Wait-Free Synchronization

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
One Kendall Square
Cambridge, MA 02139

January 11, 1993

Abstract

A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The problem of constructing a wait-free implementation of one data object from another lies at the heart of much recent work in concurrent algorithms, concurrent data structures, and multiprocessor architectures. In the first part of this paper, we introduce a simple and general technique, based on reduction to a consensus protocol, for proving statements of the form "there is no wait-free implementation of X by Y." We derive a hierarchy of objects such that no object at one level has a wait-free implementation in terms of objects at lower levels. In particular, we show that atomic read/write registers, which have been the focus of much recent attention, are at the bottom of the hierarchy: they cannot be used to construct wait-free implementations of many simple and familiar data types. Moreover, classical synchronization primitives such as *test&set* and *fetch&add*, while more powerful than *read* and *write*, are also computationally weak, as are the standard message-passing primitives. Nevertheless, in the second part of the paper, we show that there do exist simple universal objects from which one can construct a wait-free implementation of any sequential object.

A preliminary version of this paper appeared in the Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1988 [12].

1 Introduction

A *concurrent object* is a data structure shared by concurrent processes. Algorithms for implementing concurrent objects lie at the heart of many important problems in concurrent systems. The traditional approach to implementing such objects centers around the use of *critical sections*: only one process at a time is allowed to operate on the object. Nevertheless, critical sections are poorly suited for asynchronous, fault-tolerant systems: if a faulty process is halted or delayed in a critical section, non-faulty processes will also be unable to progress. Even in a failure-free system, a process can encounter unexpected delay as a result of a page fault or cache miss, by exhausting its scheduling quantum, or if it is swapped out. Similar problems arise in heterogeneous architectures, where some processors may be inherently faster than others, and some memory locations may be slower to access.

A *wait-free* implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The wait-free condition provides fault-tolerance: no process can be prevented from completing an operation by undetected halting failures of other processes, or by arbitrary variations in their speed. The fundamental problem of wait-free synchronization can be phrased as follows:

> Given two concurrent objects X and Y, does there exist a wait-free implementation of X by Y?

It is clear how to show that a wait-free implementation exists: one displays it. Most of the current literature takes this approach. Examples include "atomic" registers from non-atomic "safe" registers [19], complex atomic registers from simpler atomic registers [4, 5, 16, 23, 25, 26, 29, 31], read-modify-write operations from combining networks [11, 15], and typed objects such as queues or sets from simpler objects [14, 18, 20].

It is less clear how to show that such an implementation does *not* exist. In the first part of this paper, we propose a simple new technique for proving statements of the form "there is no wait-free implementation of X by Y." We derive a hierarchy of objects such that no object at one level can implement any object at higher levels (see Figure 1). The basic idea is the following: each object has an associated *consensus number*, which is the maximum number of processes for which the object can solve a simple consensus problem. In a system of n or more concurrent processes, we show that it is impossible to construct a wait-free implementation of an object with consensus number n from an object with a lower consensus number.

These impossibility results do not by any means imply that wait-free synchronization is impossible or infeasible. In the second part of this paper, we show that there exist *universal* objects from which one can construct a wait-free implementation of any object. We give a simple test for universality, showing that an object is universal in a system of n processes if and only if it has a consensus number greater than or equal to n. In Figure 1, each object at level n is universal for a system of n processes. A machine architecture or programming language is computationally powerful enough to support arbitrary wait-free synchronization if and only if it provides a universal object as a primitive.

Most recent work on wait-free synchronization has focused on the construction of atomic read/write registers [4, 5, 16, 19, 23, 25, 26, 29, 31]. Our results address a basic question: what are these registers good for? Can they be used to construct wait-free implementations of more complex data structures? We show that atomic registers have few, if any, interesting applications in this area. From a set of atomic registers, we show that it is impossible to construct a wait-free implementation of (1) common data types such as sets, queues, stacks, priority queues, or lists, (2) most if not all the classical synchronization primitives, such as *test&set*, *compare&swap*, and *fetch&add*, and (3) such simple memory-to-memory operations as *move* or memory-to-memory *swap*. These results suggest that further progress in understanding wait-free synchronization requires turning our attention from the conventional *read* and *write* operations to more fundamental primitives.

Our results also illustrate inherent limitations of certain multiprocessor architectures. The NYU Ultracomputer project [10] has investigated architectural support for wait-free implementations of common synchronization primitives. They use combining networks to implement *fetch&add*, a generalization of *test&set*. IBM's RP3 [8] project is investigating a similar approach. The *fetch&add* operation is quite flexible: it can be used for semaphores, for highly concurrent queues, and even for database synchronization [11, 14, 30]. Nevertheless, we show that it is not universal, disproving a conjecture of Gottlieb et al. [11]. We also show that message-passing architectures such as hypercubes [28] are not universal either.

This paper is organized as follows. Section 2 defines a model of computation, Section 3 presents impossibility results, Section 4 describes some universal objects, and Section 5 concludes with a summary.

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
$\vdots$	$\vdots$
$2n-2$	n-register assignment
$\vdots$	$\vdots$
∞	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

Figure 1: Impossibility and Universality Hierarchy

2 The Model

Informally, our model of computation consists of a collection of sequential threads of control called *processes* that communicate through shared data structures called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the only means to manipulate that object. Each process applies a sequence of operations to objects, issuing an invocation and receiving the associated response. The basic correctness condition for concurrent systems is *linearizability* [14]: although operations of concurrent processes may overlap, each operation appears to take effect instantaneously at some point between its invocation and response. In particular, operations that do not overlap take effect in their "real-time" order.

2.1 I/O Automata

Formally, we model objects and processes using a simplified form of I/O automata [22]. Because the wait-free condition does not require any fairness or liveness conditions, and because we consider only finite sets of processes and objects, we do not make use of the full power of the I/O automata formalism. Nevertheless, simplified I/O automata provide a convenient way to describe the basic structure of of our model, and to give the basic definition of what it means for one object to implement another. For brevity, our later constructions and impossibility results are expressed less formally using pseudocode. It is a straightforward exercise to translate this notation into I/O automata.

An *I/O automaton* A is a non-deterministic automaton with the following components[1]:

- *States*(A) is a finite or infinite set of states, including a distinguished set of starting states.
- *In*(A) is a set of *input events*,
- *Out*(A) is a set of *output events*,
- *Int*(A) is a set of *internal events*,
- *Steps*(A) is a transition relation given by a set of triples (s', e, s), where s and s' are states and e is an event. Such a triple is called a *step*, and it means that an automaton in state s' can undergo a transition to state s, and that transition is associated with the event e.

If (s', e, s) is a step, we say that e is *enabled* in s'. I/O automata must satisfy the additional condition that inputs cannot be disabled: for each input event e and each state s', there exist a state s and a step (s', e, s).

An *execution fragment* of an automaton A is a finite sequence $s_0, e_1, s_1, \ldots e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and events such that each (s_i, e_{i+1}, s_{i+1}) is a step of A. An *execution* is an execution fragment where s_0 is a starting state. A *history fragment* of an automaton is the subsequence of events occurring in an execution fragment, and a *history* is the subsequence occurring in an execution.

A new I/O automaton can be constructed by *composing* a set of compatible I/O automata. (In this paper we consider only finite compositions.) A set of automata are *compatible* if they share no output or internal events. A state of the composed automaton S is a tuple of component states, and a starting state is a tuple of component starting states. The set of events of S, *Events*(S), is the union of the components' sets of events. The set of output events of S, *Out*(S), is the union of the components' sets of output events; the set of internal events, *Int*(S), is the union of the components' sets of internal events; and the set of input events of S, *In*(S), is *In*(S) − *Out*(S), all the input events of S that are not output events for some component. A triple (s', e, s) is in *Steps*(S) if and only if, for all component automata A, one of the following holds: (1) e is an event of A, and the projection of

[1] To remain consistent with the terminology of [14], we use "event" where Lynch and Tuttle use "operation," and "history" where they use "schedule."

the step onto A is a step of A, or (2) e is not an event of A, and A's state components are identical in s' and s. Note that composition is associative. If H is a history of a composite automaton and A a component automaton, $H|A$ denotes the subhistory of H consisting of events of A.

2.2 Concurrent Systems

A *concurrent system* is a set of processes and a set of objects. *Processes* represent sequential threads of control, and *objects* represent data structures shared by processes. A process P is an I/O automaton with output events $\text{INVOKE}(P, op, X)$, where *op* is an operation [2] of object X, and input events $\text{RESPOND}(P, res, X)$, where *res* is a result value. We refer to these events as *invocations* and *responses*. Two invocations and responses *match* if their process and object names agree. To capture the notion that a process represents a single thread of control, we say that a process history is *well-formed* if it begins with an invocation and alternates matching invocations and responses. An invocation is *pending* if it is not followed by a matching response. An *object* X has input events $\text{INVOKE}(P, op, X)$, where P is a process and *op* is an operation of the object, and output events $\text{RESPOND}(P, res, X)$, where *res* is a result value. Process and object names are unique, ensuring that process and object automata are compatible.

A *concurrent system* $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is an I/O automaton composed from processes $P_1, \ldots, P_n$ and objects $A_1, \ldots, A_m$, where processes and objects are composed by identifying corresponding INVOKE and RESPOND events. A history of a concurrent system is *well-formed* if each $H|P_i$ is well-formed, and a concurrent system is *well-formed* if each of its histories is well-formed. Henceforth, we restrict our attention to well-formed concurrent systems.

An execution is *sequential* if its first event is an invocation, and it alternates matching invocations and responses. A history is sequential if it is derived from a sequential execution. (Notice that a sequential execution permits process steps to be interleaved, but at the granularity of complete operations.) If we restrict our attention to sequential histories, then the behavior of an object can be specified in a particularly simple way: by giving pre- and postconditions for each operation. We refer to such a specification as a *sequential specification.* In this paper, we consider only objects whose sequential specifications are *total*: if the object has a pending invocation,

[2] *Op* may also include argument values.

then it has a matching enabled response. For example, a partial *deq* might be undefined when applied to an empty queue, while a total *deq* would return an exception. We restrict out attention to objects whose operations are total because it is unclear how to interpret the wait-free condition for partial operations. For example, the most natural way to define the effects of a partial *deq* in a concurrent system is to have it wait until the queue becomes non-empty, a specification that clearly does not admit a wait-free implementation.

Each history H induces a partial "real-time" order $\prec_H$ on its operations: $op_0 \prec_H op_1$ if the response for op_0 precedes the invocation for op_1. Operations unrelated by $\prec_H$ are said to be *concurrent.* If H is sequential, $\prec_H$ is a total order. Let $complete(H)$ denote the maximal subsequence of H consisting only of invocations and matching responses. A concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is *linearizable* if, for each history H, there exists a sequential history S such that:

- For all P_i, $complete(H')|P_i = S|P_i$
- $\prec_H \subset \prec_S$

In other words, the history "appears" sequential to each individual process, and this apparent sequential interleaving respects the real-time precedence ordering of operations. Equivalently, each operation appears to take effect instantaneously at some point between its invocation and its response. A concurrent object A is *linearizable* [14] if, for every history H of every concurrent system $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, $H|A_j$ is linearizable. A linearizable object is thus "equivalent" to a sequential object, and its operations can also be specified by simple pre- and postconditions. Henceforth, all objects are assumed to be linearizable. Unlike related correctness conditions such as sequential consistency [17] or strict serializability [24], linearizability is a *local* property: a concurrent system is linearizable if and only if each individual object is linearizable [14]. We restrict our attention to linearizable concurrent systems.

2.3 Implementations

An *implementation* of an object A is a concurrent system $\{F_1, \ldots, F_n; R\}$, where the F_i are called *front-ends*, and R is called the *representation* object. Informally, R is the data structure that implements A, and F_i is the procedure called by process P_i to execute an operation. An object implementation is shown schematically in Figure 2.

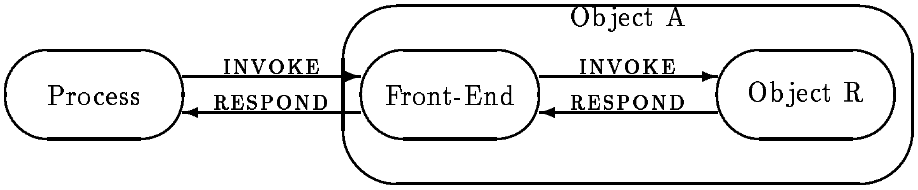

Figure 2: Schematic View of Object Implementation

- The external events of the implementation are just the external events of A: each input event $\text{INVOKE}(P_i, op, A)$ of A is an input event of F_i, and each output event $\text{RESPOND}(P_i, res, A)$ of A is an output event of F_i.

- The implementation has the following internal events: each input event $\text{INVOKE}(F_i, op, R)$ of R is composed with the matching output event of F_i, and each output event $\text{RESPOND}(F_i, res, R)$ of R is composed with the matching input event of F_i.

- To rule out certain trivial solutions, front-ends share no events; they communicate indirectly through R.

Let I_j be an implementation of A_j. I_j is *correct*, if for every history H of every system $\{P_1, \ldots, P_n; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a history H' of $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, such that $H|\{P_1, \ldots, P_n\} = H'|\{P_1, \ldots, P_n\}$.

An implementation is *wait-free* if:

- It has no history in which an invocation of P_i remains pending across an infinite number of steps of F_i.

- If P_i has a pending invocation in a state s, then there exists a history fragment starting from s, consisting entirely of events of F_i and R, that includes the response to that invocation.

The first condition rules out unbounded busy-waiting: a front-end cannot take an infinite number of steps without responding to an invocation. The second condition rules out conditional waiting: F_i cannot block waiting for another process to make a condition true. Note that we have not found it necessary to make fairness or liveness assumptions: a wait-free implementation guarantees only that that if R eventually responds to all invocations of F_i, then F_i will eventually respond to all invocations of P_i, independently of process speeds.

An implementation is *bounded wait-free* if there exists N such that there is no history in which an invocation of P_i remains pending across N steps of F_i. Bounded wait-free implies wait-free, but not vice-versa. We use the wait-free condition for impossibility results, and the bounded wait-free condition for universal constructions.

For brevity, we say that R *implements* A if there exists a wait-free implementation $\{F_1, \ldots, F_n; R\}$ of A. It is immediate from the definitions that *implements* is a reflexive partial order on the universe of objects. In the rest of the paper, we investigate the mathematical structure of the *implements* relation. In the next section, we introduce a simple technique for proving that one object does *not* implement another, and in the following section we display some "universal" objects capable of implementing any other object.

3 Impossibility Results

Informally, a *consensus protocol* is a system of n processes that communicate through a set of shared objects $\{X_1, \ldots, X_m\}$. The processes each start with an input value from some domain $\mathcal{D}$, they communicate with one another by applying operations to the shared objects, they eventually agree on a common input value and halt. A consensus protocol is required to be:

- *Consistent*: distinct processes never decide on distinct values.
- *Wait-free*: each process decides after a finite number of steps.
- *Valid*: the common decision value is the input to some process.

For our purposes, it is convenient to express the consensus problem using the terminology of abstract data types. A *consensus object* provides a single operation:

decide(input: value) returns(value)

A protocol's sequential specification is simple: all *decide* operations return the argument value of the first *decide* (c.f., Plotkin's "sticky-bit" [27]). This common value is called the history's *decision value.* A wait-free linearizable implementation of a consensus object is called a *consensus protocol* (c.f., Fisher, Lynch, and Paterson [9]).

We will investigate the circumstances under which it is possible to construct consensus protocols from particular objects. Most of the constructions presented in this paper use multi-reader/multi-writer registers in addition to the object of interest. For brevity we say "X solves n-process consensus" if there exists a consensus protocol $\{F_1, \ldots, F_n; W, X\}$, where W is a set of read/write registers, and W and X may be initialized to any state.

Definition 1 *The* consensus number *for X is the largest n for which X solves n-process consensus. If no largest n exists, the consensus number is said to be infinite.*

It is an immediate consequence of our definitions that if Y implements X, and X solves n-process consensus, then Y also solves n-process consensus.

Theorem 2 *If X has consensus number n, and Y has consensus number $m < n$, then there exists no wait-free implementation of X by Y in a system of more than m processes.*

Proof: As noted above, all front-end and object automata are compatible by definition, and thus their composition is well-defined. Let $\{F_1, \ldots, F_k; W, X\}$ be a consensus protocol, where $k > m$ and W is a set of read/write registers. Let $\{F'_1, \ldots, F'_k; Y\}$ be an implementation of X. It is easily checked that $\{F_1, \ldots, F_n; W, \{F'_1, \ldots, F'_n; Y\}\}$ is wait-free, and because composition is associative, it is identical to $\{F_1 \cdot F'_1, \ldots, F_n \cdot F'_n; W, Y\}$, where $F_1 \cdot F'_1$ is the composition of F_i and F'_i. Since the former is a consensus protocol, so is the latter, contradicting the hypothesis that Y has consensus number m. ∎

In the rest of this section, we consider a number of objects, displaying consensus protocols for some, and impossibility results for others. For impossibility proofs, we will usually assume the existence of a consensus protocol, and then derive a contradiction by constructing a sequential execution that forces the protocol to run forever. When constructing a consensus protocol for a particular linearizable object, we observe that the linearizability condition implies that if there exists an execution in which consensus fails, either because it is inconsistent, invalid, or it runs forever, then there exists an

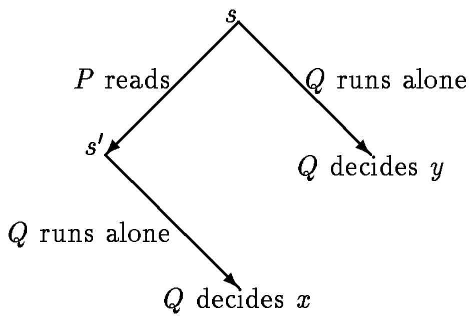

Figure 3: P reads first.

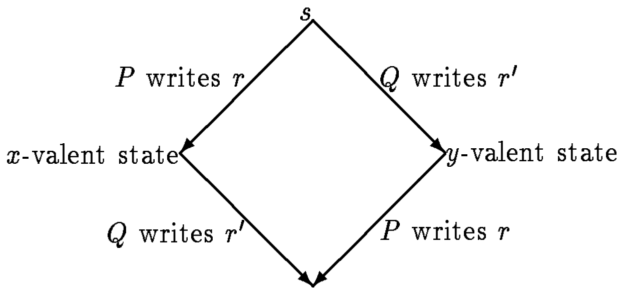

Figure 4: P and Q write different registers.

equivalent sequential execution with the same property. As a consequence, a consensus protocol is correct if and only if all its sequential executions are correct. For brevity, protocols are defined informally by pseudo-code; their translations into I/O automata should be self-evident.

3.1 Atomic Read/Write Registers

In this section, we show there exists no two-process consensus protocol using multi-reader/multi-writer atomic registers. First, some terminology. A protocol state is *bivalent* if either decision value is still possible: i.e., the current execution can be extended to yield different decision values. Otherwise it is *univalent*. An *x-valent* state is a univalent state with eventual decision value x. A *decision step* is an operation that carries a protocol from a bivalent to a univalent state.

Theorem 3 *Read/write registers have consensus number 1.*

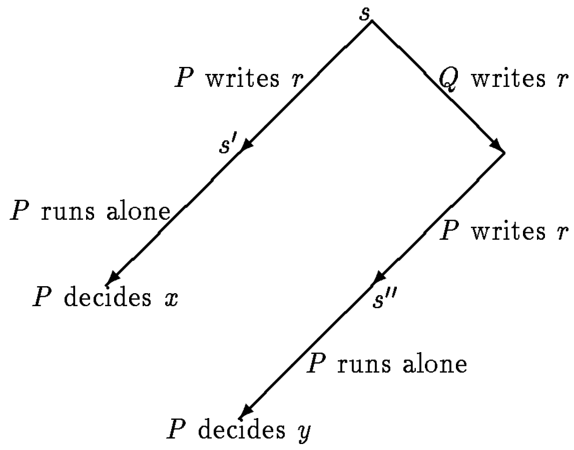

Figure 5: P and Q write the same register.

Proof: Assume there exists a two-process consensus protocol implemented from atomic read/write registers. We derive a contradiction by constructing an infinite sequential execution that keeps any such protocol in a bivalent state. If the processes have different input values, the validity condition implies that the initial state is bivalent. Consider the following sequential execution, starting from the initial state. In the first stage, P executes a sequence of operations (i.e., alternates matching invocation and response events) until it reaches a state where the next operation will leave the protocol in a univalent state. P must eventually reach such a state, since it cannot run forever, and it cannot block. In the second stage, Q executes a sequence of operations until it reaches a similar state, and in successive stages, P and Q alternate sequences of operations until each is about to make a decision step. Because the protocol cannot run forever, it must eventually reach a bivalent state s in which any subsequent operation of either process is a decision step. Suppose P's operation carries the protocol to an x-valent state, and Q's operation carries the protocol to a y-valent state, where x and y are distinct.

- Suppose the decision step for one process, say P, is to read a shared register (Figure 3). Let s' be the protocol state immediately following the read. The protocol has a history fragment starting from s, con-

```
RMW(r: register, f: function) returns(value)
    previous := r
    r := f(r)
    return previous
    end RMW
```

Figure 6: Read-Modify-Write

sisting entirely of operations of Q, yielding decision value y. Since the states s and s' differ only in the internal state of P, the protocol has the same history fragment starting in s', an impossibility because s' is x-valent.

- Suppose the processes write to different registers (Figure 4). The state that results if P's write is immediately followed by Q's is identical to the state that results if the writes occur in the opposite order, which is impossible, since one state is x-valent and the other is y-valent.

- Suppose the processes write to the same register (Figure 5). Let s' be the x-valent state immediately after P's write. There exists a history fragment starting from s' consisting entirely of operations of P that yields the decision value x. Let s'' be the y-valent state reached if Q's write is immediately followed by P's. Because P overwrites the value written by Q, s' and s'' differ only in the internal states of Q, and therefore the protocol has the same history fragment starting from s'', an impossibility since s'' is y-valent.

∎

Similar results have been shown by Loui and Abu-Amara [21], Chor, Israeli, and Li [6], and Anderson and Gouda [1]. Our contribution lies in the following corollary:

Corollary 4 *It is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic read/write registers.*

3.2 Read-Modify-Write Operations

```
decide(input: value) returns(value)
  prefer[P] := input
  if RMW(r,f) = v
    then return prefer[P]
    else  return prefer[Q]
    end if
  end decide
```

Figure 7: Read-Modify-Write: Two-Process Consensus

Kruskal, Rudolph, and Snir [15] have observed that many, if not all, of the classical synchronization primitives can be expressed as *read-modify-write* operations, defined as follows. Let r be a register, and f a function from values to values. The operation $RMW(r, f)$ is informally defined by the procedure shown in Figure 6, which is executed atomically. If f is the identity, $RMW(r, f)$ is simply a *read* operation. A read-modify-write operation is *non-trivial* if f is not the identity function. Examples of well-known non-trivial read-modify-write operations include *test&set*, *swap*, *compare&swap*, and *fetch&add*. Numerous others are given in [15].

Theorem 5 *A register with any non-trivial read-modify-write operation has a consensus number at least 2.*

Proof: Since f is not the identity, there exists a value v such that $v \neq f(v)$. Let P and Q be two processes that share a two-register array *prefer*, where each entry is initialized to $\perp$, and a read-modify-write register r, initialized to v. P executes the protocol shown in Figure 7 (Q's protocol is symmetric.)

Expressed in terms of the I/O automaton model, the read-modify-write register r is the object X, the *prefer* array is the set of atomic registers W, and the pseudo-code in Figure 7 defines the front-end automaton for P. The front-end has three output events: the *write* and *RMW* invocations sent to r and *prefer*, and the decision value returned to P. Similarly, its input events are P's invocation of *decide*, and the responses to the *write* and *RMW* invocations.

As noted above, because r and *prefer* are linearizable, it suffices to check correctness for sequential executions. The only operations that do not commute are the two read-modify-write operations applied to r. The protocol chooses P's input if P's operation occurs first, and Q's input otherwise. ∎

Corollary 6 *It is impossible to construct a wait-free implementation of any non-trivial read-modify-write operation from a set of atomic read/write registers in a system with two or more processes.*

Although read-modify-write registers are more powerful than read/write registers, many common read-modify-write operations are still computationally weak. In particular, one cannot construct a wait-free solution to three process consensus using registers that support any combination of *read*, *write*, *test&set*, *swap*, and *fetch&add* operations. Let F be a set of functions indexed by an arbitrary set S. Define F to be *interfering* if for all values v and all i and j in S, either (1) f_i and f_j commute: $f_i(f_j(v)) = f_j(f_i(v))$, or (2) one function "overwrites" the other: either $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$.

Theorem 7 *There is no wait-free solution to three-process consensus using any combination of read-modify-write operations that apply functions from an interfering set F.*

Proof: By contradiction. Let the three processes be P, Q, and R. As in the proof of Theorem 2, we construct a sequential execution leaving the protocol in bivalent state where every operation enabled for P and Q is a decision step, some operation of P carries the protocol to an x-valent state, and some operation of Q carries the protocol to a y-valent state, where x and y are distinct. By the usual commutativity argument, P and Q must operate on the same register; say, P executes $RMW(r, f_i)$ and Q executes $RMW(r, f_j)$.

Let v be the current value of register r. There are two cases to consider. First, suppose $f_i(f_j(v)) = f_j(f_i(v))$. The state s that results if P executes $RMW(r, f_i)$ and Q executes $RMW(r, f_j)$ is x-valent, thus there exists some history fragment consisting entirely of operations of R that yields decision value x. Let s' be the state that results if P and Q execute their operations in the reverse order. Since the register values are identical in s and s', the protocol has the same history fragment starting in s', contradicting the hypothesis that s' is y-valent.

Second, suppose $f_i(f_j(v)) = f_j(v)$. The state s that results if P executes $RMW(r, f_i)$ and Q executes $RMW(r, f_j)$ is x-valent, thus there exists some history fragment consisting entirely of operations of R that yields decision value x. Let s' be the state that results if Q alone executes its operation. Since the register values are identical in s and s', the protocol has the same

```
compare&swap(r: register, old: value, new: value)
   returns(value)
   previous := r
   if previous = old
      then r := new
      end if
   return previous
   end compare&swap
```

Figure 8: Compare&Swap

history fragment starting in s', contradicting the hypothesis that s' is y-valent. ∎

It follows that one cannot use any combination of these classical primitives to construct a wait-free implementation of any object with consensus number greater than 2.

Another classical primitive is *compare&swap*, shown in Figure 8. This primitive takes two values: *old* and *new*. If the register's current value is equal to *old*, it is replaced by *new*, otherwise is left unchanged. The register's old value is returned.

Theorem 8 *A* compare&swap *register has infinite consensus number.*

Proof: In the protocol shown in Figure 9, the processes share a register r initialized to $\perp$. Each process attempts to replace $\perp$ with its input; the decision value is established by the process that succeeds.

This protocol is clearly wait-free, since it contains no loops. Consistency follows from the following observations: (1) $r \neq \perp$ is a postcondition of *compare&swap*, and (2) for any $v \neq \perp$, the assertion $r = v$ is *stable* — once it becomes true, it remains true. Validity follows from the observation that if $r \neq \perp$, then r contains some process's input. ∎

Corollary 9 *It is impossible to construct a wait-free implementation of a* compare&swap *register from a set of registers that support any combination of* read, write, test&set, swap, *or* fetch&add *operations in a system of three or more processes.*

```
decide(input: value) returns(value)
   first := compare&swap(r, ⊥, input)
   if first = bottom
      then return input
      else  return first
      end if
   end decide
```

Figure 9: Compare&Swap: n-Process Consensus

```
decide(input: value) returns(value)
   prefer[P] := input
   if deq(q) = 0
      then return prefer[P]
      else  return prefer[Q]
      end if
   end decide
```

Figure 10: FIFO Queues: Two-Process Consensus

3.3 Queues, Stacks, Lists, Etc.

Consider a FIFO queue with two operations: *enq* places an item at the end of the queue, and *deq* removes the item from the head of the queue, returning an error value if the queue is empty.

Theorem 10 *The FIFO queue has consensus number at least 2.*

Proof: Figure 10 shows a two-process consensus protocol. The queue is initialized by enqueuing the value 0 followed by the value 1. As above, the processes share a two-element array *prefer*. P executes the protocol shown in Figure 10 (Q's protocol is symmetric). Each process dequeues an item from the queue, returning its own preference if it dequeues 0, and the other's preference if it dequeues 1.

The protocol is wait-free, since it contains no loops. If each process returns its own input, then they must both have dequeued 0, violating the queue specification. If each returns the others' input, then they must both have dequeued 1, also violating the queue specification. Let the "winner" be the process that dequeues 0. Validity follows by observing that the winner's position in *prefer* is initialized before the first queue operation. ∎

Trivial variations of this program yield protocols for stacks, priority queues, lists, sets, or any object with operations that return different results if applied in different orders.

Corollary 11 *It is impossible to construct a wait-free implementation of a queue, stack, priority queue, set, or list from a set of atomic read/write registers.*

Although FIFO queues solve two-process consensus, they cannot solve three-process consensus.

Theorem 12 *FIFO queues have consensus number 2.*

Proof: By contradiction. Assume we have a consensus protocol for processes P, Q, and R. As before, we maneuver the protocol to a state where P and Q are each about to make a decision step. Assume that P's operation would carry the protocol to an x-valent state and Q's to a y-valent state. The rest is a case analysis.

First, suppose P and Q both execute *deq* operations. Let s be the protocol state if P dequeues and then Q dequeues, and let s' be the state if the dequeues occur in the opposite order. Since s is x-valent, there exists a history fragment from s, consisting entirely of operations of R, yielding decision value x. But s and s' differ only in the internal states of P and Q, thus the protocol has the same history fragment from s', a contradiction because s' is y-valent.

Second, suppose P does an *enq* and Q a *deq*. If the queue is non-empty, the contradiction is immediate because the two operations commute: R cannot observe the order in which they occurred. If the queue is empty, then the y-valent state reached if Q dequeues and then P enqueues is indistinguishable to R from the x-valent state reached if P alone enqueues.

Finally, suppose both P and Q do *enq* operations. Let s be the state at the end of the following execution:

1. P and Q enqueue items p and q in that order.

2. Run P until it dequeues p. (Since the only way to observe the queue's state is via the *deq* operation, P cannot decide before it observes one of p or q.)

3. Run Q until it dequeues q.

Let s' be the state after the following alternative execution:

```
decide(input: value) returns(value)
  enq(q, input)
  return peek(q)
  end decide
```

Figure 11: Augmented FIFO Queue: n-Process Consensus

1. Q and P enqueue items q and p in that order.
2. Run P until it dequeues q.
3. Run Q until it dequeues p.

Clearly, s is x-valent and s' is y-valent. Both of P's executions are identical until it dequeues p or q. Since P is halted before it can modify any other objects, Q's executions are also identical until it dequeues p or q. By a now-familiar argument, a contradiction arises because s and s' are indistinguishable to R. ∎

Trivial variations of this argument can be applied to show that many similar data types, such as sets, stacks, double-ended queues, and priority queues, all have consensus number 2.

A *message-passing architecture* (e.g., a hypercube, [28]) is a set of processors that communicate via shared FIFO queues. Theorem 12 implies that message-passing architectures cannot solve three-process consensus or implement any object that can. Dolev, Dwork, and Stockmeyer [7] give a related result: point-to-point FIFO message channels cannot solve two-process consensus. That result does not imply Theorem 12, however, because a queue item, unlike a message, is not "addressed" to any particular process, and hence it can be dequeued by anyone.

3.4 An Augmented Queue

Let us augment the queue with one more operation: *peek* returns but does not remove the first item in the queue.

Theorem 13 *The augmented queue has infinite consensus number.*

Proof: In the protocol shown in Figure 11, the queue q is initialized to *empty*, and each process enqueues its own input. The decision value is the input of the process whose *enq* occurs first.

```
decide(input: value) returns(value)
   prefer[P] := input
   r[P,2] ← r[P,1]
   for i in P+1 .. n do
      r[i, 1] := 0
      end for
   for i in n .. 1 do
      if r[i,2] = 1
        then return prefer[i]
        end if
      end for
   end decide
```

Figure 12: Memory-To-Memory Move: n-Process Consensus

As usual, the protocol is wait-free, since it contains no loops. Consistency follows from the following observations: (1) "the queue is non-empty" is a postcondition of each *enq*, and hence a precondition for each *peek*, and (2) for any v, "v is the first item in the queue" is stable. Validity follows from the observation that the first item in the queue is some process's input. ∎

Corollary 14 *It is impossible to construct a wait-free implementation of the augmented queue from a set of registers supporting any combination of* read, write, test&set, swap, *or* fetch&add *operations.*

Corollary 15 *It is impossible to construct a wait-free implementation of the augmented queue from a set of regular queues.*

The *fetch&cons* operation atomically threads an item onto the front of a linked list. By an argument virtually identical to the one given for Theorem 13, a linked list with *fetch&cons* has infinite consensus number.

3.5 Memory-To-Memory Operations

Consider a collection of atomic read/write registers having one additional operation: *move* atomically copies the value of one register to another [3]. We use the expression "$a \leftarrow b$" to move the contents of b to a.

[3] Memory-to-memory *move* should not be confused with assignment; the former copies values between two public registers, while the latter copies values between public and private registers.

Theorem 16 *An array of registers with* move *has infinite consensus number.*

Proof: An n-process consensus protocol appears in Figure 12. The processes share two arrays: *prefer*$[1..n]$ and $r[1..n, 1..2]$, where $r[P, 1]$ is initialized to 1 and $r[P, 2]$ to 0, for $1 \leq P \leq n$. The protocol is clearly wait-free, since all loops are bounded.

To show consistency, we use the following assertions:

$$\begin{aligned} \mathcal{P}(P) &\equiv r[P,1] = 0 \wedge r[P,2] = 0 \\ \mathcal{Q}(P) &\equiv r[P,2] = 1 \\ \mathcal{S}(P) &\equiv \mathcal{P}(P) \vee \mathcal{Q}(P) \end{aligned}$$

It is easily checked that $\mathcal{P}(P)$, $\mathcal{Q}(P)$, and $\mathcal{S}(P)$ are stable for each P, that $\mathcal{P}(P)$ and $\mathcal{Q}(P)$ are mutually exclusive, that $\mathcal{S}(P)$ is true after P executes Statement #2, and that $\mathcal{S}(i)$ is true after each execution of Statement #4. We say that a process P has *stabilized* if $\mathcal{S}(P)$ holds.

We claim that if $\mathcal{P}(P)$ holds for some P, then $\mathcal{Q}(Q)$ holds for some $Q < P$, and that every process between Q and P has stabilized. Let P be the least process for which $\mathcal{P}(P)$ holds. Since $r[P, 1]$ and $r[P, 2]$ are both 0, some $Q < P$ must have assigned 0 to $r[P, 1]$ (Statement #4) before P executed Statement #2. Q, however, executes Statement #2 before Statement #4, hence $\mathcal{S}(Q)$ holds. Since $\mathcal{P}(Q)$ is false by hypothesis, $\mathcal{Q}(Q)$ must hold. Moreover, if Q has assigned to $r[P, 1]$, then it has assigned to every $r[P', 1]$ for $Q < P' < P$, thus each such P' has stabilized.

Define the *termination assertion* as follows:

$$\mathcal{T}(P) \equiv \mathcal{Q}(P) \wedge (\forall Q > P)\, \mathcal{P}(Q).$$

$\mathcal{T}$ is stable, and it holds for at most one process. When P finishes the first loop (Statements #3-5), every process greater than or equal to P has stabilized. If any of them satisfies $\mathcal{T}$, we are done. Otherwise, there exists a largest $Q < P$ satisfying $\mathcal{Q}(Q)$, and all the processes between P and Q have stabilized, implying that $\mathcal{T}(Q)$ holds. When P's protocol terminates, it chooses the input of the unique Q satisfying $\mathcal{T}(Q)$. Since the termination assertion is stable, all processes agree.

Validity follows because *prefer*$[P]$ must be initialized before $\mathcal{T}(P)$ can become true. ∎

```
decide(input: value) returns(value)
   prefer[P] := input
   swap(a[P],r)
   for Q in 1 .. n do
      if a[Q] = 1
        then return prefer[Q]
        end if
      end for
   end decide
```

Figure 13: Memory-To-Memory Swap: n-Process Consensus

Theorem 17 *An array of registers with memory-to-memory* swap [4] *has infinite consensus number.*

Proof: The protocol is shown in Figure 13. The processes share an array of registers $a[1..n]$ whose elements are initialized to 0, and a single register r, initialized to 1. The first process to swap 1 into a wins. The protocol is wait-free because the loop is bounded. To show consistency, consider the following assertions, where "$\exists! P$" means "there exists a unique P."

$$r = 1 \vee (\exists! P)\, a[P] = 1$$
$$r = 0$$

The first assertion is invariant, and the second is stable and becomes true after the first *swap*. It follows that each process observes a unique, stable P such that $a[P] = 1$.

Validity follows because each process initializes its position in *prefer* before executing a *swap*. ∎

Corollary 18 *It is impossible to construct a wait-free implementation of memory-to-memory* move *or* swap *from a set of registers that support any combination of* read, write, test&set, swap, *or* fetch&add *operations.*

Corollary 19 *It is impossible to construct a wait-free implementation of memory-to-memory* move *or* swap *from a set of FIFO queues.*

[4]The memory-to-memory *swap* should not be confused with the read-modify-write *swap*; the former exchanges the values of two public registers, while the latter exchanges the value of a public register with a processor's private register.

3.6 Multiple Assignment

The expression:

$$r_1, \ldots, r_m := v_1, \ldots, v_m$$

atomically assigns each value v_i to each register r_i.

Theorem 20 *Atomic m-register assignment has consensus number at least m.*

Proof: The protocol uses m "single-writer" registers $r_1, \ldots, r_m$, where P_i writes to register r_i, and $m(m-1)/2$ "multi-writer" registers r_{ij}, where $i > j$, where P_i and P_j both write to register r_{ij}. All registers are initialized to $\perp$. Each process atomically assigns its input value to m registers: its single-writer register and its $m-1$ multi-writer registers. The decision value of the protocol is the first value to be assigned.

After assigning to its registers, a process determines the relative ordering of the assignments for two processes P_i and P_j as follows.

- Read r_{ij}. If the value is $\perp$, then neither assignment has occurred.
- Otherwise, read r_i and r_j. If r_i's value is is $\perp$, then P_j precedes P_i, and similarly for r_j.
- If neither r_i nor r_j is $\perp$, reread r_{ij}. If its value is equal to the value read from r_i, then P_j precedes P_i, else vice-versa.

By repeating this procedure, a process can determine the value written by the earliest assignment. ∎

This result can be improved.

Theorem 21 *Atomic m-register assignment has consensus number at least* $2m-2$.

Proof: Consider the following two-phase protocol. Each process has two single-writer registers, one for each phase, and each pair of processes share a register. Divide the processes into two predefined groups of $m-1$. In the first phase, each group achieves consensus within itself using the protocol from Theorem 20. In the second phase, each process atomically assigns its group's value to its phase-two single-writer register and the $m-1$ multi-writer registers shared with processes in the other group. Using the ordering procedure described above, the process constructs a directed graph G with the property that there is an edge from P_j to P_k if P_j and P_k are in different groups and the former's assignment precedes the latter's. It then locates a *source* process having at least one outgoing edge but no incoming edges, and returns that process's value. At least one process has performed an assignment, thus G has edges. Let Q be the process whose assignment is first in the linearization order. Q is a source, and it has an outgoing edge to every process in the other group, thus no process in the other group is also a source. Therefore, all source processes belong to the same group. ∎

This algorithm is optimal with respect to the number of processes.

Theorem 22 *Atomic m-register assignment has consensus number exactly $2m-2$.*

Proof: We show that atomic m-register assignment cannot solve $2m-1$-process consensus for $m>1$. By the usual construction, we can maneuver the protocol into a bivalent state s in which any subsequent operation executed by any process is a decision step. We refer to the decision value forced by each process as its *default.*

We first show that each process must have a "single-writer" register that it alone writes to. Suppose not. Let P and Q be processes with distinct defaults x and y. Let s' be the state reached from s if P performs its assignment, Q performs its assignment, and the other processes perform theirs. Because P went first, s' is x-valent. By hypothesis, every register written by P has been overwritten by another process. Let s'' be the state reached from s if P halts without writing, but all other processes execute in the same order. Because Q wrote first, s'' is y-valent. There exists a history fragment from s', consisting entirely of operations of Q, with decision value x. Because the values of the registers are identical in s' and s'', the protocol has the same history fragment from s'', a contradiction because s'' is y-valent.

We next show that if P and Q have distinct default values, then there must be some register written only by those two processes. Suppose not. Let s' be the state reached from s if P performs its assignment, Q performs its assignment, followed by all other processes' assignments. Let s'' be the state reached by the same sequence of operations, except that P and Q execute their assignments in the reverse order. Because s' is x-valent, there exists a history fragment from s' consisting of operations of P that with decision value x. But because every register written by both P and Q has been overwritten by some other process, the register values are the same in both s and s', hence the protocol has the same history fragment from s'', a contradiction.

It follows that if P has default value x, and there are k processes with different default values, then P must assign to $k+1$ registers. If there are $2m-1$ processes which do not all have the same default, then some process must disagree with at least m other processes, and that process must must assign to $m+1$ registers. ∎

The last theorem shows that consensus is irreducible in the following sense: it is impossible to achieve consensus among $2n$ processes by combining protocols that achieve consensus among at most $2m < 2n$ processes. If it were possible, one could implement each individual $2m$-process protocol using $m-1$-register assignment, yielding a $2n$-process consensus protocol, contradicting Theorem 22.

3.7 Remarks

Fischer, Lynch, and Paterson [9] have shown that there exists no two-process consensus protocol using message channels that permit messages to be delayed and reordered. That result does not imply Theorem 3, however, because atomic read/write registers lack certain commutativity properties of asynchronous message buffers. (In particular, Lemma 1 of [9] does not hold.)

Dolev, Dwork, and Stockmeyer [7] give a thorough analysis of the circumstances under which consensus can be achieved by message-passing. They consider the effects of thirty-two combinations of parameters: synchronous vs. asynchronous processors, synchronous vs. asynchronous communication, FIFO vs. non-FIFO message delivery, broadcast vs. point-to-point transmission, and whether *send* and *receive* are distinct primitives. Expressed in their terminology, our model has asynchronous processes, synchronous communication, and distinct *send* and *receive* primitives. We model *send* and *receive* as operations on a shared message channel object; whether delivery is FIFO and whether broadcast is supported depends on the type of the channel. Some of their results translate directly into our model: it is impossible to achieve two-process consensus by communicating through a shared channel that supports either broadcast with unordered delivery, or point-to-point transmission with FIFO delivery. Broadcast with ordered delivery, however, does solve n-process consensus.

A *safe* read/write register [19] is one that behaves like an atomic read/write register as long as operations do not overlap. If a *read* overlaps a *write*, however, no guarantees are made about the value read. Since atomic registers implement safe registers, safe registers cannot solve two-process consensus, and hence the impossibility results we derive for atomic registers apply equally to safe registers. Similar remarks apply to atomic registers that restrict the number of readers or writers.

Loui and Abu-Amara [21] give a number of constructions and impossibility results for consensus protocols using shared read-modify-write registers, which they call "test&set" registers. Among other results, they show that n-process consensus for $n > 2$ cannot be solved by read-modify-write operations on single-bit registers.

Lamport [18] gives a queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. With minor changes, this implementation can be transformed into a wait-free implementation using atomic read/write registers. Theorem 3 implies that Lamport's queue cannot be extended to permit concurrent *deq* operations without augmenting the *read* and *write* operations with more powerful primitives.

A concurrent object implementation is *non-blocking* if it guarantees that some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. The non-blocking condition guarantees that the system as a whole will make progress despite individual halting failures or delays. A wait-free implementation is necessarily non-blocking, but not vice-versa, since a non-blocking implementation may permit individual processes to starve. The impossibility and universality results presented in this paper hold for non-blocking implementations as well as wait-free implementations.

Elsewhere [14], we give a non-blocking implementation of a FIFO queue, using *read*, *fetch&add*, and *swap* operations, that permits an arbitrary number of concurrent *enq* and *deq* operations. Corollary 14 implies that this queue implementation cannot be extended to support a non-blocking *peek* operation without introducing more powerful primitives.

4 Universality Results

An object is *universal* if it implements any other object. In this section, we show that any object with consensus number n is universal in a system of n (or fewer) processes. The basic idea is the following: we represent the object as a linked list, where the sequence of cells represents the sequence of operations applied to the object (and hence the object's sequence of states).

A process executes an operation by threading a new cell on to the end of the list. When the cell becomes sufficiently old, it is reclaimed and reused. Our construction requires $O(n^3)$ memory cells to represent the object, and $O(n^3)$ worst-case time to execute each operation. We assume cells can hold integers of unbounded size. Our presentation is intended to emphasize simplicity, and omits many obvious optimizations.

Let INVOC be the object's domain of invocations, RESULT its domain of results, and STATE its domain of states. An object's behavior may be specified by the following relation:

$$apply \subset \text{INVOC} \times \text{STATE} \times \text{STATE} \times \text{RESULT}.$$

This specification means that applying operation p in state s leaves the object in a state s' and returns result value r, where $\langle p, s, s', r\rangle \in apply$. *Apply* is a relation (rather than a function) because the operation may be non-deterministic. For brevity, we use the notation $apply(p, s)$ to denote an arbitrary pair $\langle s', r\rangle$ such that $\langle p, s, s', r\rangle \in apply$.

4.1 The Algorithm

An object is represented by a doubly-linked list of *cells* having the following fields:

- *Seq* is the cell's sequence number in the list. This field is zero if the cell is initialized but not yet threaded onto the list, and otherwise it is positive. Sequence numbers for successive cells in the list increase by one.
- *Inv* is the invocation (operation name and argument values).
- *New* is a consensus object whose value is the pair $\langle new.state, new.result\rangle$. The first component is the object's state following the operation, and the second is the operation's result value, if any.
- *Before* is a pointer to the previous cell in the list. This field is used only for free storage management.
- *After* is consensus object whose value is a pointer to the next cell in the list.

If c and d are cells, the function $\max(c, d)$ returns the cell with the higher sequence number.

Initially, the object is represented by a unique *anchor* cell with sequence number 1, holding a creation operation and an initial state.

The processes share the following data structures.

- *Announce* is an n-element array whose P^{th} element is a pointer to the cell P is currently trying to thread onto the list. Initially all elements point to the anchor cell.

- *Head* is an n-element array whose P^{th} element is a pointer to the last cell in the list that P has observed. Initially all elements point to the anchor cell.

Let $\max(head)$ be $\max(head[1].seq, \ldots, head[n].seq)$, and let "$c \in head$" denote the assertion that a pointer to cell c has been assigned to $head[Q]$, for some Q.

We use the following auxiliary variables:

- *concur*(P) is the set of cells whose addresses have been stored in the *head* array since P's last announcement.

- *start*(P) is the the value of $\max(head)$ at P's last announcement.

Notice that:

$$|concur(P)| + start(P) = \max(head) \tag{1}$$

Auxiliary variables do not affect the protocol's control flow; they are present only to facilitate proofs.

The protocol for process P is shown in Figure 14. In this figure, "v: T := e" declares and initializes variable v of type T to a value e, and the type "*cell" means "pointer to cell." Sequences of statements enclosed in angle brackets are executed atomically. In each of these compound statements, only the first affects shared data or control flow; the remainder are "bookkeeping operations" that update auxiliary variables. For readability, auxiliary variables are shown in italics.

Informally, the protocol works as follows. P allocates and initializes a cell to represent the operation (Statement #1). It stores a pointer to the cell in *announce*$[P]$ (Statement #2), ensuring that if P itself does not succeed in threading its cell onto the list, some other process will. To locate a cell near the end of the list, P scans the *head* array, setting *head*$[P]$ to the cell with the maximal sequence number (Statement #3). P then enters the main loop of the protocol (Statement #4), which it executes until its own cell has been threaded onto the list (detected when its sequence number becomes non-zero). P chooses a process to "help" (Statement #6), and checks whether that process has an unthreaded cell (Statement #7). If so, then P will try to thread it, otherwise it tries to thread its own cell. (If this helping step were omitted, the protocol would be non-blocking rather than wait-free.) P tries to set $head[P].after$ to point to the cell it is trying to thread (Statement #8). The *after* field must be a consensus cell to ensure that only one process succeeds in setting it. Whether or not P succeeds, it then initializes the remaining fields of the next cell in the list. Because the operation may be non-deterministic, different processes may try to set the *new* field to different values, so this field must be a consensus object (Statement #9). The values of the other fields are computed deterministically, so they can simply be written as atomic registers (Statements #10 and #11). For brevity, we say that a process *threads* a cell in Statement #7 if the *decide* operation alters the value of the *after* field, and it *announces* a cell at Statement #2 when it stores the cell's address in *announce*.

Lemma 23 *The following assertion is invariant:*

$$|concur(P)| > n \Rightarrow announce(P) \in head$$

Proof: If $|concur(P)| > n$, then $concur(P)$ includes successive cells q and r with respective sequence numbers equal to $P - 1 \bmod n$ and $P \bmod n$, threaded by processes Q and R. Because q is in $concur(P)$, Q threads q after P's announcement. Because R cannot modify an unthreaded cell, R reads $announce[P]$ (Statement #5) after Q threads q. It follows that R reads $announce[P]$ after P's announcement, and therefore either $announce[P]$ is already threaded, or r is p. ∎

Lemma 23 places a bound on the number of cells that can be threaded while an operation is in progress. We now give a sequence of lemmas showing that when P finishes scanning the *head* array, either $announce[P]$ is threaded, or $head[P]$ lies within $n + 1$ cells of the end of the list.

Lemma 24 *The following assertion is invariant:*

$$\max(head) \geq start(P).$$

Proof: The sequence number for each $head[Q]$ is non-decreasing. ∎

```
universal(what: INVOC) returns(RESULT)
   mine: cell := [seq: 0,
                  inv: what,
                  new: create(consensus_object),
                  before: create(consensus_object)
                  after: null]
   ⟨announce[P] := mine; start(P) := max(head)⟩
   for each process Q do
       head[P] := max(head[P], head[Q])
       end for
   while announce[P].seq = 0 do
          c: *cell := head[P]
          help: *cell := announce[(c.seq mod n) + 1]
          if help.seq = 0
             then prefer := help
             else  prefer := announce[P]
             end if
          d := decide(c.after, prefer)
          decide(d.new, apply(d.inv, c.new.state))
          d.before := c
          d.seq := c.seq + 1
          ⟨head[P] := d; (∀Q) concur(Q) := concur(Q) ∪ {d}⟩
          end while
   ⟨head[P] := announce[P]; (∀Q) concur(Q) := concur(Q) ∪ {d}⟩
   return (announce[P].new.result)
   end universal
```

Figure 14: A Universal Construction

Lemma 25 *The following is a loop invariant for Statement #3:*

$$\max(head[P], head[Q], \ldots, head[n]) \geq start(P).$$

where Q is the loop index.

Proof: When Q is 1, the assertion is implied by Lemma 24. The truth of the assertion is preserved at each iteration, when $head[P]$ is replaced by $\max(head[P], head[Q])$. ∎

Lemma 26 *The following assertion holds just before Statement #4:*

$$head[P].seq \geq start(P).$$

Proof: After the loop at Statement #3, $\max(head[P], head[Q], \ldots, head[n])$ is just $head[P].seq$, and the result follows from Lemma 25. ∎

Lemma 27 *The following is invariant:*

$$|concur(P)| \geq head[P].seq - start(P) \geq 0$$

Proof: The lower bound follows from Lemma 26, and the upper bound follows from Equation 1. ∎

Theorem 28 *The protocol in Figure 14 is correct and bounded wait-free.*

Proof: Linearizability is immediate, since the order in which cells are threaded is clearly compatible with the natural partial order of the corresponding operations.

The protocol is bounded wait-free because P can execute the main loop no more than $n+1$ times. At each iteration, $head[P].seq$ increases by one. After $n+1$ iterations, Lemma 27 implies that

$$|concur(P)| \geq head[P].seq - start(P) \geq n.$$

Lemma 23 implies that $announce[P]$ must be threaded. ∎

4.2 Memory Management

In this section we discuss how cells are allocated and reclaimed. To reclaim a cell, we assume each consensus object provides a *reset* operation that restores the object to a state where it can be reused for a new round of consensus. Our construction resets a consensus object only when there are no concurrent operations in progress.

The basic idea is the following: a process executing an operation will traverse no more than $n+1$ cells before its cell is threaded (Theorem 28). Conversely, each cell will be traversed no more than $n+1$ times. When a process is finished threading its cell, it *releases* each of the $n+1$ preceding cells by setting a bit. When a cell has been released $n+1$ times, it is safe to recycle it. Each cell holds an additional field, an array *released* of $n+1$ bits, initially all *false*. When a process completes an operation, it scans the $n+1$ earlier cells, setting *released*$[i]$ to *true* in the cell at distance i.

Each process maintains a private pool of cells. When a process needs to allocate a new cell, it scans its pool, and reinitializes the first cell whose *released* bits are all *true*. We assume here that each object has its own pool; in particular, the cell's new sequence number exceeds its old sequence number. While a process P is allocating a new cell, the list representing an object includes at most $n-1$ incomplete operations, and each such cell can inhibit the reclamation of at most $n+1$ cells. To ensure that P will find a free cell, it needs a pool of at least n^2 cells. Note that locating a free cell requires at worst $O(n^3)$ *read* operations, since the process may have to scan n^2 cells, and each cell requires reading $n+1$ bits. If an atomic *fetch&add* operation is available, then a counter can be used instead of the *released* bits, and a free cell can be located in $O(n^2)$ *read* operations.

The proof of Lemma 23 remains unchanged. For Lemma 24, we observe that a cell can be reclaimed only if it is followed in the list by at least $n+1$ other cells, hence reclaiming a cell cannot affect the value of $\max(head)$. The statement of Lemma 26 needs to be strengthened:

Lemma 29 *The following assertion holds just before Statement #4:*

$$announce[P] \in head \lor head[P].seq \geq start(P).$$

Proof: When P announces its cell, there is some process Q such that $head[Q]$ has sequence number greater than or equal to $start(P)$. This cell can be reclaimed only if $n+1$ other cells are threaded in front of it, implying that $|concur(P)| \geq n+1$, and hence that $announce[P] \in head$ (Lemma 23). ∎

The proof of Theorem 28 proceeds as before. There is one last detail to check: if P's cell has not been threaded by the time it finishes scanning *head*, then we claim that none of the cells it traverses will be reclaimed while the operation is in progress. Lemma 23 states that the list cannot have grown by more than n cells since P's announcement, thus every cell reachable from $head[P]$ lies within $n + 1$ cells of the end of the list, or of $announce[P]$ if it is threaded. In either case, those cells cannot be reclaimed while P's operation is in progress, since they must have at least one *released* bit unset.

4.3 Remarks

The first universal construction [12] used unbounded memory. Plotkin [27] describes a universal construction employing "sticky-byte" registers, a kind of write-once memory. In Plotkin's construction, cells are allocated from a common pool, and reclaimed in a way similar to ours. The author [13] describes a universal construction using *compare&swap* that is currently being implemented on a multiprocessor.

A *randomized wait-free* implementation of a concurrent object is one that guarantees that any process can complete any operation in a finite *expected* number of steps. Elsewhere [2], we give a randomized consensus protocol using atomic registers whose expected running time is polynomial in the number of processes. This protocol has several important implications. If the wait-free guarantee is allowed to be probabilistic in nature, then the hierarchy shown in Figure 1 collapses because atomic registers become universal. Moreover, combining the randomized consensus protocol with our universal construction yields a polynomial-time randomized universal construction. Bar-Noy and Dolev [3] have adapted our randomized consensus protocol to a message-passing model; that protocol can be used to manage randomized wait-free *replicated* data objects.

5 Conclusions

Wait-free synchronization represents a qualitative break with the traditional locking-based techniques for implementing concurrent objects. We have tried to suggest here that the resulting theory has a rich structure, yielding a number of unexpected results with consequences for algorithm design, multiprocessor architectures, and real-time systems. Nevertheless, many interesting problems remain unsolved. Little is known about lower bounds for universal constructions, both in terms of time (rounds of consensus) and space (number of cells). The *implements* relation may have additional structure not shown in the impossibility hierarchy of Figure 1. For example, can atomic registers implement any object with consensus number 1 in a system of two or more processes? Can *fetch&add* implement any object with consensus number 2 in a system of three or more processes? Does the *implements* relation have a different structure for bounded wait-free, wait-free, or non-blocking synchronization? Finally, little is known about practical implementation techniques.

Acknowledgments

I am grateful to Jim Aspnes, Vladimir Lanin, Michael Merritt, Serge Plotkin, Mark Tuttle, Jennifer Welch, and the anonymous referees for many invaluable suggestions.

References

[1] J.H. Anderson and M.G. Gouda. The virtue of patience: Concurrent programming with and without waiting. Private Communication.

[2] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. Technical Report CMU-CS-88-205, CMU Computer Science Dept., December 1988. To appear, Journal of Algorithms.

[3] A. Bar-Noy and D. Dolev. Shared memory vs. message-passing in an asynchronous distributed environment. In *Eighth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 307–318, August 1989.

[4] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[5] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[6] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[7] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[8] G.H. Pfister et al. The ibm research parallel processor prototype (rp3): introduction and architecture. In *International Conference on Parallel Processing*, 1985.

[9] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[10] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer – designing an mimd parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[11] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[12] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[13] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 1990. To appear.

[14] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[15] C.P. Kruskal, L. Rudolph, , and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[16] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[18] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[19] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[20] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[21] M.C. Loui and H.H. Abu-Amara. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes*, volume 4, pages 163–183. JAI Press, 1987.

[22] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, M.I.T. Laboratory for Computer Science, November 1988.

[23] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[24] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[25] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[26] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[27] S.A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–176, 1989.

[28] C.L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1), January 1985.

[29] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, 1987.

[30] H.S. Stone. Database applications of the fetch-and-add instruction. *IEEE Transactions on Computers*, C-33(7):604–612, July 1984.

[31] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of of the 27th IEEE Symposium on Foundations of Computer Science*, pages 223–243, 1986. See also errata in SIGACT News 18(4), Summer, 1987.