Constructing Two-Writer Atomic Registers

Bard Bloom*
M.I.T. Laboratory for Computer Science
545 Technology Square, Cambridge, Mass. 02139
ARPAnet: bard@theory.lcs.mit.edu

May 15, 1987

Abstract

In this paper, we construct a 2-writer, n-reader atomic memory register from two 1-writer, $(n+1)$-reader atomic memory registers. There are no restrictions on the size of the constructed register. The simulation requires only a single extra bit per real register, and can survive the failure of any set of readers and writers. This construction is a part of a systematic investigation of register simulations, by several researchers.

1 Introduction

There are several models of memory registers, of varying strength. The most familiar is the single-processor memory register, as used by isolated computers. The intuition of a single-processor register is clear enough: a read of the register returns the value written by the last write to the register. We call this the *register property*. If the processor was initialized (equivalently, the first action on the register is a write), this property uniquely determines the behavior of the register, as a function of the sequence of reads and writes. The object of strong shared memory, atomicity in particular, is to provide as much of the power and familiarity as possible of single-processor memory to the multiprocessor environment.

Consider a model of memory in which each processor has a separate channel to each shared memory register.[1] We assume that the processors using the register are each sequential, but completely asynchronous. Since there are several processors, the register may be trying to do more than one thing at a time. When a write overlaps some other action, the register property does not uniquely specify the register's behavior. Which of two overlapping writes, for example, should be considered "the last write"? If a read overlaps several writes, which is the last one? These questions cannot be answered from first principles.

Fortunately, the register property does tell what some actions should do. Given an action A, with no action overlapping it, then A should work correctly. If R is a read, and only reads overlap R, then R should return the correct value. The register should only contain legitimate values; a boolean-valued register can't return fue or its complement tralse.

There are several standard models for one-writer registers[L2]. All of these models work correctly when only one processor is using a given memory register at a time; the differences between models restrict how the register can react when several processors are acting on it at once.

The strongest and most commonly used of these models is *atomicity*. Reads and writes act as if they do not overlap, as if they occurred in some definite order. The register property and this ordering of actions determines the effect of every action.

Atomic runs are helpful for practical purposes. Since all actions act as if they did not overlap, we need not worry about what happens on overlaps. For example, it is not possible for one reader, reading twice during a write, to get the new value on the

*This research is supported in part by the Defense Advanced Research Projects Agency under Contract N00014-80-C-0622, and in part under an NSF Graduate Fellowship.

[1] Few if any multiprocessors have this architecture.

first read and the old on the second; this sequence of events is impossible for non-overlapping actions, and therefore impossible for overlapping ones as well.

Since shared memory of this sort is expensive, it seems reasonable to try to simulate it with cheaper kinds of shared memory. (In general, memory from a weaker model of computation is cheaper than from a stronger.)

Register protocols are unusual among resource-sharing protocols, in that they provide atomicity (*i.e.*, serialization) without requiring mutual exclusion. Although they are usually phrased in terms of memory, they can be used for any object with the same fundamental operations of reading and writing. For example, consider a collection of computers, each permitted to read all the others' file systems, but only able to write on their own. Multi-writer register algorithms could allow them to simulate a shared file system. Leslie Lamport [L2] has given many simulations for single-writer memory. In this paper, I present a protocol for simulating two-writer n-reader atomic memory with two one-writer, $n+1$-reader atomic registers.

There are a number of properties that a good simulation will have. No processor should have to wait for another's action to use a register. The failure of one processor — even during a write — should not prevent other processors from reading or writing.[2] Shared memory is likely to be slow compared to local memory, so the algorithm should use the smallest possible number of accesses of shared memory.

2 I/O Automata (Simplified Lynch-Tuttle Model)

In this paper, we will use a considerably simplified version of the Lynch-Tuttle I/O automaton model. The full model is considerably more powerful than required by register algorithms; see [LT] and [LM]. Most of the detail available in our restricted version is not used *per se* in the proofs; it is presented to give a flavor of a fully formal proof in this model. A process (*e.g.*, a program or a register) is modeled as an automaton, with possibly an infinite number of states and infinite fanout from any state. The automaton may be countably nondeterministic. The transitions of the automaton A are labeled with ***actions***, members of the automaton's ***alphabet***.

The automaton alphabet is divided into three subalphabets, the Input, Output, and Internal alphabets. The Input and Output alphabets are sets of signals which the automaton can accept and produce. The Internal alphabet is the set of actions which other processors should not be allowed to see. We will use Internal actions to mark times at which register events actually take place. We insist that an I/O automaton be ***input-enabled***; *i.e.*, have an edge labeled with each input action out of every state. Thus, an automaton is always ready to deal, perhaps trivially, with any input. It may be programmed to buffer the input, by changing to a state with the input added to a queue; it may be programmed to ignore inputs that it is not ready to pay attention to.

Automata may be composed into systems in the following way. If $A_1, \ldots, A_n$ are automata with disjoint sets of output actions and disjoint sets of internal actions, their composition B has as its set of states the set of tuples of states of the A_i's. B has a state transition from $\langle a_1, \ldots, a_n \rangle$ to $\langle a'_1, \ldots, a'_n \rangle$ with action α if either one component has a so-labeled transition and no other component moves (in which case α has the same classification for B that it does for that component), or one component has α as an input action and another has it as an output action, and no other component takes a step (in which case α is internal to B).

An *execution* of an I/O automaton is an alternating sequence of states and actions, starting with an initial state of the automaton and proceeding as long as possible (which may be infinite), such that whenever $s_1 a_1 s_2$ is a subsequence starting with a state, the automaton can make an a_1-labeled transition from s_1 to s_2. A *fair execution* of the composition B is one in which, whenever a component A_i wants to take a step, it is eventually allowed to. More formally, whenever the system is in a state $\langle a_1, \ldots, a_n \rangle$ and A_i has an output or internal action enabled in state a_i, eventually the composition B takes one such action. A *schedule* of an automaton is a sequence of actions obtained by removing all the states from an execution; it is a finite or infinite sequence of actions taken by the automaton as it moves from state to state. A *fair schedule* is a schedule derived from a fair execution. Fair schedules correspond to the usual notion of asyn-

[2] These requirements disallow mutual exclusion algorithms, which delay some processes in favor of others and are not in general crash-resistant.

chronous communication: one process may take arbitrarily but finitely many steps before another takes one.

An *execution module*, for our purposes, is a set E of executions. A *schedule module*, is a set S of schedules. Also of interest are *external schedules*: for each schedule s, the subsequence of s formed by omitting all the internal actions of s. In particular, a protocol (for an arbitrary problem) is considered correct if it has an appropriate set of external fair schedules.

We will talk about I/O channels between automata. A channel that can pass signals (*i.e.*, actions) in the set S between automata A and B is simply a convention that A and B share the actions in S, that the actions in S are internal to the composition, and that no other processors in the system have actions from S in their signature.

3 Formal Model

Let Val be a set of values that the register is to hold.

A register can be described by a schedule module. Each reader and writer has a bidirectional channel to the register; if a computer can both read and write, it has two channels. A read channel allows messages of the form $\mathsf{R}^c_{\text{start}}$ (a constant signal meaning a command to read from channel c) to the register, and $\mathsf{R}^c_{\text{finish}}(v)$ (a meaning that the value $v \in \mathsf{Val}$ was read) to the reader. A write channel allows messages $\mathsf{W}^c_{\text{start}}(v)$ (command to write) and $\mathsf{W}^c_{\text{finish}}$ (acknowledgment). (The channel c names the source and destination; for example, $\mathsf{R}^{\mathsf{Wr}_0,1}_{\text{start}}$ is a request from processor Wr_0, writer number 0, to read the value in register 1.) A process equipped with such channels and no others is said to *have the signature of a register*.

The input—the sequence of read and write requests—is correct if no reader or writer initiates a second action before the first has finished. In our formalism, a sequence α of actions *is input-correct* if there are no two requests on that channel without an intervening acknowledgment. A non-input-correct schedule is one in which case the user has used the interface to the register improperly and so any behavior by the register is legitimate.

For $v_0 \in \mathsf{Val}$, we say that a schedule α of a system of automata (with the signature of a register) is *atomic initialized to* v_0 if either it is not input-correct, or the following conditions hold.

1. There is a bijection ("matching") between the requests and acknowledgments along each channel, such that the acknowledgment corresponding to a given request is the first action along that channel following the request.

2. The reads and writes in α can be shrunk to points. Formally, α can be extended to a sequence β, by the addition of signals $\mathsf{R}^c_\star(v)$ and $\mathsf{W}^c_\star(v)$ inside the $\mathsf{R}^c_{\text{start}}$-$\mathsf{R}^c_{\text{finish}}(v)$ and $\mathsf{W}^c_{\text{start}}(v)$-$\mathsf{W}^c_{\text{finish}}$ pairs, precisely one signal per pair, such that, for each matched pair $\mathsf{R}^c_{\text{start}}$-$\mathsf{R}^c_{\text{finish}}(v)$, v is the value of the latest $\mathsf{W}^c_\star(v')$ preceding the $\mathsf{R}^c_\star(v)$, or v_0 if there is no such $\mathsf{W}^c_\star(v')$.

The signals $\mathsf{R}^c_\star(v)$ and $\mathsf{W}^c_\star(v)$ mark the instants that the actions "actually occurred." Such actions are called $\star$-actions or internal actions.

If $\mathcal{R}$ is a system of automata, such that every fair schedule of $\mathcal{R}$ is atomic initialized to v_0, then we say that $\mathcal{R}$ implements an atomic register with initial value v_0. $\mathcal{R}$ is *atomic* if it is atomic initialized to some v_0. (As is usual in the Lynch-Tuttle model, the requirement that an automaton be input-enabled excludes degenerate solutions by forcing a system to have appropriate behavior for every sequence of inputs. In particular, degenerate cases such as an automaton which never does anything are excluded.)

This definition is a formalization in terms of sequences of the usual definition ([L2]): that every read and write can be shrunk to a point inside its interval, with distinct actions shrinking to distinct points, such that the resulting sequence has the register property.

We will use the term "a read" to refer to a matched $\mathsf{R}^c_{\text{start}}$-$\mathsf{R}^c_{\text{finish}}(v)$ pair, and similarly for a write. An atomic sequence may be considered a set of reads and writes, partially ordered by precedence. Lamport [L2] among others uses this formalism.

4 Architecture of the Solution

We present a protocol for simulating a two-writer, n-reader atomic register with two one-writer, $n+1$-reader atomic registers. Both the simulated and the real[3] registers are defined as in the previous section.

[3] It is somewhat deceptive to call the registers used in the simulation "real". They may be simulated using more primitive regular and safe one-reader, one-writer registers, using protocols from Lamport [L2] and others. However, they are as real as anything at this level of abstraction.

$\mathbf{R}^c_{\text{start}}$	Command to read.
$\mathbf{R}^c_\star(v)$	Internal (to simulated register) event marking a read of v.
$\mathbf{R}^c_{\text{finish}}(v)$	Read acknowledgment; Communication of the read value v to the reader.
$\mathbf{W}^c_{\text{start}}(v)$	Command to write value v.
$\mathbf{W}^c_\star(v)$	Internal event marking a write of v.
$\mathbf{W}^c_{\text{finish}}$	Acknowledgment of a write.

Figure 1: Actions of a Register Automaton

The architecture of the simulated register is as follows; see also Figure 2.

There are $n+4$ automata in the simulated register. Two of these automata, Reg_0 and Reg_1, are 1-writer, $(n+1)$-reader atomic registers. Two others are the writers, called Wr_0 and Wr_1; the rest are the readers, named Rd_1 through Rd_n. Each of the readers and writers has one channel to the outside world; these channels are the ports of the simulated register. The readers and writers are collectively known as processors. Each processor has a channel to each of the real registers. The channels allow reading and writing messages as appropriate: The readers can read both real registers; Wr_i can write to Reg_i and read (but not write) $\mathsf{Reg}_{\neg i}$. The external ports of the system give it the signature of a register. The problem solved in this paper is to find implementations for the $(n+2)$ processes Wr_0, Wr_1, and Rd_1 through Rd_n such that the system actually is an atomic register.

In practice, the reader and writer automata will be subroutines running on real processors. The requests are the calls to the subroutines; the acknowledgments are the returns.

Certain other properties are desirable. The reading and writing protocols should be deterministic. Also, the status of one processor should not affect that of another. For example, a protocol could be cobbled together from a fair mutual exclusion protocol. This would require processes to wait for each other, an undesirable trait for memory. Furthermore, one processor could crash while reading the register and block all further access, which is rather undesirable. Finally, the algorithm should use as little extra memory as possible. The algorithm presented in this paper has all these properties.

5 The Algorithm

To simulate an atomic register with values in Val, we use registers Reg_0 and Reg_1 with enough space to hold one value in Val and a single *tag bit*. To construct a register initialized to v_0, use two real registers both initialized to value v_0 and tag bit 0.[4] Whenever a writer writes, it tries to make the sum of the tag bits equal to its own index, modulo 2. This is similar to Peterson's tournament algorithm [PF]. If one writer is quiescent while the other writes, it is clear that the active writer can set the sum of the tag bits to its own index.

The writers use the following code to write the value v. Each writer knows its own identity, $i = 0$ or $i = 1$. If i is the index of a processor, $\neg i$ (the complement of i) is the index of the other processor. The symbol $\oplus$ denotes addition modulo 2.

> **read** t', v' **from** $\mathsf{Reg}_{\neg i}$
> $t := (i \oplus t')$
> **write** t, v **to** Reg_i

We say that a simulated write request $\mathbf{W}^i_{\text{start}}(v)$ occurs at the call to this routine, and an acknowledgment at the return. Since the routine is running on a sequential processor, there can never be two write requests without an intervening acknowledgment; the input will always be input-correct. By hypothesis, the real reads and writes always terminate; since the write routine has no loops, it too will always terminate.

Notice that the writer writes only once, at the end of its protocol. This has the advantage that, at any time except for the instant of the atomic write, either nothing of the write is visible or everything is. In particular, if the writer crashes at some point in the protocol, the write either occurs or does not occur;

[4]The initial value — but not the initial tag bit — of Reg_1 is irrelevant.

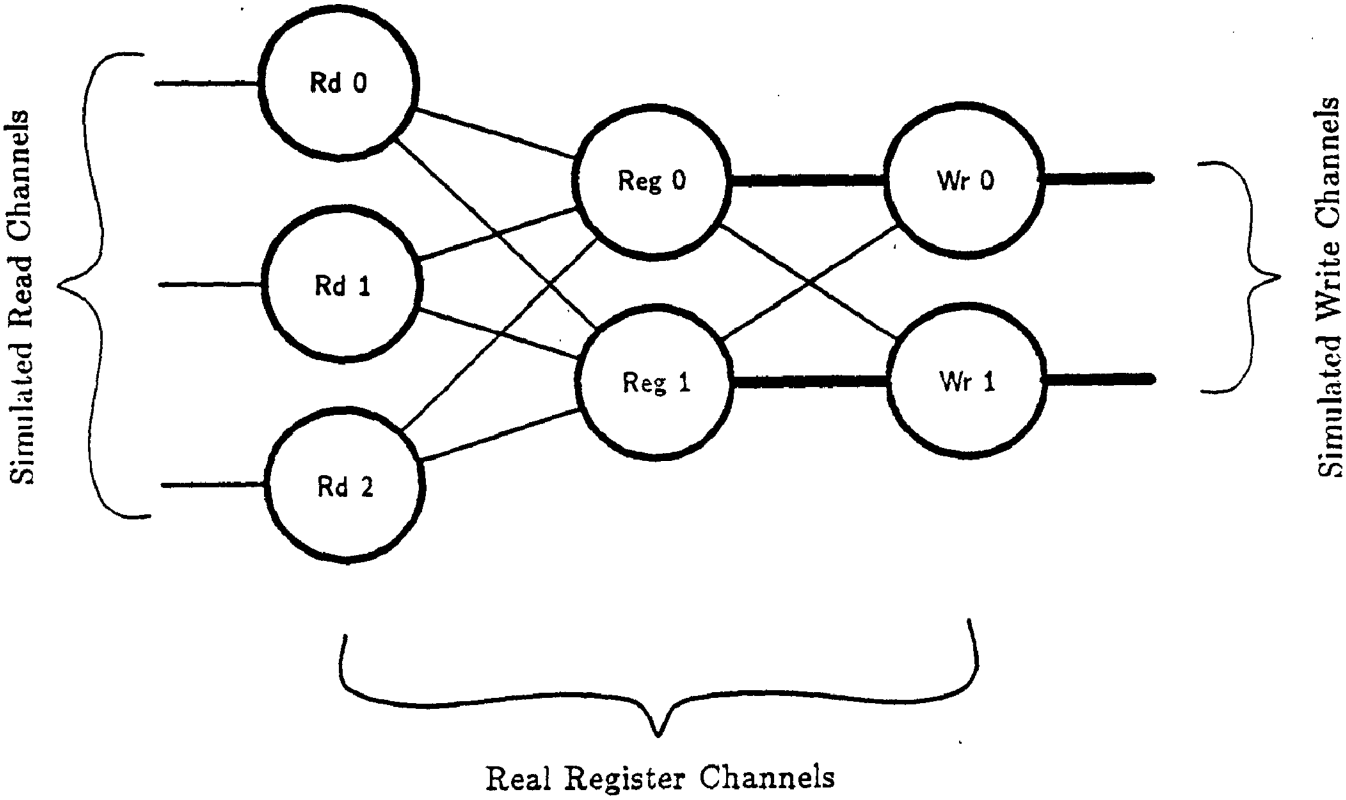

Figure 2: Architecture of the Simulated Register

it does not leave the register in an inconsistent state. (Dealing with failures formally would complicate the model slightly, and we shall not do so.)

The readers use the following code:

> **read** t_0, v_0 **from** Reg_0
> **read** t_1, v_1 **from** Reg_1
> $r := (t_0 \oplus t_1)$
> **read** t_2, v_2 **from** Reg_r
> **return** v_2

As was the case for writes, simulated read requests are defined to occur at the start of this code, and their acknowledgments at the end. Notice that one simulated read involves *three* real reads. In many applications, the writers will be allowed to read the simulated register as well; that is, a single automaton is connected to one read port and one write port. The number of real reads that such a writer performs in a simulated read may be reduced to one or two by having the writer keep a local copy of its own real register.

6 Correctness

Consider any fair external schedule α of the system. Since there are no requirements on a schedule that contains a violation of sequentiality on any channel, we assume that α has no such violations. We will denote the actions by processor P on the simulated register by $\mathbf{R}^{P,s}_{start}$ etc.; those on real register i will be $\mathbf{R}^{P,i}_{start}$. We must show that α is atomic by inserting $\star$-actions. We omit the P superscript when it is clear from context.

Since α is the external schedule of the system, it is the schedule resulting from some execution sequence α^*, including the states and real actions of the implementing I/O automata. We will not need to consider the states of the processors formally, so we will use the sequence β consisting of the actions in α^* for the real as well as the simulated registers.

The real registers are atomic, and therefore any schedule of them can be extended by actions $\mathbf{R}^{c,i}_{\star}(v)$ and $\mathbf{W}^{c,i}_{\star}(v)$ as given by the definition of atomicity. Extend β to a sequence γ with these internal actions included.

If R and W are simulated read and write respectively, we say that R *reads the value written by* W if

W is a write by Wr_i, R's final real read reads $\mathbf{Reg}_i$, and the $\star$-action of W's real write is the last $\star$-action of any real write to $\mathbf{Reg}_i$ in γ before the $\star$-action of R's final real read. *R reads the initial value* if it does not read the value written by any write W.

A nonempty finite prefix γ' of the schedule γ is a listing of the history of events that have happened up to and including some point; we refer to such a prefix as a *time*. We write $\gamma_1 < \gamma_2$ for "γ_1 is a proper prefix of γ_2". (Unlike other notions of time, only one action can happen at a time — which is to say, there is only one action at the end of a nonempty finite sequence of actions.)

Let γ' be a time. We say that Reg_i *contains* $\langle v_i, t_i\rangle$ *after* γ' if the last $\mathbf{W}^i_\star(\langle v'_i, t'_i\rangle)$ in γ' has $t_i = t'_i$ and $v_i = v'_i$, or if no such action exists and $\langle v_i, t_i\rangle$ is the initial value for Reg_i. If the real registers contain $\langle v_0, t_0\rangle$ and $\langle v_1, t_1\rangle$ after γ' respectively, then we say that the sum of the tag bits after γ' is $t_0 \oplus t_1$.

Since the real registers are atomic, we will speak of the $\star$-actions of real register accesses as if they were the whole access. For example, we say "The real write W_0 precedes the real read R_1" for "The $\star$-action of W_0 precedes the $\star$-action of W_1." When we have $\star$-actions defined for simulated register accesses, we will speak of them similarly. Also, if A and B are distinct register accesses by the same processor (with or without $\star$-actions), one of them entirely precedes the other and we will speak of them as such. We say "Wr_i real-reads at time T" for the more precise "Wr_i performs a real read with its $\star$-action occurring as the last element of the prefix T of γ", and use similar language for real-reading and simulated-writing.

7 Proof of Correctness

We will insert internal actions $\mathbf{R}_\star^{\mathsf{Rd}_{i,s}}(v)$ and $\mathbf{W}_\star^{\mathsf{Wr}_{i,s}}(v)$ in the sequence γ in several steps. We first consider only the writes, divided into "potent" and "impotent" writes; then we consider the reads, divided into "reads of potent writes", "reads of impotent writes", and "reads of the initial value." In each step, we insert a $\star$-action between the start and finish of each simulated register access of the appropriate type. Furthermore, for each read R we show that the value returned by R is the value written by the immediately preceding $\star$-action of a simulated write.

At each stage, we will be inserting a possibly infinite number of actions into a possibly infinite sequence. It is essential to know that the insertions can be done, and that the result is a sequence containing all the actions in the original; consider adding an infinite number of actions at the front of γ. However, we will always add actions between pairs of actions in the original sequence; therefore we will never add more than n elements before the n'th element of the sequence. This is sufficient to guarantee that the addition of elements is well-defined, and is in fact results in a sequence containing all the elements of the first in the correct order.

We begin with definitions and useful lemmas. We say that a simulated write W by Wr_i is *potent* if the sum of the tag bits immediately after the $\star$-action of W's real write is i. Otherwise, it is impotent. Note that the potency of a write depends only on the values in the real registers immediately after the write; a write is not potent at one point in γ and impotent at another.

We say that one simulated write W_0 *is prefinished by* another W_1 if the real write of W_1 occurs between the real read and the real write of W_0, and (3) W_1 is the last such write.

Lemma 1 *Every impotent write W_0 is prefinished by precisely one write W_1.*

Proof: Uniqueness of the W_1 follows from the definition. Let W_0 be a write by Wr_0 not prefinished by any other write. Then, there is no real write by Wr_1 between the start and the real write of of W_0. In particular, the tag bit t_1 of $\mathbf{Reg}_1$ has the same value when Wr_0 real-reads it and when Wr_0 real-writes to its register. Following the protocol, Wr_0 chooses the bit $t_0 = t_1$, and writes t_0 as the tag bit of its real write. So, the tag bits at the time of W_0's real write are $t_0 = t_1$ in $\mathbf{Reg}_0$ and t_1 in $\mathbf{Reg}_1$; their mod-2 sum is 0, and so W_0 is potent. The same argument applies, mutatis mutandis, for Wr_1. Thus, any write not prefinished by another is potent, *i.e.*, any impotent write is prefinished by some write. □

Definition 1 *The* prefinisher *of an impotent write is the unique write which prefinishes it.*

Lemma 2 *The prefinisher W_1 of an impotent write W_0 is potent.*

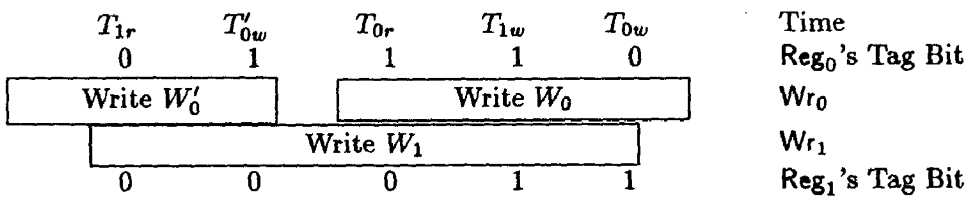

Figure 3: Figure for Lemma 2

Proof: Suppose that the lemma is false. Let W_0 be the first impotent write with impotent prefinisher, in order of time of atomic real write. Let W_1 be the prefinisher of W_0. W_1 is impotent; let W'_0 be W_1's prefinisher. Let T_{0r} and T_{1r} be the times of W_0's and W_1's real reads, T_{0w}, T'_{0w} and T_{1w} the times of the real writes of W_0, W'_0, and W_1 respectively. By the definition of prefinishing, $T'_{0w} < T_{1w} < T_{0w}$; since the processors are sequential we have $T'_{0w} < T_{0r} < T_{0w}$ and $T_{1r} < T_{1w}$. (See Figure 3.)

We assume that Wr$_0$ is the writer of W_0, exploiting the symmetry of the protocol. Further, suppose that Reg$_0$'s tag bit after the real write W_0 is 0; the same argument with 0 and 1 exchanged applies if the bit is 1.

Since Wr$_0$ writes the tag bit value 0, it must read Reg$_1$'s tag bit set to 0 at T_{0r}. Since W_0 is impotent, Reg$_1$'s tag bit at time T_{0w} must be 1. Since Reg$_1$'s tag bit changes between T_{0r} and T_{0w}, Wr$_1$ must real-write between these times. W_1's real write is the last real write by Wr$_1$ before T_{0w}, and therefore $T_{0r} < T_{1w} < T_{0w}$.

Wr$_1$ does not write between T_{1w} and T_{0w}; since Reg$_1$'s tag bit at T_{0w} is 1, its tag bit at T_{1w} is 1. Since W_1 is impotent, the sum of the tag bits must be 0 at time T_{1w}. Hence the tag in Reg$_0$ at time T_{1w} is 1. Since W_1 writes the bit 1, it must read Reg$_0$'s tag bit as 0 at time T_{1r}.

However, Reg$_0$'s tag bit is 1 at time T_{1w}. Since W'_0 is the prefinisher of W_1, its real write is the last real write to Reg$_0$ before T_{1w}; it must write 1 as its tag bit. In particular, Reg$_0$'s tag bit is 1 at T'_{0w}. Since Wr$_0$ does not write between T'_{0w} and T_{0w}, Reg$_0$'s tag bit is constantly 1 in this interval. However, W_1 real-reads tag bit 0 at time T_{1r}; therefore $T_{1r} < T'_{0w}$. The five times are now fully ordered $T_{1r} < T'_{0w} < T_{0r} < T_{1w} < T_{0w}$ as in Figure 3.

Since Reg$_1$'s tag bit is 0 at time T_{0r}, it must be 0 at time T_{1r}; the tag bit is not changed until the real write occurs. In particular it is 0 at time T'_{0w}, when W'_0 real-writes.

At time T'_{0w}, Reg$_0$ has tag 1 and Reg$_1$ has tag 0; therefore, W'_0 is impotent. However, we had assumed that W_0 is the first (ordered by time of real write) impotent write by either writer not prefinished by some potent write by the other processor; and we have discovered that W_1 is another such write that precedes W_0 in this order. This contradiction proves the lemma for Wr$_0$ and tag bit 0; the other cases are similar. □

At this point we have divided the simulated writes into two categories: potent and impotent writes. There are three categories of reads: those which read from potent writes, from impotent writes, and from the initial value. We will insert $\star$-actions for actions in stages in this order; for brevity, we will process all writes at the same time.

7.1 Writes

Step 1 *Let γ_1 be γ with a $\star$-action for each potent write inserted immediately after the $\star$-action of its real write, and a $\star$-action for each impotent write W_0 placed immediately before the $\star$-action in γ_1 of the write W_1 which prefinishes it.*

It is clear that $\star$-actions of potent writes are within the intervals of the writes. If W_0 is an impotent write, its prefinisher W_1 is potent. W_1's $\star$-action is adjacent to its real write in γ_1. By definition of prefinishing, W_1's real write is between W_0's real read and real write. Therefore, W_1's $\star$-action is inside W_0's interval as well, and so this is a legitimate assignment of times.

7.2 Reads of Potent Writes

It is evident from the code that a read always returns a value written by some write, or the initial value. It may read the value of an impotent write. Consider a very slow reader, which reads the tag bits and then goes to sleep for a long time while the writers continue to work. When it wakes up, its tag bits have no relevance to the current state of the register, and it may read from either real register, and so return the value of an impotent write. This is acceptable behavior; in step 4, we will assign such a read a time immediately after the $\star$-action of the impotent write.

We will first choose times for all simulated reads R which return values v written by *potent* writes W.

Step 2 *Let γ_2 be γ_1 with a $\star$-actions for each read R of a potent write W inserted immediately after the later of the $\star$-action for the first real read of R and the $\star$-action of W in γ_1. If several $\star$-actions are to be inserted in the same position* (e.g., *immediately after the $\star$-action of a particular write W), we insert them in arbitrary order.*

It is clear that we are assigning $\star$-actions to reads within the intervals of the reads. Since the reads by a given reader are sequential, no more than n $\star$-actions will have to be inserted in one position, and therefore γ_2 is well-defined.

Lemma 3 *If R is a read of a potent write W, then the $\star$-action of W in γ_2 precedes that of R, and is the last $\star$-action of any write preceding the $\star$-action of R.*

Proof: The identity Rd_k of the reader makes no difference to the argument; all readers have exactly the same algorithm. There are three real reads over the course of R: R_0 and R_1 of Reg_0 and Reg_1, and R_2 of one of the two registers. Let T_0, T_1, and T_2 be the times in γ_2 of these three actions; let T_w be the time in γ_2 of W's $\star$-action. R's $\star$-action is the action immediately after the later of T_w and T_0. Note that R_0 precedes R_1 precedes R_2.[5] There are two cases, depending on the relative order of T_0 and T_w.

The first alternative is that $T_0 < T_w$. Since R_2 returns the value written by the real write of W, R_2 must follow the real write. Since W is potent, the $\star$-action of W immediately follows its real write. By construction, the $\star$-action of R follows that of W, with only $\star$-actions of other simulated reads intervening.

The other alternative is that $T_0 > T_w$. Suppose that Wr_0 is the writer of W. Since W writes v at time T_w and R_2 reads v at position $T_2 > T_w$, Wr_0 did not real-write between T_w and T_2. Therefore, it did not perform a potent simulated write between these times. If it performed an impotent write assigned a $\star$-action in the interval $(T_w \ldots T_0)$, the prefinisher of that write would also be in that interval (since the times of such writes are adjacent in γ_2 by construction). It suffices to show that Wr_1 does not write in this interval.

Suppose that Wr_1 performs a write W' of value v' after W with W''s $\star$-action before R_0. By symmetry, assume that W set Reg_0's tag bit to 0. Then, at time T_w, Reg_1's tag bit must be 0 as well, because W is potent.

If W' were impotent, then W' had to be prefinished by some write by Wr_0. If W prefinished W', then W' would have been assigned a time immediately before the time for W; this is not the case. If some later write W'' prefinished W', then we would have assigned W' a time immediately before W'' in γ_1. The $\star$-action of W'''s is next to its real write in γ_1. Since W''s $\star$-action is between T_w and T_0; therefore, W'''s real write is in the same interval. This contradicts the fact that Wr_0 does not real-write between T_w and T_2. So, W' cannot be impotent.

Therefore W' must be potent; Wr_1 must have set Reg_1's tag bit to 1 at the real write of W'. Since W' is potent, its $\star$-action is adjacent to its real write in γ_1, and hence the tag bit is set to 1 before R_0. Since Wr_0 did not real-write between T_w and T_2, its tag bit is 0 over that interval. Any simulated write by Wr_1 after W' and before T_2 will real-read Reg_0's tag bit of 0 and write 1 as Reg_1's tag bit. So, the tag bits are 0 and 1 from the time of W' until T_2. In particular, the bits are set to 0 and 1 when the reader reads them at T_0 and T_1. But then the reader would have read Reg_1 instead of Reg_0, which is a contradiction.

The case that Wr_1 wrote W is essentially symmetric; the preceding argument works mutatis mutandis. Therefore, there are no $\star$-actions of writes between T_w and T_0. □

7.3 Reads of Impotent Writes

Step 3 *Let γ_3 be γ_2 with a $\star$-action for each read R of an impotent write W_0 inserted immediately after the $\star$-action of W_0.*

[5] The proof would work with trivial changes if the reading protocol performed its first two reads in parallel. In this case, R_0 should be defined as the first read to happen in the atomic order on real reads.

It is not immediately obvious that this is a legitimate assignment of $\star$-actions. Consider the scenario of Figure 4, in which the $\star$-action of W_0 precedes the start of R. Fortunately, this scenario is impossible.

Lemma 4 *If R is a read of an impotent write W_0, then the $\star$-action of W_0 occurs within the interval of R.*

Proof: Let Rd_j be the reader of R, and T_0, T_1, and T_2 the times of its real actions. By symmetry assume that Wr_0 performed W_0. Let W_1 be the prefinisher of W_0. Let T_{s0} and T_{s1} be the times in γ_2 of the $\star$-actions of these simulated writes. By construction, $T_0 < T_1 < T_2$ and $T_{s0} < T_{s1}$. Furthermore, there are no actions in γ_2 between T_{s0} and T_{s1}. We show that $T_0 < T_{s0} < T_2$

Since R returns the value was written by W_0, the real read R_2 must precede the real write of W_0. The assignment of times to impotent writes places their internal actions before their real writes occur. So, $T_{s0} < R_2$.

We must now show $T_0 < T_{s0}$. Suppose not; *i.e.*, $T_{s0} < T_0$. Since T_{s0} and T_{s1} are consecutive, $T_{s1} < T_0$ as well. The five times must be ordered $T_{s0} < T_{s1} < T_0 < T_1 < T_2$, as shown in Figure 4.

We observe that W_0 does not change the tag bit of Reg_0. For, since W_1 is potent, the sum of the tag bits at time T_{s1} is 1. Since W_0 is impotent, the sum of the tag bits at T'_{s0} — the time of W_0's real write — is also 1. (The only other thing the sum could be is 0, but if it were 0 then W_0 would be potent). Since the sum of the bits does not change after W_0 writes, neither does Reg_0's bit.

At time T_2, the reader reads the value written by W_0. Therefore, Wr_0 does not real-write between times T'_{s0} and T_2. Wr_1 may write several times in this interval; however, all such writes use the same tag bit as W_1. Reg_0's tag bit does not change in the interval $[T_{s0} \ldots T_2]$, and Reg_1's does not change in the interval $[T_{s1} \ldots T_1]$. Let t_0, t_1 be the tag bits during this interval, which Rd_j reads at times T_0, T_1 respectively.

Since Rd_j following the protocol reads Reg_0, we have $t_0 \oplus t_1 = 0$. By the preceding argument from W_1's potency, $t_0 \oplus t_1 = 1$. This is absurd. This contradiction shows that R_0 precedes T_{s0}; *i.e.*, that the time assigned to W_0 is in the interval of the read R. □

Lemma 5 *If R is a read of an impotent write W_0, then the $\star$-action of W_0 in γ_3 precedes that of R, and is the last $\star$-action of any write preceding the $\star$-action of R.*

Proof: Obvious. □

7.4 Reads of the Initial Value

Lemma 6 *If R is a simulated read of the initial value, consisting of real reads R_0, R_1, and R_2, then there are no real write actions by either processor in γ_3 preceding R_1.*

Proof: Recall that both tag bits are initialized to 0. If R reads from Reg_1, then clearly it reads from a write by Wr_1 rather than the initial value. Suppose that R reads from Reg_0. If there were a real write W_0 by Wr_0 before R_2 (which is later than R_1), then R_2 would return the value from W_0's simulated write rather than the initial value. Since there are no writes by Wr_0 before R_2, Reg_0's tag bit is constantly 0 until at least R_2. If there were a real write by Wr_1 before R_1, then the tag bit of Reg_1 would be set to 1, and R would have read the value written by such a write rather than the initial value. □

Step 4 *Let δ be γ_3 with a $\star$-action inserted immediately after the second real read (R_1 in Lemma 6) of each read R reading the initial value.*

7.5 Concluding the Proof

Let ϵ be δ with the real actions omitted; ϵ is the external schedule of a particular run of the protocol with $\star$-actions of the simulated register accesses inserted. By construction, each $\star$-action $\mathsf{R}^c_\star(v)$ or $\mathsf{W}^c_\star(v)$ is between the appropriate $\mathsf{R}^c_{\text{start}}$-$\mathsf{R}^c_{\text{finish}}(v)$ or $\mathsf{W}^c_{\text{start}}(v)$-$\mathsf{W}^c_{\text{finish}}$ pair. During the construction, we were careful to be sure that each read returned the value written by the write which immediately preceded it (ordered by $\star$-actions), or the initial value if no such write exists. As we have already observed, each call to the subroutines of the protocol returns; therefore, each request is eventually acknowledged. Therefore, α is atomic initialized to v_0 as required. Since α was chosen arbitrarily, this protocol implements an atomic register. □

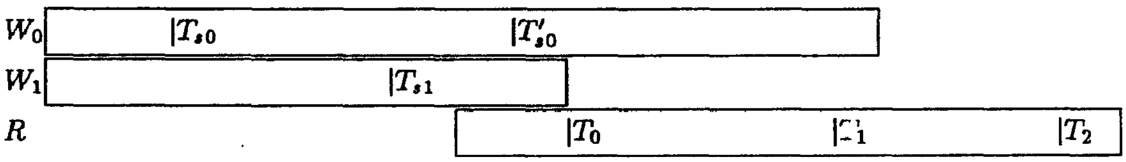

Figure 4: Timing for Read of Impotent Write (for contradiction)

8 Conclusions

There are several obvious ways to try to extend this algorithm to more than two writers; none of them work. A number of researchers have investigated this topic, and there are two or three proposed multi-writer atomic register protocols [VA], [PB]. It is difficult to design such protocols. As an example, we show how the natural extension of the two-writer protocol fails.

This protocol is reminiscent of Peterson's tournament protocol for mutual exclusion [PF]. Consider $N = 2^n$ writers arranged in a tournament in the same way. Divide the processors into pairs; each pair simulates a two-writer register from two real one-writer registers. Each pair of pairs then participates in the protocol, and so forth. However, this does not work. [This counterexample is due to Leslie Lamport, personal communication]

Suppose that we have four processors simulating a shared register in this scheme. With a gain of generality[6], we may ignore the simulation of the two-writer registers, and pretend that we are simulating a four-writer register on two real two-writer registers with the above protocol. Call the writers Wr_{00} and Wr_{01}, (who share real register 0) and Wr_{10} and Wr_{11} (and share real register 1). Processor Wr_{10} will not participate in this example. The sequence of events is given in Figure 5.

At the start, processor Wr_{00} will start trying to write the value 'x'. It will perform the real reads for its simulated write (denoted "(reads)" in the table), compute the tag bit that it will write, and go to sleep for a while. While it is asleep, Wr_{11} will write 'c'; then Wr_{01} will write 'd'. At this point, Wr_{11}'s value 'c' is obsolete. Then Wr_{00} will wake up and finish its write. Wr_{11}'s value will magically reappear in the register.

[6]This counterexample does not depend on any characteristics of my two-writer protocol; it works for any protocol, or even hardware atomic two-writer registers.

Processor	Action	Reg_0	Reg_1	Value
initial	—	'a',0	'b',0	'a'
Wr_{00}	real reads	'a',0	'b',0	'a'
Wr_{11}	sim. writes	'a',0	'c',1	'c'
Wr_{01}	sim. writes	'd',1	'c',1	'd'
Wr_{00}	real writes	'x',0	'c',1	'c'

When Wr_{01} writes, the value 'c' becomes obsolete. When Wr_{00} finishes its write, 'c' reappears.

Figure 5: Four-Writer Counterexample

Concurrent-access register algorithms may be of some use to other forms of concurrency control. For example, many database applications demand serializability (*i.e.*, atomicity) of requests. Most algorithms require some form of mutual exclusion. When many processors are sharing memory, the protocols for mutual exclusion can get relatively expensive. Register protocols are an example of a form of memory communication which provides atomicity without requiring synchronization or mutual exclusion.

An atomic register may be considered an object with abstract data type register[T], admitting the operations read and write(v), with all the operations atomic. It would be interesting to find protocols allowing more general data types, or perhaps even arbitrary abstract data types, to be shared atomically without waiting.

9 Acknowledgments

Leslie Lamport provided much helpful criticism and advice, as well as the example showing the failure of the extension of the protocol. I would like to thank Nancy Lynch, Mark Tuttle, Paul Vitányi, Jennifer Lundelius Welch, and Miller Maley, for their generous

help and perhaps excessive patience, and the first two for the use of their I/O Automaton model.

10 References

[BP] Burns, James E. and Gary L. Peterson, "Constructing multi-reader atomic values from non-atomic values.", Proceedings of PODC '87

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

[L1] L. Lamport, "A new solution to Dijkstra's concurrent programming problem," *CACM 17* **8** (August 1974), pp. 453-455.

[L2] L. Lamport, "On interprocess communication, Parts I and II," *Distributed Computing 1* **2** (1986), pp 77-85 and 86-101.

[LM] Nancy A. Lynch and Michael Merritt, "Introduction to the Theory of Nested Transactions", MIT-LCS Technical Report MIT/LCS/TR-367.

[LT] Nancy A. Lynch and Mark R. Tuttle, "Correctness Proofs for Distributed Algorithms", MIT-LCS Technical Report, to appear.

[P] G.L. Peterson, "Concurrent reading while writing," *ACM TOPLAS 5* **1** (January 1983), pp. 46-55.

[PB] G.L. Peterson and J.E. Burns, "Concurrent Reading While Writing II: The Multi-writer Case,", distributed manuscript.

[PF] G.L. Peterson and M.J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System", Proc 9^{th} STOC (May 1977), pp. 91-97.

[VA] P. Vitányi and B. Awerbuch, "Atomic shared register access by asynchronous hardware," in *Proceedings 27th IEEE Symp. on Foundations of Computer Science* (1986), pp. 233-243.