THE VIRTUE OF PATIENCE: CONCURRENT PROGRAMMING WITH AND WITHOUT WAITING*

James H. Anderson and Mohamed G. Gouda

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-90-23 July 1990

Abstract

We consider the implementation of atomic operations that either write several shared variables, or that both read and write shared variables. We show that, in general, such operations cannot be implemented in a wait-free manner using atomic registers.

Keywords: atomic registers, atomicity, history, interleaving semantics, leader election, shared variable, waiting

CR Categories: D.4.1, D.4.2, D.4.4, F.1.1, F.1.2

1 Introduction

The concept of an atomic register is of fundamental importance in the theory of concurrent programming [11]. An *atomic register* is a shared data object that can either be read or written in a single, indivisible operation. Such a data object is characterized by the number of processes that can write it, the number of processes that can read it, and the number of bits that it stores. The most primitive atomic register can be read by one process, written by one process, and store a one-bit value; the most complicated can be read or written by several processes and store any number of bits.

*Work supported in part by Office of Naval Research Contract N00014-86-K-0763.

The notion of an atomic register lies at the heart of recent research on the wait-free implementation of concurrent shared data objects. The major objective of this research has been to characterize those operations that can be implemented without waiting in terms of single-reader, single-writer, atomic bits. It has been shown in a series of papers that multi-reader, multi-writer, multi-bit atomic registers can be implemented in terms of such bits [4, 5, 10, 11, 12, 14, 15, 16, 17, 18]. In addition, new results by Afek et al. [1] and by Anderson [2, 3] show that it is possible to implement a "snapshot" operation that reads several shared variables in terms of such bits.

In this paper, we consider the other two possibilities: operations that write several shared variables, and operations that both read and write shared variables. We show that, in general, such operations cannot be implemented in a wait-free manner using atomic registers. This result is established by considering the problem of two-process leader election; we call our version of this problem the "binary election problem." The result is proved in two steps. First, we show that the binary election problem can be solved without waiting if a program is allowed to have operations that write several shared variables or that both read and write shared variables. Then, we prove that the binary election problem cannot be solved without waiting by a program with processes that communicate only by means of atomic registers.

The rest of the paper is organized as follows. In Section 2, we present our model of concurrent programs, and in Section 3 we define what it means for a process of a concurrent program to wait. Then, in Section 4, we define the binary election problem and present our main result. Concluding remarks appear in Section 5.

2 Concurrent Programs

In this section, we give our definition of a concurrent program.

Definition: A *concurrent program* consists of a finite set of processes (defined below) and a finite set of variables. □

Definition: A *process* is a finite set of actions. Each action is of the form $B \rightarrow x_1, \ldots, x_m := E_1, \ldots, E_m$ for some $m \geq 1$, where B, called the action's *guard*, is a predicate over private variables (defined below); each x_i is a distinct variable; and each E_j is an expression over a finite number of variables. □

Definition: A variable is *shared* iff it appears among the actions of more than one process; otherwise, it is *private*. For simplicity, we limit our attention to programs with only simple variables, e.g., integers, booleans, and the like. □

Definition: A *state* of a program is an assignment of values to the variables of the program. One or more states of a program are designated as *initial states*. □

Definition: An action is *enabled* at some state iff its guard is true at that state. An enabled action is *executed* by performing its assignment statement. □

Definition: Let s be a state of a program and let c be an action that is enabled at state s. If t is the result of executing action c at state s, then we write $s \xrightarrow{c} t$. We call t a *successor* of state s. □

Definition: A *history* of a program is either an infinite sequence $s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} \cdots$ or a finite sequence $s_0 \xrightarrow{c_0} s_1 \xrightarrow{c_1} \cdots \xrightarrow{c_{k-1}} s_k$ with no enabled action in the final state s_k. In either case, the first state s_0 is either an initial state of the program — in which case the history is called an *initialized history* — or is a state appearing in a history whose first state is an initial state. □

3 Definition of Waiting

In this section, we define what it means for a process of a concurrent program to wait. We also establish a necessary and sufficient condition for a particular kind of program — namely, one that is "supposed" to terminate — to be wait-free.

Definition: A process *waits at* a subset C of its actions iff there exists a history h that contains an infinite number of actions in C such that

- some action in C is enabled at every state of h, and
- starting from each state of h, there exists a history that contains a state in which no action in C is enabled.

We say that a process *waits* iff it waits at some subset of its actions. □

Definition: A program *waits* iff at least one of its component processes waits. □

Definition: A program is called *potentially terminating* iff for each state s in any initialized history there exists at least one finite history that starts from s. □

Theorem 1: A potentially terminating program is wait-free iff all of its histories are finite.

Proof: If a potentially terminating program has only finite histories, then it is clearly wait-free. If, on the other hand, a potentially terminating program has some infinite history h, then there exists a process, call it p, that executes an infinite number of actions in h. Let C denote the set containing all actions of p. Since p executes an infinite number of actions in h (and since the guard of each action is an expression over private variables), p has an enabled

action at every state state of h. Therefore, some action in C is enabled at every state of h. Because the given program is potentially terminating, there is a finite history starting from each state of h. Each finite history has a state in which no action in C is enabled, namely its final state. Therefore, p waits at C, and the given program is not wait-free. □

4 Main Result

In this section, we prove that, in general, actions that write several shared variables or that both read and write shared variables cannot be implemented without waiting using atomic registers. We begin by defining the class of "atomic" programs; this class includes all programs whose processes communicate by means of reading and writing atomic registers or by means of "snapshot" operations, i.e., operations that read several shared variables. (The results of Afek et al. [1] and Anderson [2, 3] show that snapshot operations can be implemented without waiting in terms of reads and writes of atomic registers).

Definition: Consider the action $B \rightarrow x_1, \ldots, x_m := E_1, \ldots, E_m$. We say that this action *reads* each variable appearing in some E_i, and *writes* each variable x_j. □

Definition: A program is *atomic* iff its actions satisfy the following two constraints.

- *Single-Writing*: Each action writes at most one shared variable.
- *Single-Phase*: No action both reads a shared variable and writes a shared variable. □

Thus, if an action of an atomic program accesses shared variables, then it either writes a single shared variable or reads one or more shared variables.

In the remainder of this section, we show that there exists a program violating Single-Writing (respectively, Single-Phase) that cannot be implemented in a wait-free manner by an atomic program. This result is proved by considering the problem of two-process leader election; our version of this problem is called "the binary election problem." The proof is in two parts. We first give two wait-free programs that solve the binary election problem, one that violates Single-Writing and another that violates Single-Phase. We then prove that the binary election problem cannot be solved without waiting by an atomic program. This implies that neither of our proposed solutions can be implemented in a wait-free manner by an atomic program. It follows, then, that actions that violate Single-Writing and Single-Phase cannot, in general, be implemented without waiting in terms of actions that satisfy Single-Writing and Single-Phase. As pointed out in Section 5, similar results have been obtained independently by Herlihy [9].

We now define the binary election problem.

Binary Election Problem: We are required to construct a potentially terminating pro-

process 0

$a \rightarrow X, Y, a, b := 1, 1, \mathit{false}, \mathit{true}$
$b \rightarrow y0, z0, b, c := Y, Z, \mathit{false}, \mathit{true}$
$c \wedge (y0, z0) = (1, 2) \rightarrow \mathit{elect0}, c := \mathit{true}, \mathit{false}$
$c \wedge (y0, z0) \neq (1, 2) \rightarrow \mathit{elect0}, c := \mathit{false}, \mathit{false}$

process 1

$e \rightarrow Y, Z, e, f := 2, 2, \mathit{false}, \mathit{true}$
$f \rightarrow x1, y1, f, g := X, Y, \mathit{false}, \mathit{true}$
$g \wedge (x1, y1) = (1, 2) \rightarrow \mathit{elect1}, g := \mathit{true}, \mathit{false}$
$g \wedge (x1, y1) \neq (1, 2) \rightarrow \mathit{elect1}, g := \mathit{false}, \mathit{false}$

Figure 1: Proof of Theorem 2.

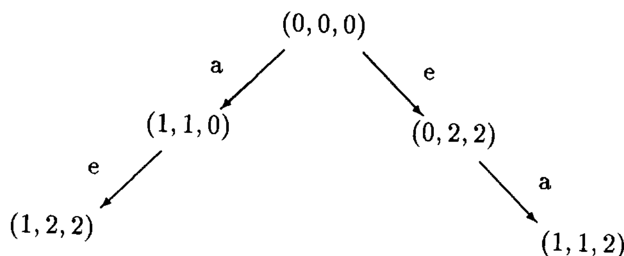

Figure 2: Possible values of (X, Y, Z).

gram consisting of two processes. Each process has a private boolean *decision variable* that it writes only once. The program must satisfy the following two conditions.

- *Equity*: For each process, there exists a finite initialized history with a final state in which the value of that process's decision variable is true.
- *Validity*: In the final state of every finite initialized history, the value of one of the decision variables is true and the value of the other decision variable is false. □

We now show that the binary election problem can be solved without waiting if either Single-Writing or Single-Phase is violated.

Theorem 2: The binary election problem can be solved without waiting by a program that violates Single-Writing.

Proof: Consider the program in Figure 1. We adopt the convention of using upper-case letters for shared variables, and lower-case letters for private variables. Variables X, Y, and Z range over $\{0, 1, 2\}$; each is initially 0. Variables $x1$, $y0$, $y1$, and $z0$ also range over $\{0, 1, 2\}$. All other variables are boolean. Initially, a and e are true, and b, c, f, and g are false. The decision variables are *elect0* and *elect1*.

process 0	**process 1**
$b \rightarrow X, elect0, b := \neg X, X, false$	$c \rightarrow X, elect1, c := \neg X, X, false$

Figure 3: Proof of Theorem 3.

The first action of each process violates Single-Writing. It is easy to check that this program is wait-free. To see that the program satisfies Equity and Validity, consider Figure 2. This figure depicts the possible values of (X, Y, Z); each arrow is labeled by the guard of the action that causes the state change. Based on this figure, we conclude that either the final value of the pair $(y0, z0)$ equals $(1, 2)$ and the final value of the pair $(x1, y1)$ differs from $(1, 2)$, or vice versa. This implies that Equity and Validity are satisfied. □

Theorem 3: The binary election problem can be solved without waiting by a program that violates Single-Phase.

Proof: Consider the program in Figure 3. All variables in this program are boolean. Variables b and c are initially true. The decision variables are *elect*0 and *elect*1. Note that the single action of each process both reads and writes the shared variable X and hence violates Single-Phase. It is easy to see that this program is wait-free and solves the binary election problem. □

In the remainder of this section, we prove that the binary election problem cannot be solved without waiting by an atomic program. Assume, to the contrary, that there exists a wait-free program P that solves the binary election problem and satisfies both Single-Writing and Single-Phase. By the problem definition, P is a potentially terminating program; therefore, by Theorem 1, we can derive a contradiction by showing that P has an infinite history. In the remainder of this section, we prove that there exists an infinite history of P in which neither process ever decides on a value. The proof is similar in structure to the impossibility proof of [8].

We denote the two processes of P as "process 0" and "process 1." The following definition is adopted from [8].

Definition: A state of P is called ***0-valent*** (respectively, ***1-valent***) iff each history starting from that state ends with a final state in which the value of process 0's (respectively, process 1's) decision variable is true. A state of P is called ***bi-valent*** iff it is neither 0- nor 1-valent. □

By the Equity and Validity conditions, the initial state of P is bi-valent. Therefore, we can show that P has an infinite history by proving that each bi-valent state has a bi-valent successor. Before proving this result, we first state and prove two lemmas. The following

definitions are used in the proof (the first is adopted from [6]).

Definition: Two states of P are *compatible* iff it is not the case that one is 0-valent and the other is 1-valent.

□

Definition: Suppose that $s \xrightarrow{c} t$ and $s \xrightarrow{d} u$, where $e \neq f$. Then, the two actions c and d *commute from state* s iff there exists a state v such that $s \xrightarrow{c} t \xrightarrow{d} v$ and $s \xrightarrow{d} u \xrightarrow{c} v$. □

Let s denote an arbitrary bi-valent state of P, and let R (respectively, S) denote the set of successors of s that are reached by executing an action of process 0 (respectively, process 1).

Lemma 1: Both R and S are nonempty.

Proof: By symmetry, it suffices to prove that R is nonempty. Because, by assumption, the guard of each action of P is an expression over private variables, we have the following: for every pair of consecutive states in any history, if a process has no enabled action in the first state, then it has no enabled action in the second state. Thus, it suffices to prove that an action of process 0 is executed in some history starting from state s.

Because s is bi-valent, there exist two histories h and h', both of which start from s, such that the value of process 0's decision variable is true (false) in the final state of h (h'). Thus, because process 0 writes its decision variable only once, it updates its decision variable in each of h and h'.

□

Lemma 2: Suppose that $s \xrightarrow{c} t$ and $s \xrightarrow{d} u$, where c and d are actions of processes 0 and 1, respectively. Then, t and u are compatible.

Proof: We first dispose of the case in which c and d commute from s. In this case, there is a state v such that $s \xrightarrow{c} t \xrightarrow{d} v$ and $s \xrightarrow{d} u \xrightarrow{c} v$. If v is bi-valent, then t and u are both bi-valent, and hence are compatible. If v is 0-valent (1-valent), then neither t nor u is 1-valent (0-valent), and thus t and u are compatible.

For the remainder of the proof, assume that c and d do not commute from s. Because P satisfies Single-Writing and Single-Phase, by symmetry, we only need to consider the following case: c writes some shared variable x and accesses no other shared variable; d either reads x and writes no shared variable, or writes x and accesses no other shared variable.

Consider the state v such that $u \xrightarrow{c} v$. Note that the local state of process 0 is the same in both states t and v. Also, because P is wait-free, by Theorem 1, each history starting from state s is finite. Therefore, starting from each of t and v, we can inductively construct a history consisting only of actions of process 0 such that in both histories, the sequence

of local states of process 0 is the same, and there is no enabled action of process 0 in the last state (i.e., process 0 terminates). It follows that process 0 assigns the same value to its decision variable in both histories, implying that t and u are compatible. □

Theorem 4: The binary election problem cannot be solved by a program that satisfies both Single-Writing and Single-Phase.

Proof: We prove that the arbitrary bi-valent state s, defined above, has a successor that is bi-valent. This implies that, contrary to our assumption, P has an infinite history, and hence is not wait-free. Since s is bi-valent, by Lemma 1, it has at least one successor. If s has exactly one successor (in which case R and S are identical), then that successor is obviously bi-valent, so assume that s has at least two successors. If all successors of s are compatible, then at least one of them is bi-valent, since all of them cannot be 0-valent or 1-valent. If there exist two successors of s that are not compatible, then, by Lemma 2, both are either in $R-S$ or in $S-R$. In the former case, since $R-S$ contains both a 0-valent state and a 1-valent state, by Lemma 2, each state in S is bi-valent. Similarly, in the latter case, each state in R is bi-valent. Therefore, because R and S are both nonempty, the theorem follows. □

5 Concluding Remarks

We have shown that, in general, actions that violate Single-Writing or Single-Phase cannot be implemented without waiting in terms of actions that satisfy Single-Writing and Single-Phase. This implies that operations that write several shared variables, or that both read and write shared variables, cannot be implemented without waiting in terms of atomic registers. Our results imply that most interesting atomic primitives — e.g., test-and-set, fetch-and-add, etc. — cannot be implemented in a wait-free manner using atomic registers.

Similar results have been obtained independently by several other researchers, including Chor, Israeli, and Li [7], Herlihy [9], and Loui and Abu-Amara [13]. Herlihy, in particular, provides a thorough comparison of some of the more well-known atomic primitives that fall within the class of actions violating Single-Writing and Single-Phase. He also shows that these primitives cannot be implemented in terms of atomic registers. Perhaps the most important contribution of Herlihy is a general means for proving that a primitive is universal, i.e., that it can be used to construct a wait-free implementation of any "sequential" data object.

Acknowledgements: We would like to thank Ken Calvert and Ambuj Singh for their comments on this paper.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, to appear.

[2] J. Anderson, "Composite Registers," *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, to appear. Also, available as Technical Report TR.89.25, Department of Computer Sciences, University of Texas at Austin, 1989.

[3] J. Anderson, "Multiple-Writer Composite Registers," Technical Report TR.89.26, Department of Computer Sciences, University of Texas at Austin, 1989.

[4] B. Bloom, "Constructing Two-Writer Atomic Registers," *IEEE Transactions on Computers*, Vol. 37, No. 12, December 1988, pp. 1506-1514.

[5] J. Burns and G. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[6] K. Chandy and J. Misra, "On the Nonexistence of Robust Commit Protocol," unpublished manuscript, Department of Computer Sciences, The University of Texas at Austin, November 1985.

[7] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware," *Principles of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 86-97.

[8] M. Fischer, N. Lynch, and M. Patterson, "Impossibility of Distributed Consensus with one Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, April 1985, pp. 374-382.

[9] M. Herlihy, "Wait-Free Implementation of Concurrent Objects," *Proceedings of the Seventh Annual Symposium on Principles of Distributed Computing*, 1988.

[10] L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register," Proceedings of the Second International Workshop on Distributed Computing, Springer Verlag Lecture Notes in Computer Science 312, 1987, pp. 278-296.

[11] L. Lamport, "On Interprocess Communication, Parts I and II," *Distributed Computing*, Vol. 1, 1986, pp. 77-101.

[12] M. Li, J. Tromp, and P. Vitanyi, "How to Construct Wait-Free Variables," unpublished manuscript, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.

[13] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, JAI Press, 1987, pp. 163-183.

[14] R. Newman-Wolfe, "A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[15] G. Peterson and J. Burns, "Concurrent Reading While Writing II: The Multi-Writer case," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.

[16] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register, Revisited," *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221.

[17] J. Tromp, "How to Construct an Atomic Variable," unpublished manuscript, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.

[18] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 233-243.