1=1 \wedge in[1]+\cdots+in[i-1]+1+in[i+1]+\cdots+in[N]\leq 1).\end{aligned}$$

Since all elements of *in* are either 0 or 1, and $in[i]$ is zero before the assignment, this simplifies to

$$in[1]+\cdots+in[N]=0.$$

This is chosen as the guard for the first atomic action since no weaker predicate suffices. For the second assignment, which clears $in[i]$, *wp* is again computed:

$$\begin{aligned}&wp(in[i]:=0, in[i]=0 \wedge CS)\\ &=(0=0 \wedge in[1]+\cdots+in[i-1]+0+in[i+1]+\cdots+in[N]\leq 1).\end{aligned}$$

Since this is implied directly by precondition "$in[i]=1 \wedge CS$," the second atomic action need not be guarded. Adding the guard to the first atomic action, the solution becomes:

```
var $in[1:N]$ : integer := ([$N$] 0)   # Invariant $CS$
$P[i$: 1..$N]$:: do true → {$in[i]=0 \wedge CS$}
                        ⟨await $in[1]+\cdots+in[N]=0 \rightarrow in[i]:=1$⟩
                        {$in[i]=1 \wedge CS$}
                        Critical Section
                        ⟨$in[i]:=0$⟩
                        {$in[i]=0 \wedge CS$}
                        Noncritical Section
             od
```

Since the construction has ensured that *CS* is invariant, the above is a valid proof outline. Moreover, the solution is correct since the preconditions for critical sections in different processes cannot simultaneously be true, and hence the critical sections execute with mutual exclusion [22].

The remaining derivation step is to use semaphores to implement the atomic statements. Here this can be done by *changing variables* so that each atomic statement becomes a semaphore operation. Let *mutex* be a semaphore whose value is

$$mutex = 1-(in[1]+\cdots+in[N]).$$

This relation is chosen since it makes *mutex* nonnegative, as required for a semaphore. With this change, the atomic statements in the above solution can be replaced by

$$\langle \textbf{await } mutex > 0 \rightarrow mutex := mutex - 1;\ in[i] := 1\rangle$$

and

$$\langle mutex := mutex + 1;\ in[i] := 0\rangle.$$

But now *in* is an *auxiliary variable*: it is used only in assignments to itself. Thus, the program has the same properties if *in* is deleted [23]. After deleting *in*, the atomic statements are simply semaphore operations, so the final solution is:

```
var mutex : semaphore := 1
P[i: 1..N]:: do true → P(mutex)
                       Critical Section
                       V(mutex)
                       Noncritical Section
             od
```

This technique of changing variables leads to a compact solution. It can be employed whenever the following conditions hold:

(1) Semantically different guards reference disjoint sets of variables, and these variables are referenced only in atomic statements.
(2) Each guard can be put in the form $expr > 0$ where *expr* is an integer expression.
(3) Each guarded atomic statement contains one assignment that decrements the value of the expression in the transformed guard.
(4) Each unguarded atomic statement increments the value of the expression in one transformed guard.

Given these conditions, one semaphore can be used for each different guard. The variables that were in the guards then become auxiliary variables and the atomic statements simplify to semaphore operations.

3.2. Producers and consumers: Split binary semaphores

Although the above solution to the critical section problem is obvious to those familiar with semaphores, it was derived systematically in a way that made clear why the solution is correct. This section examines a problem whose solution is somewhat less obvious, but no harder to derive. The solution illustrates another use of changing variables. It also illustrates the important concept of a split binary semaphore [9, 14].

In the producers/consumers problem, producers send messages that are received by consumers. The processes communicate using a single shared buffer, which is manipulated by two operations: *deposit* and *fetch*. Producers insert messages into the buffer by calling *deposit*; consumers receive messages by calling *fetch*. To ensure that messages are not overwritten before being received and are only received once, execution of *deposit* and *fetch* must alternate, with *deposit* executed first.

The starting point is to specify the required alternation property. In the critical section problem, the concern was only whether a process was inside or outside its critical section; thus one variable per process was sufficient to specify the mutual exclusion property. Here, however, it is necessary to know how many times *deposit* and *fetch* have been executed, then to bound the difference to ensure alternation.

The way to specify this property—or similar properties such as repeated rendezvous at a barrier—is to use incrementing counters to indicate when a process reaches critical execution points. Here the critical points are starting and completing execution of *deposit* and *fetch*. Thus, let *inD* and *afterD* be integers that count the number of times producers have started and finished executing *deposit*. Also, let *inF* and *afterF* be integers that count the number of times consumers have started and finished executing *fetch*. Then the required alternation property can be expressed by the predicate

$$PC\colon\quad inD \leqslant afterF + 1 \ \wedge\ inF \leqslant afterD.$$

In words, this says that *deposit* can be started at most one more time than *fetch* has been completed, and that *fetch* can be started no more times than *deposit* has been completed.[4]

For this problem, the shared variables are the above counters and a variable *buf* that holds one message of some arbitrary type *T*. Since the main concern is only how producers and consumers communicate and synchronize, each process simply executes a loop; producers repeatedly deposit messages and consumers repeatedly fetch them. Annotating the processes with appropriate assignments to the shared variables yields the following solution outline.

```
var buf: T   #for some type T
var inD, afterD, inF, afterF : integer := 0, 0, 0, 0   # Invariant PC
Producer[i: 1..M]:: do true → produce message m
                              deposit: ⟨inD := inD + 1⟩
                                       buf := m
                                       ⟨afterD := afterD + 1⟩
                    od
Consumer[j: 1..N]:: do true → fetch: ⟨inF := inF + 1⟩
                                     m := buf
                                     ⟨afterF := afterF + 1⟩
                              consume message m
                    od
```

No assertions are included in the above program since there are no meaningful ones that would not be interfered with. Also the references to *buf* are not enclosed in angle brackets since it will be ensured that *deposit* and *fetch* alternate, and hence that access to *buf* is atomic.

To extend the above outline to a correct solution, assignments are guarded as necessary to ensure the invariance of synchronization property *PC*. Again, *wp* is used to compute the guards. The increments of *inD* and *inF* need to be guarded, but the increments of *afterD* and *afterF* need not be since they clearly preserve the

[4] Again, the property could be specified by characterizing the bad state, then using the negation of that predicate. Here the bad state is one in which two or more deposits or fetches are executed in a row.

invariant.[5] Adding guards that ensure the invariance of *PC* yields the solution:

```
var buf: T    # for some type T
var inD, afterD, inF, afterF : integer := 0, 0, 0, 0    # Invariant PC
Producer[i: 1..M]:: do true → produce message m
                              deposit: ⟨await inD ≤ afterF → inD := inD + 1⟩
                                       buf := m
                                       ⟨afterD := afterD + 1⟩
                    od
Consumer[j: 1..N]:: do true → fetch: ⟨await inF < afterD → inF := inF + 1⟩
                                     m := buf
                                     ⟨afterF := afterF + 1⟩
                              consume message m
                    od
```

The final step is to implement the statements that access the counters. Again the technique of changing variables can be used since the required conditions are met. In particular, let *empty* and *full* be semaphores whose values are:

$empty = afterF - inD + 1,$

$full = afterD - inF.$

With this change, the four counters become auxiliary variables so can be deleted. Thus, the first statements in *deposit* and *fetch* become ***P*** operations and the last become ***V*** operations. This yields the final solution.

```
var buf: T    # for some type T
var empty, full : semaphore := 1, 0    # Invariant 0 ≤ empty + full ≤ 1
Producer[i: 1..M]:: do true → produce message m
                              deposit: P(empty)
                                       buf := m
                                       V(full)
                    od
Consumer[j: 1..N]:: do true → fetch: P(full)
                                     m := buf
                                     V(empty)
                              consume message m
                    od
```

In the solution, *empty* and *full* are both binary semaphores. Moreover, together they form what is called a *split binary semaphore*: at most one of them is one at a time. In general a split binary semaphore is a collection of binary semaphores the sums of whose values is always 0 or 1. More precisely, if $b[1], \dots, b[N]$ are binary

[5] In general, it is never necessary to delay when leaving a critical section of code.

semaphores, they form a split binary semaphore if the following assertion is invariant:

$$SPLIT: \quad 0 \leq b[1] + \cdots + b[N] \leq 1.$$

The term "split binary semaphore" comes from the fact that the $b[i]$ can be viewed as being the result of splitting a single binary semaphore b into N binary semaphores such that *SPLIT* is invariant.

The importance of split binary semaphores comes from the way in which they can be used to implement mutual exclusion. Given a split binary semaphore, suppose that one of the constituent semaphores has initial value one (hence the others are initially 0). Further suppose that every process uses the semaphores by alternately executing a ***P*** operation then a ***V*** operation. Then, all statements between any ***P*** and the next ***V*** execute with mutual exclusion. This is because while one process is between a ***P*** and a ***V***, the semaphores are all zero and hence no other process can complete a ***P*** until the first process executes a ***V***. The above solution to the producer/consumer problem illustrates this.

3.3. Readers and writers: Passing the baton

As a final example, a new solution is derived for the classic readers/writers problem [7]. The solution introduces a general programming paradigm called "passing the baton." This paradigm employs split binary semaphores to provide exclusion and to control which delayed process is next to proceed.

In the readers/writers problem, two kinds of processes share a database. Readers examine the database; writers both examine and alter it. To preserve database consistency, a writer requires exclusive access. However, any number of readers may execute concurrently. To specify the synchronization property, let nr and nw be nonnegative integers that respectively record the number of readers and writers accessing the database. The bad state to be avoided is one in which both nr and nw are positive, or nw is greater than one. The inverse set of good states is characterized by the predicate:

$$RW: \quad (nr = 0 \vee nw = 0) \wedge nw \leq 1.$$

The first term says readers and writers cannot access the database at the same time; the second says there is at most one active writer. Assuming each process executes a perpetual loop, annotating the processes yields the solution outline:

```
var nr, nw : integer := 0, 0    # Invariant RW
Reader[i: 1..M]:: do true → ⟨nr := nr + 1⟩
                             read the database
                             ⟨nr := nr − 1⟩
                  od
Writer[j: 1..N]:: do true → ⟨nw := nw + 1⟩
                             write the database
                             ⟨nw := nw − 1⟩
                  od
```

The assignments to the shared variables now need to be guarded so that RW is invariant. From

$$wp(nr := nr+1, RW) \;=\; (nr=-1 \vee nw=0)$$

and the fact that nr and nw are nonnegative, $nr := nr+1$ must be guarded by $nw=0$. Similarly, $nw := nw+1$ must be guarded by $(nr=0 \wedge nw=0)$. Neither decrement need be guarded, however. Informally this is because it is never necessary to delay a process that is giving up use of a resource. More formally,

$$wp(nr := nr-1, RW) \;=\; ((nr=1 \vee nw=0) \wedge nw \leqslant 1).$$

This is true since $(nr>0 \wedge RW)$ is true before nr is decremented. The reasoning for $nw := nw-1$ is analogous. Inserting the guards yields the solution:

```
var nr, nw : integer := 0, 0    # Invariant RW
Reader[i: 1..M]:: do true → ⟨await nw = 0 → nr := nr + 1⟩
                            read the database
                            ⟨nr := nr − 1⟩
                  od
Writer[j: 1..N]:: do true → ⟨await nr = 0 and nw = 0 → nw := nw + 1⟩
                            write the database
                            ⟨nw := nw − 1⟩
                  od
```

Here, the two guards overlap so the technique of changing variables cannot be used to implement the atomic statements. This is because no one semaphore could discriminate between the guards. Thus a different technique is required. The one introduced here is called *passing the baton*, for reasons explained below. Of note is that it is powerful enough that it can always be used.

Using the derivation method, after the third step the solution will contain atomic statements having either of two forms:

F_1: $\langle S_i \rangle$

or

F_2: $\langle$**await** $B_j \rightarrow S_j\rangle$.

These statements can be implemented using split binary semaphores as follows. First, let e be a binary semaphore whose initial value is one. It is used to control *entry* into the atomic statements. Second, associate one semaphore c_j and one counter d_j with each semantically different guard B_j; these are all initially zero. Semaphore c_j is used to delay processes waiting for condition B_j to become true; d_j is a count of the number of processes delayed (or about to delay) on c_j.

The entire set of semaphores—e and the c_j—are used as follows so they form a split binary semaphore. Statements of form F_1 are replaced by the program fragment

```
F₁: $P(e)\ \{I\}$
    $S_i\ \{I\}$
    $SIGNAL$
```

and statements of form F_2 are replaced by the program fragment

```
F₂: $P(e)\ \{I\}$
    **if** $B_j \rightarrow$ **skip** □ **not** $B_j \rightarrow d_j := d_j + 1;\ V(e);\ P(c_j)$ **fi** $\{I \wedge B_j\}$
    $S_j\ \{I\}$
    $SIGNAL$
```

The program fragments are annotated with assertions that are true at critical points, with I being the synchronization invariant. In both schemes, *SIGNAL* is the program fragment

```
SIGNAL: **if** $B_1$ **and** $d_1 > 0 \rightarrow \{I \wedge B_1\}\ d_1 := d_1 - 1;\ V(c_1)$
        □ · · ·
        □ $B_N$ **and** $d_N > 0 \rightarrow \{I \wedge B_N\}\ d_N := d_N - 1;\ V(c_N)$
        □ **else** $\rightarrow \{I\}\ V(e)$
        **fi**
```

where **else** is an abbreviation for the negation of the disjunction of the other guards (i.e., **else** is true if none of the other guards is true). The first N guards in *SIGNAL* check whether there is some process waiting for a condition that is now true. Again, the program fragments are annotated with assertions that are true at critical points.

With these replacements, the semaphores form a split binary semaphore since at most one semaphore at a time is one and every execution path starts with a P and ends with a single V. Hence the statements between any P and V execute with mutual exclusion. The synchronization invariant I is true before each V operation, so is true whenever one of the semaphores is one. Moreover, B_j is guaranteed to be true whenever S_j is executed. This is because either the process checked B_j and found it to be true, or the process delayed on c_j, which is signaled only when B_j is true. In the latter case, the predicate B_j is effectively transferred to the delayed process. Finally, the transformation does not introduce deadlock since c_j is signaled only if some process is waiting on or about to be waiting on c_j.

The method is called *passing the baton* because of the way in which semaphores are signaled. When a process is executing within a critical region, think of it as holding a baton that signifies permission to execute. When that process reaches a *SIGNAL* fragment, it passes the baton to one other process. If some process is waiting for a condition that is now true, the baton is passed to one such process, which in turn executes the critical region and passes the baton to another process. When no process is waiting for a condition that is true, the baton is passed to the next process trying to enter the critical region for the first time—i.e., a process waiting on $P(e)$.

This replacement scheme can be applied to the abstract solution to the readers/writers problem as follows. In that program there are two different guards, so two condition semaphores and associated counters are needed. Let semaphores r and w represent the reader delay condition $nw = 0$ and the writer delay condition $(nr = 0 \wedge nw = 0)$, respectively. Let dr and dw be the associated counters. Finally, let e be the entry semaphore. Performing the baton passing replacements described above, the solution becomes:

```
var nr, nw : integer := 0, 0    # Invariant RW
var e, r, w : semaphore := 1, 0, 0    # Invariant 0 ≤ (e + r + w) ≤ 1
var dr, dw : integer := 0, 0    # Invariant dr ≥ 0 ∧ dw ≥ 0
Reader[i: 1..M]:: do true → P(e)
                      if nw = 0 → skip
                      □ nw > 0 → dr := dr + 1; V(e); P(r)
                      fi
                      nr := nr + 1
                      SIGNAL_1
                      read the database
                      P(e)
                      nr := nr − 1
                      SIGNAL_2
                  od
Writer[j: 1..N]:: do true → P(e)
                      if nr = 0 and nw = 0 → skip
                      □ nr > 0 or nw > 0 → dw := dw + 1; V(e); P(w)
                      fi
                      nw := nw + 1
                      SIGNAL_3
                      write the database
                      P(e)
                      nw := nw − 1
                      SIGNAL_4
                  od
```

Above, $SIGNAL_i$ is an abbreviation for

```
SIGNAL_i:  if nw = 0 and dr > 0 → dr := dr − 1; V(r)
           □ nr = 0 and nw = 0 and dw > 0 → dw := dw − 1; V(w)
           □ (nw > 0 or dr = 0) and (nr > 0 or nw > 0 or dw = 0) → V(e)
           fi
```

Note that $SIGNAL_i$ ensures that nw is zero when semaphore r is signaled and that both nr and nw are zero when semaphore w is signaled.

Here, and in general, the preconditions of the *SIGNAL* fragments allow many of the guards to be simplified or eliminated. In reader processes, $nr > 0 \wedge nw = 0$ is true before $SIGNAL_1$, and $nw = 0 \wedge dr = 0$ is true before $SIGNAL_2$. In writer processes, $nr = 0 \wedge nw > 0$ is true before $SIGNAL_3$, and $nr = 0 \wedge nw = 0$ is true before $SIGNAL_4$. Using these facts to simplify the signal protocols yields the final solution:

```
var nr, nw : integer := 0, 0   # Invariant RW
var e, r, w : semaphore := 1, 0, 0   # Invariant $0 \leq (e + r + w) \leq 1$
var dr, dw : integer := 0, 0   # Invariant $dr \geq 0 \wedge dw \geq 0$
Reader[i: 1..M]:: do true → P(e)
                            if $nw = 0$ → skip
                            □ $nw > 0$ → $dr := dr + 1$; V(e); P(r)
                            fi
                            $nr := nr + 1$
                            if $dr > 0$ → $dr := dr - 1$; V(r)
                            □ $dr = 0$ → V(e)
                            fi
                            read the database
                            P(e)
                            $nr := nr - 1$
                            if $nr = 0$ and $dw > 0$ → $dw := dw - 1$; V(w)
                            □ $nr > 0$ or $dw = 0$ → V(e)
                            fi
                  od
Writer[j: 1..N]:: do true → P(e)
                            if $nr = 0$ and $nw = 0$ → skip
                            □ $nr > 0$ or $nw > 0$ → $dw := dw + 1$; V(e); P(w)
                            fi
                            $nw := nw + 1$
                            V(e)
                            write the database
                            P(e)
                            $nw := nw - 1$
                            if $dr > 0$ → $dr := dr - 1$; V(r)
                            □ $dw > 0$ → $dw := dw - 1$; V(w)
                            □ $dr = 0$ and $dw = 0$ → V(e)
                            fi
                  od
```

Note that the last **if** statement in writer processes is nondeterministic: if there are both delayed readers and delayed writers, either could be signaled.

Like the classic solution to this problem [7], the above solution gives readers preference over writers. Unlike the classic solution, however, the above solution can readily be modified to schedule processes in other ways. For example, to give writers preference it is necessary to ensure (1) that new readers are delayed if a writer is waiting, and (2) that a delayed reader is awakened only if no writers are waiting. The first requirement is met by changing the first **if** statement in reader processes to

```
if $nw = 0$ and $dw = 0 \rightarrow$ skip
□ $nw > 0$ or $dw > 0 \rightarrow dr := dr + 1$; $V(e)$; $P(r)$
fi
```

The second requirement is met by strengthening the first guard in the last **if** statement in writer processes to $dr > 0$ **and** $dw = 0$. This eliminates the nondeterminism, which is always safe to do. Note that these changes in no way alter the structure of the solution.

It is also possible to ensure fair access to the database, assuming semaphore operations are themselves fair. For example, readers and writers can be forced to alternate turns when both are waiting: When a writer finishes, all waiting readers get a turn; when they finish, one waiting writer gets a turn, etc. This alternation can be implemented by adding a Boolean variable *writer_last* that is set true when a writer starts writing, and is cleared when a reader starts reading. Also change the last **if** statement in writer processes to

```
if $dr > 0$ and ($dw = 0$ or $writer\_last$) $\rightarrow dr := dr - 1$; $V(r)$
□ $dw > 0$ and ($dr = 0$ or not $writer\_last$) $\rightarrow dw := dw - 1$; $V(w)$
□ $dr = 0$ and $dw = 0 \rightarrow V(e)$
fi
```

Again the structure of the solution is unchanged.

This technique of passing the baton can also be used to provide finer-grained control over the order in which processes use resources. In the extreme, one semaphore can be associated with each process and thus can be used to control exactly which delayed process is awakened. This might be used, for example, by a memory allocator, in which case a process would before delaying record the amount of memory it required.

In fact, the passing the baton paradigm can be used to solve almost any synchronization problem. This is because (1) most safety properties can be expressed as an invariant, (2) an invariant can be ensured by guarding atomic actions, (3) any collection of guarded atomic actions can be implemented using the passing the baton transformation, and (4) the exact order delayed processes are serviced can be controlled by associating one condition semaphore with each process. The only thing that cannot be controlled is the order in which processes delayed on the entry semaphore are serviced. This depends on the underlying implementation of semaphores.

4. Discussion

The examples have illustrated how it is possible to derive solutions to synchronization problems in a systematic way. The key to the approach is viewing processes as invariant maintainers and the critical first step is to come up with an invariant. The general idea is to specify either the bad state to avoid or the good state to ensure. The examples illustrated three specific techniques: using variables to record when a process is in a critical section, to record passage through key execution points, and to count the number of processes in a certain state. These same techniques can be used for numerous additional problems such as the dining philosophers, barrier synchronization, parallel garbage collection, disk scheduling, and the sleeping barber problem. These examples and several others are described in [1].

Given an invariant, the second and third derivation steps are essentially mechanical: make collections of assignments atomic, then guard atomic assignments to ensure the invariant, with *wp* being used to compute the necessary guard. The result is an abstract solution that uses atomic statements and **await** statements.

The final step is to implement the abstract solution. This paper has shown how to do so using semaphores. In particular, two different techniques were introduced: changing variables and passing the baton, which employs split binary semaphores. When the changing variables technique can be employed, it results in a compact solution. In any event, however, the passing the baton technique can always be employed. Another virtue of passing the baton is that different scheduling policies can be realized by making slight modifications to a solution. The readers/writers solution illustrated this.

Other synchronization mechanisms can also be used to implement the abstract solution.[6] Busy waiting is a form of synchronization in which a process repeatedly checks a synchronization condition until it becomes true. Using busy waiting, $\langle S\rangle$ can be implemented as

```
CSenter; S; CSexit
```

where CSenter and CSexit are entry and exit protocols for some solution to the critical section problem (e.g., Peterson's algorithm [24], the bakery algorithm [17], or test-and-set instructions). Similarly, $\langle$**await** $B \rightarrow S\rangle$ can be implemented as

```
CSenter; do not B → CSexit; CSenter od; S; CSexit
```

[6] How this is done is described in detail in [1].

where CSenter and CSexit are the same critical section protocols as above. (To reduce memory contention on a multiprocessor, a random delay can be inserted between the exit and re-enter protocols within the **do** loop.)

Conditional critical regions (CCRs) are another notation for specifying synchronization [4, 13]. CCRs employ resources and **region** statements, which are quite similar to **await** statements. A resource is a collection of shared variables that are used together. A region statement has the form

region r **when** $B \rightarrow S$ **end**

where r is a resource name, B is a Boolean expression, and S is a statement list. Evaluation of B and execution of S are an atomic action, as with an **await** statement. Thus, an abstract solution resulting from the derivation method can be implemented using CCRs by grouping shared variables into a resource and using **region** statements to implement atomic actions. The synchronization invariant associated with the resource is now what is called a resource invariant [22].

It is also straightforward to use monitors [5, 14]. A monitor consists of a collection of permanent variables and procedures that implement operations on these variables. The permanent variables are analogous to a CCR resource, and the procedures implement the different region statements. Procedures in a monitor automatically execute with mutual exclusion. Condition synchronization is provided by means of condition variables, and **wait** and **signal** statements. Condition variables are very similar to semaphores: **wait**, like ***P***, delays a process and **signal**, like ***V***, awakens a process. One difference is that execution of **signal** immediately transfers control to the awakened process.[7] The second difference is that **signal** has no effect if no process is waiting on the signaled condition variable whereas a ***V*** operation on a semaphore always increments the semaphore.

Because of this similarity, the passing the baton method can be used almost directly to implement **await** statements. In particular, each atomic action in an abstract solution becomes a monitor procedure. The body of the procedure for an action $\langle S_i \rangle$ becomes

S_i; *SIGNAL*

and the body of the procedure for an action $\langle$**await** $B_j \rightarrow S_j\rangle$ becomes

if $B_j \rightarrow$ **skip** $\Box$ **not** $B_j \rightarrow$ **wait**(c_j) **fi**; S_j; *SIGNAL*

where c_j is a condition variable associated with B_j. In both cases *SIGNAL* is the program fragment

if B_1 **and not empty**$(c_1) \rightarrow$ **signal**(c_1)

[7] Other signaling semantics have also been proposed [3, 16]. They affect the way in which **await** statements are implemented, as described in [1].

```
□ · · ·
□ $B_N$ and not empty($c_N$) → signal($c_N$)
□ else → skip
fi
```

where **empty**(c) is an operation that returns true if there is no process waiting on condition variable c. The main difference between these transformations and those used with semaphores is the absence of an analog of the entry semaphore, which is not needed since exclusion is implicitly provided.

As with semaphores, the monitor signaling protocol can often be simplified by taking its precondition into account and by taking advantage of the fact that executing **signal** has no effect if a condition queue is empty. Also, a priority **wait** statement can sometimes be used to order delayed processes; this decreases the number of condition variables required to solve many scheduling problems.[8] The main point, though, is that the basic approach when using monitors is the same as with semaphores. The advantages of monitors relative to semaphores are that shared variables are encapsulated and exclusion is implicit.

The above synchronization mechanisms are all based on shared variables. Another class of synchronization constructs is based on message passing. In this case, processes share only channels along which messages are sent and received.

Any concurrent program that employs shared variables can be converted to a distributed program that uses message passing. For example, a monitor-based program can be converted by changing each monitor into a server process that repeatedly services requests from the other processes, which are called its *clients* [20]. Procedure call and return are simulated using message passing primitives. Permanent monitor variables become variables local to the server process. The server executes a perpetual loop; the loop invariant is what was the monitor invariant. Within the loop, the server repeatedly receives a request to perform an operation, and either executes it or defers it until later. A request is deferred exactly when a monitor procedure would have waited on a condition variable; it is serviced when doing so will result in a state in which the loop invariant is true. The exact way in which the clients and server are programmed depends on the specific kind of message passing that is employed (the possibilities are surveyed in [3]).

Rather than converting a monitor-based program, a client/server program can be derived directly using a variation on the basic derivation method. The steps are the same but are realized somewhat differently. First, for each abstract resource, define the operations its server will provide and specify the invariant the server will maintain. Then outline the server process, which executes a perpetual loop consisting of operation fragments of the form

```
$op_i(params_i)$:  $S_i$;
```

[8] Multiple operations can also be combined into a single procedure, which may then have internal delay points, preceded as needed by signals to awaken other processes.

there is one such fragment for each operation the server implements. Third, guard each operation fragment to ensure that the loop invariant will be true after executing S_i. Finally, use some collection of message passing primitives to implement client/server communication and the operation fragments.

The biggest difference between using message passing as opposed to shared variables is the way in which the implementation step is realized. In fact, there are a number of possibilities since there are numerous different message passing primitives. At one extreme, the SR language facilitates a straightforward implementation [2] since operations can be serviced by a single input statement, which can contain guards that reference operation parameters. At the other extreme, in a language such as PLITS [12], the server repeatedly waits for a message, then executes a **case** statement based on the kind of request, then either honors or saves the request, depending on its local state. The implementation step is realized by a combination of these two extremes in other languages such as Ada [25] and CSP [15]. For example, the server structure in Ada is similar to that in SR when the guards do not depend on operation parameters; otherwise a structure like that of the PLITS solution must be used (although families of entries can sometimes be employed).

This derivation method can be used to design shared-variable solutions to synchronization problems. It can also be used directly to design servers in distributed programs. The main concepts of the method—invariants and guarding actions to ensure them—are also applicable to more general distributed programs. For example, [18] shows how a shared variable program for maintaining routing tables in a network can be converted to a distributed program, and then shows that the program maintains a global invariant that implies correctness. A similar approach is applied in [11] to develop two algorithms for computing the topology of a network (see also [21]).

Invariants and guarded actions also play a key role in the Unity notation for parallel programming [6]. Unity is an abstract, architecture-independent notation that was designed to provide a foundation for parallel program design. A Unity program contains four sections: variable declarations, invariant equations that always hold, initialization equations, and a set of assignment statements. Assignments are guarded when necessary to ensure that the invariant equations are always true. A distinctive attribute of Unity is that the order of assignments is immaterial; at any point in time, any one of them can be selected for execution. In contrast, the approach presented here applies to traditional, imperative programs with explicit control flow and synchronization.

Acknowledgement

Fred Schneider suggested the approach described in this paper and inspired the search for general techniques for implementing atomic actions. He, Ron Olsson, and Rick Schlichting provided numerous useful comments on an earlier version of this paper. The three referees also provided suggestions that helped clarify the presentation.

References

[1] G.R. Andrews, *Concurrent Programming: Principles and Practice* (Benjamin/Cummings, Menlo Park, CA, to appear).

[2] G.R. Andrews, R.A. Olsson, et al., An overview of the SR language and implementation, *ACM Trans. Programming Languages Syst.* **10** (1) (1988) 51-86.

[3] G.R. Andrews and F.B. Schneider, Concepts and notations for concurrent programming, *ACM Comput. Surveys* **15** (1) (1983) 3-43.

[4] P. Brinch Hansen, Structured multiprogramming, *Comm. ACM* **15** (1972) 574-578.

[5] P. Brinch Hansen, *Operating System Principles* (Prentice-Hall, Englewood Cliffs, NJ, 1973).

[6] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).

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

[8] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[9] E.W. Dijkstra, A tutorial on the split binary semaphore, EWD703, Nuenen, Netherlands (1979).

[10] E.W. Dijkstra, The superfluity of the general semaphore, EWD734, Nuenen, Netherlands (1980).

[11] I.J.P. Elshoff and G.R. Andrews, The development of two distributed algorithms for network topology, TR 88-13, Department of Computer Science, The University of Arizona, Tucson, AZ (1988).

[12] J.A. Feldman, High level programming for distributed computing, *Comm. ACM* **22** (1979) 353-368.

[13] C.A.R. Hoare, Towards a theory of parallel programming, in: *Operating Systems Techniques* (Academic Press, New York, 1972).

[14] C.A.R. Hoare, Monitors: An operating system structuring concept, *Comm. ACM* **17** (1974) 549-557.

[15] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.

[16] J.H. Howard, Signaling in monitors, in: *Proceedings 2nd International Conference on Software Engineering*, San Francisco, CA (1976) 47-52.

[17] L. Lamport, A new solution of Dijkstra's concurrent programming problem, *Comm. ACM* **17** (1974) 453-455.

[18] L. Lamport, An assertional correctness proof of a distributed algorithm, *Sci. Comput. Programming* **2** (1982) 175-206.

[19] L. Lamport and F.B. Schneider, The "Hoare logic" of CSP, and all that, *ACM Trans. Programming Languages Syst.* **6** (1984) 281-296.

[20] H.C. Lauer and R.M. Needham, On the duality of operating system structures, in: *Proceedings 2nd International Symposium on Operating Systems*, Paris (1978); Reprinted: *Operating Syst. Rev.* **13** (2) (1979) 3-19.

[21] R. McCurley and F.B. Schneider, Derivation of a distributed algorithm for finding paths in directed networks, *Sci. Comput. Programming* **6** (1986) 1-9.

[22] S.S. Owicki, Axiomatic proof techniques for parallel programs, TR 75-251, Ph.D. Dissertation, Department of Computer Science, Cornell University, Ithaca, NJ (1975).

[23] S.S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Inform.* **6** (1976) 319-340.

[24] G.L. Peterson, Myths about the mutual exclusion problem, *Inform. Process. Lett.* **12** (1981) 115-116.

[25] Reference manual for the Ada programming language, US Department of Defense (1980).