Impossibility and Universality Results for Wait-Free Synchronization.

Conference Paper · January 1988

DOI: 10.1145/62546.62593 · Source: DBLP

Maurice Herlihy
Brown University

1-1-1988

Impossibility and universality results for wait-free synchronization

Maurice Herlihy
Carnegie Mellon University

Impossibility and Universality Results for Wait-Free Synchronization

Maurice P. Herlihy

12 May 1988
CMU-CS-88-140

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 atomic read/write registers, multiprocessor architectures, and concurrent data structures. 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-and-set* and *fetch-and-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.

This paper will be published in the Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1988.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the:
Avionics Laboratory
Air Force Wright Aeronautical Laboratories
Aeronautical Systems Division (AFSC)
Wright-Patterson AFB, OHIO 45433-6543

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

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 halts in a critical section, non-faulty processes will also be unable to progress. Even in failure-free systems, processes may be subject to unexpected delay. For example, if a process executing in a critical region takes a page fault, exhausts its scheduling quantum, or is swapped out, then other processes needing that resource will also be delayed. Similar problems arise in heterogeneous architectures, where some processors may be faster than others.

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 read/write registers from non-atomic "safe" registers [16], complex atomic read/write registers from simpler atomic registers [3, 4, 13, 21, 23, 24, 27, 29], read-modify-write operations from combining networks [8, 12], and typed objects such as queues or sets from simpler objects [10, 15, 17].

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 1-1). The basic idea is the following: we identify a property of X that is necessarily shared by any object that implements X, hence any object that does not satisfy that property cannot implement X. The property we use is the ability to solve a simple consensus problem.

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 simple *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 if and only if it can solve consensus. Figure 1-1 shows a number of simple universal objects: 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 [3, 4, 13, 16, 21, 23, 24, 27, 29]. 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 read/write registers have few, if any, interesting applications in this area. From a set of atomic read/write 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-and-set*, *compare-and-swap*, and *fetch-and-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 [9] has investigated architectural support for wait-free implementations of common synchronization primitives. They use combining networks to implement *fetch-and-add*, a generalization of *test-and-set*. IBM's RP3 [25] project is investigating a similar approach. The *fetch-and-add* operation is quite flexible: it can be used for semaphores, for highly concurrent queues, and even for database synchronization [8, 10, 28]. Nevertheless, we show that it is not universal, disproving a conjecture of Gottlieb et al. [8]. We also show that message-passing architectures such as hypercubes [11, 26] 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.

Processes	Object
1	read/write registers
2	test-and-set, swap, fetch-and-add, queue, stack
...	...
2n-2	n-register assignment
...	...
Unbounded	memory-to-memory move and swap, augmented queue, compare-and-swap, fetch-and-cons

Figure 1-1: 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* [10]: 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. Linearizability unifies and generalizes many of the *ad hoc* correctness conditions in the literature on concurrent objects.

2.1. I/O Automata

Formally, we model processes and objects as I/O automata [19, 20]. An *I/O automaton* is a non-deterministic automaton A with the following components:[1]

[1] To remain consistent with the terminology of [10], we use "event" where Lynch and Merritt use "operation," and "history" where they use "schedule."

- *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,*
- *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* of an automaton A is a finite sequence s_0, e_1, s_1, ..., e_n, s_n or infinite sequence s_0, e_1, s_1, ... of alternating states and events such that s_0 is a starting state and each (s_i, e_{i+1}, s_{i+1}) is a step of A. A *history* of an automaton is the subsequence of events occurring in one of its executions.

A new I/O automaton can be constructed by *composing* a set of I/O automata with disjoint output 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, and the set of output events of S, *out(S)*, is the union of the components' sets of output events. The sets of input events of S, *in(S)*, is *events(S) – out(S)*, all the 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 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. If H is a history of a composite automaton and S a set of component automata, H | S denotes the subhistory of H consisting of events of automata in S. If A is an automaton, we use H | A as shorthand for H | {A}.

2.2. Sequential Systems

In this section, we define "sequential systems" that model the behavior of processes and objects in the absence of concurrency. We use sequential systems to define the basic notion of correctness for the more complex concurrent systems defined in the next section. A sequential system consists of processes, objects, and a sequential scheduler, shown schematically in Figure 2-1. Processes represent sequential threads of control, objects represent data structures shared by processes, and the sequential scheduler mediates all communication between processes and objects, ensuring that objects execute operations in a one-at-a-time, serial order. All components are I/O automata. (Lynch and Merritt [19] use a similar decomposition to model nested transaction systems.)

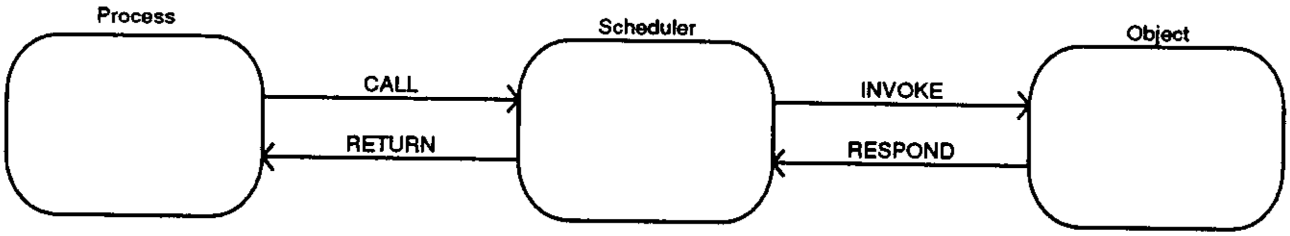

Figure 2-1: Schematic View of Processes, Objects, and the Scheduler

An *object* X has two events: the input event INVOKE(P, op, X), where P is a process and *op* is an operation[2] of the object, and the output event RESPOND(P, res, X), where *res* is a result value. Two INVOKE and RESPONSE events *match* if their process and object names agree. An invocation is *pending* if it is not followed by a matching response.

A *process* P has the output event CALL(P, op, X), where *op* is an operation of object X, and the input event RETURN(P, res, X), where *res* is a result value. Two CALL and RETURN events *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 a CALL event and alternates matching CALL and RETURN events.

The *sequential scheduler*, shown in Figure 2-2, has two input events: CALL(P, op, X) and RESPOND(P, res X), which are identified with the corresponding output events of processes and objects, and two output events: RETURN(P, res, X) and INVOKE(P, op X), which are identified with the corresponding input events of processes and objects. A state *s* of the sequential scheduler consists of three components: *called(s)* is a set of triples (P, op, X), initially empty, *responded(s)* is a set of triples (P, res, X), initially empty and *mutex(s)* is a value in {*busy*, *idle*}, initially *idle*. In postconditions, *s* denotes the object's new state, and *s'* its old state. When process P calls an operation of object X, the scheduler records the call in its state. When no other operation is in progress, the scheduler relays the invocation to the object and marks the system as busy. When the object returns the response, the scheduler marks the system as idle and records the response, which it later forwards to the calling process.

```
CALL(P, op, X)
  Postcondition:
      called(s) = called(s') ∪ {(P, op, X)}

RETURN(P, res, X)
  Precondition:
      (P, res, X) ∈ responded(s')
  Postcondition:
      responded(s) = responded(s') – {(P, res, X)}

INVOKE(P, op X)
  Precondition:
      mutex(s') = idle
      (P, op, X) ∈ called(s')
  Postcondition:
      called(s) = called(s') – {(P, op, X)}
      mutex(s) = busy

RESPOND(P, res X)
  Postcondition:
      responded(s) = responded(s') ∪ {(P, res, X)}
      mutex(s) = idle
```

Figure 2-2: The Sequential Scheduler

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

A *sequential system* $\{P_1, ..., P_n; A_1, ..., A_m\}$ is an I/O automaton composed from processes $P_1, ..., P_n$, objects $A_1, ..., A_m$, and the sequential scheduler. A history of a sequential system is *well-formed* if each $H \mid P_i$ is well-formed, and a sequential system is *well-fomed* if each of its histories is well-formed. Henceforth, we restrict our attention to well-formed sequential systems.

The behavior of an object in a sequential system can be specified in a particularly simple way: by giving pre- and postconditions for each operation. We refer to an object whose behavior is specified only for sequential systems as a *sequential object*. In this paper, we consider only sequential objects whose operations are *total*: if the object has a pending invocation, 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.

2.3. Concurrent Systems

Sequential systems permit an inadequate level of concurrency; we use them only to define correctness for more realistic "concurrent systems." A *concurrent system* $\{P_1, ..., P_n; A_1, ..., A_m\}$ is an I/O automaton composed from process automata $P_1, ..., P_n$, object automata $A_1, ..., A_m$, and a *concurrent scheduler*. The concurrent scheduler is very simple: it asynchronously but reliably relays invocations from processes to objects and results from objects to processes. It is constructed from the sequential scheduler in Figure 2-2 simply by omitting the mutex component of the state, together with every pre- and postcondition that mentions it. Well-formedness is defined exactly as for sequential systems.

A concurrent system $\{P_1, ..., P_n; A_1, ..., A_m\}$ is *linearizable* if, for each of its histories H, there exists a history S of the corresponding sequential system such that:

$$H \mid \{P_1, ..., P_n\} = S \mid \{P_1, ..., P_n\}$$

In other words, the system "appears" sequential to the ensemble of processes. Each operation appears to take effect instantaneously at some point between its invocation and its response. A concurrent object A_j is *linearizable* if every concurrent system $\{P_1, ..., P_n; A_j\}$ 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 [14] or strict serializability [22], linearizability is a *local* property: a concurrent system is linearizable if and only if each individual object is linearizable [10].

2.4. Implementations

An *implementation* of an object A is a concurrent system $\{F_1, ..., F_n; R\}$, where the F_i are called *front-end* automata, and R is a called the *representation object*. Informally, process P_i executes an operation of A by sending the invocation to F_i, which in turn applies a sequence of operations to R before returning a result to P_i. As usual, all communication is mediated by the concurrent scheduler. Figure 2-3 illustrates an object implementation; the scheduler is omitted for clarity.

An implementation is *wait-free* if:

- There is no history in which a pending invocation of P_i is followed by an infinite number of steps of F_i.

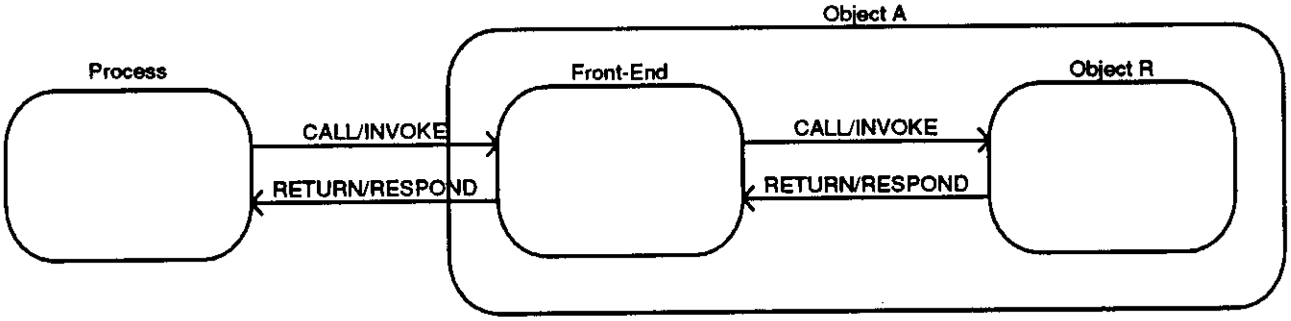

Figure 2-3: Schematic View of Object Implementation

- If P_i has a pending invocation, but F_i does not, then F_i has an enabled output.

The first condition rules out unbounded busy-waiting: any sequence of representation operations that implements an abstract operation is finite. The second condition, together with the assumption that R is total, asserts that neither F_i nor R can enter a halted state while an operation of A is in progress, ruling out conditional waiting.

An implementation is *strongly wait-free* if the sequence of representation operations that implements any abstract operation has bounded (as opposed to finite) length. Strongly wait-free implies wait-free, but not vice-versa. We use the wait-free condition for impossibility results, and the strongly 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 a "universal" object U such that for any linearizable object X, U implements X.

3. Impossibility Results

Informally, a *consensus protocol* is a system of *n* processes that communicate through a shared object X. The processes each start with an input value from some domain *D*, they communicate with one another by applying operations to X, and they eventually agree on a common input value and halt. Our notion of a consensus protocol is essentially the same as that of Fisher, Lynch, and Paterson [7], except we have found it convenient to treat consensus as a form of election: we take *D* to be the set of process names, and we assume each process uses its own name as its input value.

More precisely, an *n-process consensus protocol for X* is a concurrent system $\{P_1, \ldots, P_n; X\}$, where each P_i has an additional output event: DECIDE(P_i, v), for v in $\{P_1,\ldots,P_n\}$. A history for a protocol has *decision value v* if it includes a DECIDE(P_i, v) event. A protocol is *partially correct* if:

1. No history has more than one decision value.
2. If a history has decision value P_i, then P_i took at least one step.

The second condition rules out the trivial protocol where each process makes a predefined choice. A protocol state is *bivalent* if either decision value is still possible, otherwise it is *univalent*. An *X-valent* state is a univalent state with eventual decision value X. A *decision step* carries a protocol from a bivalent to a univalent state.

A partially correct protocol is *wait-free* if it satisfies two conditions:

1. No process takes an infinite number of steps without deciding.
2. If an undecided process has no pending invocation, then it has an enabled output transition (either a decision or another invocation).

The second condition rules out the trivial protocol where one process non-deterministically chooses a value, and the others immediately halt without deciding.

For brevity, we say "X solves *n*-process consensus" if there exists a wait-free *n*-process consensus protocol for X. It is an immediate consequence of our definitions that if Y implements X, and X solves *n*-process consensus, then "substituting" Y for X yields an *n*-process consensus protocol for Y. Consequently:

Theorem 1: If X solves *n*-process consensus but Y does not, then there exists no wait-free implementation of X by Y in a system of *n* or more processes.

In the rest of this section, we consider a number of objects, displaying wait-free consensus protocols for some, and impossibility results for others. For brevity, processes and objects are defined informally by pseudo-code; their translations into I/O automata should be self-evident.

3.1. Atomic Read/Write Registers

Theorem 2: There is no wait-free solution to two-process consensus by atomic read/write registers.

Proof: We assume such a protocol and derive a contradiction. The initial protocol state is bivalent by assumption. Consider the following execution, which starts in the initial state, and leaves the protocol in a bivalent state. In the first stage, run process P until it reaches a state where it cannot continue without executing a decision step. P must eventually reach such a state, since the wait-free condition ensures that it has an enabled step in every bivalent state, and that it cannot run forever. In the second stage, run Q until it reaches a similar state, and in successive stages, alternate running P and Q 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 step of either process is a decision step. Since *s* is bivalent, some step *p* of P carries the protocol to an X-valent state *s'*, and some step *q* of Q carries the protocol to a Y-valent state, where X and Y are distinct.

We first argue that the decision step cannot be a CALL. Since *q* carries *s* to a Y-valent state, there exists an execution σ from *s* in which Q chooses Y, and P is idle. Suppose *p* is a CALL step. Because *s* and *s'* differ only in the internal states of P and the scheduler, σ is also an execution from *s'* in which Q chooses Y, an impossibility since *s'* is X-valent. A symmetric argument shows that the decision step cannot be a RETURN.

The decision step must be an INVOKE or RESPOND step. Since registers are linearizable, we can consider complete *read* and*write* operations.

- Neither operation can be a *read*. Suppose *p* is a *read* of a shared register. Since *q* carries *s* to a Y-valent state, there exists an execution σ from *s* consisting entirely of steps of Q that yields the decision value Y. Because *s* and *s'* differ only in the internal states of P and the scheduler, σ is also an execution from *s'* in which Q chooses Y, an impossibility since *s'* is X-valent.
- Both operations cannot be writes. If the processes write to different shared registers, the state that results if *p* is immediately followed by *q* is identical to the state that results

> if q is immediately followed by p, which is impossible, since one is X-valent and the other is Y-valent. Suppose the processes write to the same shared register. Since s' is X-valent, there exists an execution σ from s' consisting entirely of steps of P that yields the decision value X. Let s'' be the Y-valent state reached by executing q followed by p. Because p overwrites the value written by q, s' and s'' differ only in the internal states of Q and the scheduler, and therefore σ is also an execution from s'' in which P chooses X, an impossibility since s'' is Y-valent.

Similar results have been shown by Loui and Abu-Amara [18], Chor, Israeli, and Li [5], and Anderson and Gouda [2]. Our contribution lies in the following corollary:

> **Corollary 3:** If X solves two-process consensus, then it is impossible to construct a wait-free implementation of X from atomic read/write registers.

Fischer, Lynch, and Paterson [7] have shown that there is no wait-free solution to two-process consensus by message channels that permit messages to be delayed and reordered. That result does not imply Theorem 2, however, because atomic read/write registers lack certain commutativity properties of asynchronous message buffers. (In particular, Lemma 1 of [7] does not hold.)

Dolev, Dwork, and Stockmeyer [6] 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 [16] 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 read/write registers implement safe read/write registers, safe read/write registers cannot solve two-process consensus, and hence the impossibility results we derive for atomic read/write registers apply equally to safe read/write registers.

3.2. Read-Modify-Write Operations

Kruskal, Rudolph, and Snir [12] 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 following procedure, which is executed atomically:

```
RMW = proc(r: register, f: function) returns(value)
  temp := r;
  r := f(r);
  return(temp);
  end RMW
```

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-and-set*, *swap*, *compare-and-swap*, and *fetch-and-add*. Numerous others are given in [12].

Loui and Abu-Amara [18] give a number of constructions and impossibility results for consensus protocols using shared read-modify-write registers, which they call "test-and-set" registers. Among other results, they show that n-process consensus for $n > 2$ cannot be solved by any read-modify-write operations on single-bit registers.

> **Theorem 4**: Two-process consensus can be solved by a register supporting any non-trivial read-modify-write operation.
>
> **Proof**: Since f is not the identity, there exists a value v such that $v \neq f(v)$. Let P and Q be the two processes, and let the shared register `r` be initialized to v. P executes Decide_P, and Q executes Decide_Q:

```
Decide_P = proc(r: register)            Decide_Q = proc(r: register)
  if RMW(r,f) = v                         if RMW(r,f) = v
    then decide (0)                         then decide (1)
    else decide (1)                         else decide (0)
    end                                     end
  end Decide_P                            end Decide_Q
```

> The protocol chooses 0 if P's operation is linearized first, and 1 otherwise.
>
> **Corollary 5**: 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.

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-and-set*, *swap*, and *fetch-and-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 6**: 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 1, construct an execution leaving the protocol in bivalent state where every step enabled for P and Q is a decision step, some step of P carries the protocol to an X-valent state, and some step of Q carries the protocol to a Y-valent state, where X and Y are distinct. By the usual case analysis, 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 execution σ consisting entirely of steps 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', σ will also yield the decision value X 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 execution σ consisting entirely of steps 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', σ will also yield the decision value X 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 that solves n-process consensus for $n > 2$.

Another classical primitive is *compare-and-swap*, which takes two values, v and v'. If the register's current value is v, it is replaced by v', otherwise is left unchanged. The register's old value is returned.

Theorem 7: *compare-and-swap* solves n-process consensus for arbitrary n.

Proof: The register is initialized to $\perp$, and process P_i executes

```
Decide_i = proc(r: register)
  prefer: bool := i
  old: bool := compare-and-swap(r, ⊥, prefer)
  if old = ⊥
      then decide(prefer)
      else decide(old)
      end
  end Decide_i
```

Corollary 8: It is impossible to construct a wait-free implementation of a *compare-and-swap* register from a set of registers that support any combination of *read*, *write*, *test-and-set*, *swap*, or *fetch-and-add* operations.

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 9: Two-process consensus can be solved by a FIFO queue.

Proof: The queue is initialized by enqueuing the value *first* followed by the value *second.*

```
Decide_P = proc(q: queue)
  if deq(q) = first
    then decide (0)
    else decide (1)
    end
  end Decide_P
```

```
Decide_Q = proc(q: queue)
  if deq(q) = first
    then decide (1)
    else decide (0)
    end
  end Decide_Q
```

P and Q each attempt to dequeue the first item in the queue; if P succeeds, the protocol decides on 0, otherwise it decides on 1.

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

Corollary 10: 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.

Lamport [15] 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 2 implies that Lamport's queue cannot be extended to permit concurrent *deq* operations without augmenting the *read* and *write* operations with more powerful primitives.

Theorem 11: There is no wait-free solution to three-process consensus using FIFO queues.

Proof: As before, we maneuver the protocol to a state where the next process to execute an operation establishes the decision value. Let P, Q, and R be the processes, and 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 Q dequeues, and let *s'* be the state if the dequeues occur in the opposite order. Since *s* is X-valent, there exists an execution σ starting in *s*, consisting entirely of steps of R, and yielding the decision value X. But *s* and *s'* differ only in the internal states of P and Q, thus σ is also an execution from *s'* yielding decision value X, 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. If the queue is empty, then the Y-valent state reached if Q dequeues and P enqueues is indistinguishable to R from the X-valent state reached if P enqueues.

Finally, suppose all three do *enq* operations. Let *s* be the state at the end of the following execution:

1. P, Q, and R enqueue in that order.
2. Run P until it completes a *deq*.
3. Run Q until it completes a *deq*.

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

1. Q, P, and R enqueue in that order.
2. Run P until it completes a *deq*.
3. Run Q until it completes a *deq*.

Clearly, *s* is X-valent and *s'* is Y-valent. Since the only way to observe the queue's state is via the *deq* operation, both of P's executions are identical until its *deq* returns. Since P is halted before it can modify any other objects, Q's executions are also identical until its *deq* returns. By a now-familiar argument, a contradiction arises because *s* and *s'* are indistinguishable to R.

The same result holds for many similar data types such as sets, stacks, or priority queues.

A *message-passing architecture* (e.g., a hypercube, [11, 26]) is a set of processors that communicate via shared FIFO queues. Theorem 11 implies that message-passing architectures cannot solve three process consensus or implement any object that can. Dolev, Dwork, and Stockmeyer [6] give a related result: point-to-point FIFO message channels cannot solve two-process consensus. That result does not imply Theorem 11, 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 12: The augmented queue solves *n*-process consensus for arbitrary *n*.

Proof: The queue *q* is initialized to *empty*, and each process enqueues its own identifier. Process P_i executes:

```
Decide_i = proc(r: register)
  enq(q, i)
  decide (peek(q))
  end Decide_i
```

The process whose *enq* is ordered first establishes the decision value.

Corollary 13: It is impossible to construct a wait-free implementation of the augmented queue from a set of registers that support any combination of *read, write, test-and-set, swap,* or *fetch-and-add* operations.

Corollary 14: It is impossible to construct a wait-free implementation of the augmented queue from a set of regular queues.

Elsewhere [10], we have given an implementation of a FIFO queue using *read*, *fetch-and-add*, and *swap* operations that permits an arbitrary number of concurrent *enq* and *deq* operations. (Although this queue does not use mutual exclusion, it is not wait-free, since a *deq* applied to an empty queue busy-waits until an item is enqueued.) Corollary 13 implies that this queue implementation cannot be extended to support a wait-free *peek* operation without making use of more powerful primitives.

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.

Theorem 15: *Move* solves *n*-process consensus for arbitrary *n*.

Proof: We first give a two-process protocol. Let `r1` and `r2` be respectively initialized to *1* and *2*. P_1 and P_2 respectively execute Decide_1 and Decide_2:

```
Decide_1 = proc(r1,r2: register)      Decide_2 = proc(r1,r2: register)
  r2 := 1;                              move(r2,r1)
  decide(r1)                            decide(r1)
  end Decide_1                          end Decide_2
```

The protocol decides 2 if P_2's *move* is linearized before P_1's *write*, and 1 otherwise.

To generalize this protocol to *n* processes, let `r[1..n,1..2]` be an array of registers, where `r[i,1]` is initialized to *i*, and `r[i,2]` to *i-1*. Process P_i executes the following procedure:

```
Decide_i = proc(r: array[array[[register]])
  move(r[i,1], r[i,2])
  for j in i+1 .. n do
    r[j, 1] := j-1
    end
  for j in n .. 1 do
    if r[j,2] = j then decide (j) end
    end
  end Decide_i
```

Each process iterates the two-process protocol given above. In the first round (the *move* operation), the process achieves consensus with the lower-numbered processes, and in subsequent rounds it achieves consensus with successively higher-numbered processes. Consensus round *i* was won by P_i if `r[i,2]` is *i*, otherwise it was won by a lower-numbered process. The overall winner is the highest-numbered process to win its first round.

Theorem 16: The memory-to-memory *swap* operation[3] solves *n*-process consensus for arbitrary *n*.

Proof: The processes share an array of registers `p[1..n]` whose elements are initialized to *0*, and a single register `r`, initialized to *1*. Process P_i executes:

[3] 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.

```
Decide_i = proc(p: array[register], r: register)
  swap(p[i],r)
  for k: int in 1 .. n do
    if p[k] = 1 then decide (k) end
    end
  end Decide_i
```

The first process to swap *1* into p wins.

Corollary 17: 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-and-set, swap,* or *fetch-and-add* operations.

Corollary 18: It is impossible to construct a wait-free implementation of memory-to-memory *move* or *swap* from a set of FIFO queues.

3.6. Multiple Assignment

The expression:

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

atomically assigns each value v_i to each register r_i.

Theorem 19: Atomic *n*-register assignment solves *n*-process consensus.

Proof: The protocol uses *n* "private" registers $r_1, \ldots, r_n$, where P_i writes to register r_i, and $n(n-1)/2$ "shared" registers r_{ij}, where $i > j$, where P_i and P_j both write to registers r_i and r_j. All registers are initialized to ⊥. Each process atomically assigns its input value to *n* registers: its private register and its *n-1* shared 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 ⊥, then neither assignment has occurred, and we are done.
- At least one assignment has occurred. Read r_i and r_j. If r_i's value is ⊥, then P_j precedes P_i, and similarly for r_j. Otherwise, let *v* be the value of r_i.
- Both assignments have occurred. Reread r_{ij}. If its value is now *v*, P_j went first, otherwise P_i went first.

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

This result can be improved.

Theorem 20: Atomic *n*-register assignment solves *2n-2*-process consensus.

Theorem 21: This protocol has two phases. Each process has two private registers, one for each phase, and each pair of processes share a register. Divide the processes into two groups of *n-1*. In the first phase, each group uses *n-2*-register assignment to achieve consensus within itself, using the previous theorem's protocol. In the second phase, each process atomically assigns its group's value to its phase-two private register and the *n-1* 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 decides on that process's value. At least one process has performed an assignment, thus G has edges. Let P_n be the process whose assignment is first in the linearization order. P_n 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 n-register assignment cannot solve $2n-1$-process consensus.

Proof: By the usual construction, we can maneuver the protocol into a bivalent state s in which any subsequent step 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 "private" 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 registers 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. Let σ be an execution, starting in s', that consists entirely of steps of Q and that yields the decision value X. Because the values of the registers are identical in s' and s'', σ also yields the decision value X from s'', a contradiction.

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 an execution σ of P that yields the 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 σ also yields the decision value X from s'', a contradiction.

It follows that if P has default value X, and there are k processes with default value Y, then any protocol requires P to assign to $k+1$ registers. If there are $2n-1$ processes, then we can minimize the number of registers any process must assign to if n processes have default X and $n-1$ have default Y, implying that each process in the first class must assign to $n+1$ registers.

These last two theorems show that there is a sense in which consensus is irreducible: if n is even, one cannot achieve consensus among n processes by calling subroutines that achieve consensus among m processes, where $m < n$.

4. Universality Results

In this section, we show that there exist *universal* objects from which one can construct a wait-free implementation of any object. How is it possible to provide a universal implementation for behaviors as disparate as those of queues, databases, counters, etc.? We use a two-step reduction. First, we show that we can systematically transform a sequential implementation of an object into a wait-free concurrent implementation if (and only if) we can atomically thread an item onto the front of a list. Second, we show that we can implement this *fetch-and-cons* operation if (and only if) we can solve n-process consensus. This reduction greatly facilitates universality proofs, since proving that X solves n-process consensus is easier than proving directly that X implements *fetch-and-cons*, and considerably easier than proving directly that X is universal.

4.1. Reduction to Fetch-And-Cons

A sequential object is *deterministic* if the result returned by each RESPONSE step is completely determined by the object's history of INVOKE steps. We first consider deterministic objects, and then introduce refinements for non-deterministic objects. As usual, all operations must be total.

Informally, our strategy is the following. We represent the object's state as a *list* of the invocations that have been applied to it, placing the most recent invocation at the head of the list. To reconstruct the object's abstract state, the deterministic sequential implementation is used to "replay" the list of invocations. A *list* object provides the usual operations: *cons, car, cdr, null*, etc. The only operation that destructively modifies a list is the read-modify-write operation *fetch-and-cons*, which atomically (1) places an item at the head of the list, and (2) returns the list of items that *follow* the new item. A process executes an operation in two steps. First, it uses *fetch-and-cons* to place the operation at the head of the list. This step is when the operation "really happens," in the sense that it determines the operation's position in the linearization order. Second, the process computes the operation's result after traversing the list to reconstruct the object's previous state.

Let OP be the object's domain of operations, RES its domain of results, and STATE its domain of states. Any sequential object whose operations are deterministic and total defines two functions:

$$\text{eval: OP}^* \rightarrow \text{STATE}$$

yields the object state after executing a sequence of operations, and:

$$\text{apply: OP} \times \text{STATE} \rightarrow \text{RES}$$

yields the response to an invocation in a particular state. Because the object is deterministic and total, these functions are total and well-defined.

A state *s* of process P's front-end (Figure 4-1) consists of three components: *incoming* is an element of OP $\cup \perp$, initially $\perp$, *outgoing* an element of RES $\cup \perp$, initially $\perp$, and *pending* a boolean value. A state *s* of the representation object (Figure 4-2) is a sequence (list) of operations *log*, initially empty, and *replyto*, in PROCESS $\cup \{\perp\}$, initially $\perp$. In the transition relations, "•" denotes concatenation of sequences, and "cdr" returns the sequence following its argument's first element.

To implement a non-deterministic object, we simply choose a deterministic implementation. For example, a *set* object with a non-deterministic *remove* operation can be implemented as a stack or a queue. Probabilistic properties of non-deterministic operations can be ensured by providing randomly-generated arguments to deterministic invocations. For example, to make *remove* equally likely to return any item in the set, the set implementation can choose a value *r* from a uniform distribution, and deterministically remove the *r*-th element modulo the set size.

This *fetch-and-cons* construction is wait-free, but not strongly wait-free, since the *k*-th operation requires *k* steps to replay the list. We can make this construction strongly wait-free by having each process truncate the list as it completes each operation. We allow each element in the list to be either an operation or a state. A process executes an operation just as before, but before it returns it destructively modifies the list, replacing the *cdr* of its operation with its newly-reconstructed state. The *eval* function is extended in the obvious way, returning immediately when it encounters a state in place of an operation. With this change, the number of operations in a list is at most *n*, since a front-end will replay at most *n* operations before it encounters a state.

```
CALL(P, op, A)
     where $op \in OP$
  Postcondition:
       $incoming = op$

INVOKE(P, fetch-and-add(op), R)
     where $op \in OP$
  Precondition:
       $incoming = op$
       $pending = false$
  Postcondition:
       $pending = true$

RESPONSE(P, log, R)
     where $log \in OP^*$
  Postcondition:
       $outgoing = apply(incoming', eval(log))$
       $pending = false$

REPLY(P, res, A)
     where $res \in RES$
  Precondition:
       $outgoing = res$
  Postcondition:
       $outgoing = \perp$
```

Figure 4-1: Front-end Automaton

```
INVOKE(P, fetch-and-add(op), R)
     where $op \in OP$
  Postcondition:
       $log = op \bullet log'$
       $replyto = P$

RESPOND(P, list, R)
  Precondition:
       $replyto = P$
       $list = cdr(log')$
  Postcondition:
       $replyto = \perp$
```

Figure 4-2: Representation Automaton

In a real implementation, *fetch-and-cons* would return a pointer to a list, not the list itself. When can an element of such a shared list be discarded? Because a front-end executing an operation will traverse no more than *n* list elements following its own operation, it is safe to discard any state elements whose *n* immediate predecessors in the list are also state elements. Since there can exist at most *n* operations in a list, and each of those can prevent the reclamation of an additional *n* elements, the worst-case space complexity of the object is reduced to $O(n^2)$.

One way to show that an object is universal is to give a direct implementation of *fetch-and-cons.* For example, Figures 4-3 and 4-4 show a constant-time implementation of *fetch-and-cons* by memory-to-memory *swap.*

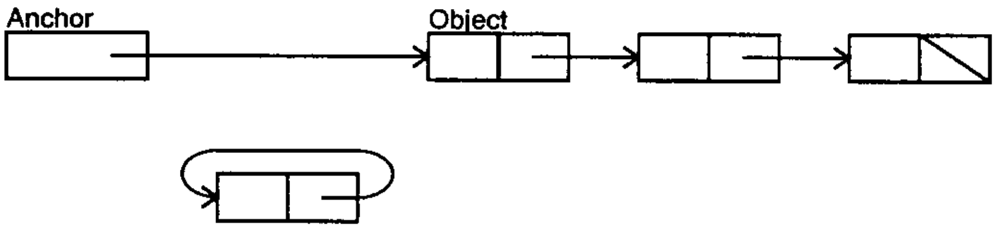

Figure 4-3: Fetch-and-cons: before executing *swap*

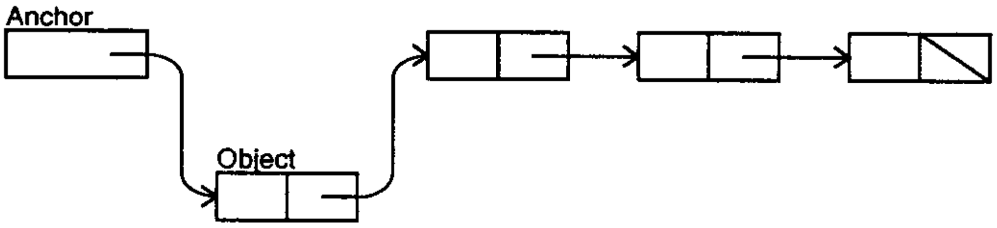

Figure 4-4: Fetch-and-cons: after executing *swap*

4.2. Reduction to Consensus

Direct constructions of *fetch-and-cons* tend to be complex. This section, however, shows that any object that solves consensus implements *fetch-and-cons*, and hence is universal. Our construction is too inefficient to apply directly in practice, since it uses unbounded storage, but it does provide a basic insight into the algorithmic complexity of wait-free synchronization. Our *fetch-and-cons* implementation requires at most *n* rounds of consensus, implying that any consensus protocol that is polynomial in *n* can be systematically transformed into a wait-free *fetch-and-cons* polynomial in *n*.

We now give a wait-free implementation of *fetch-and-cons* that calls a consensus protocol as a subroutine. First, some notation: let h and g be lists (sequences) of operations and p an operation. The notation "$p \in h$" means that p appears in h. The empty list is written "Λ", and the list constructed by prepending p to h is written "$p \bullet h$". The *merge* operator, written "\", takes two lists, a suffix and a prefix, and returns the list constructed by prepending to the suffix all the entries in the prefix but not in suffix, preserving their relative order in the prefix:

$\Lambda \setminus h = h$

$(p \bullet g) \setminus h =$ if $p \in h$ then $g \setminus h$ else $p \bullet (g \setminus h)$

A *consensus object* is any object that solves *n*-process consensus. To keep our notation consistent, we model multiple rounds of consensus as an unbounded array *consensus*. A process joins a consensus protocol by calling:

```
decide = operation(k: integer) returns (integer)
```

which takes the caller's input value and returns the decision value. In our construction, each process always inputs its own identifier; the process whose identifier is chosen in round *i* is the *winner* of that round. Each process has the following set of read/write registers.

- *Announce* is the latest operation executed by P_i, initially $\perp$.
- *round* is the latest round of consensus executed by P_i, initially 0.
- and *prefer* is P_i's decision list from its most recent consensus, initially Λ.

Each process also has a local variable *winner* that keeps track of the winner of the last round of consensus in which it participated.

```
fetch-and-cons = proc(x: item) returns (list)
  announce[i] := x
  goal := Λ
  lastRound := 0
  for each process P do
    if announce[P] ≠ ⊥ then goal := announce[P] • goal end
    lastRound := max(lastRound, round[P])
    end
  if lastRound > round[i] then
    winner := consensus[lastRound].decide(i)
    end
  for round in lastRound+1..lastRound+n do
    prefer[i] := goal \ pref[winner]
    winner := consensus[round].decide(i)
    prefer[i] := prefer[winner]
    rounds[i] := round
    if winner = i then return(trim(prefer[winner], i)) end
    end
  return(trim(prefer[winner], i))
  end fetch-and-cons
```

Figure 4-5: Implementing Fetch-And-Cons Using Rounds of Consensus

The pseudo-code for process *i* is shown Figure 4-5. In the first part of the protocol, the process announces its new operation by writing to announce[i], and then creates a *goal* history consisting of all processes' recently announced operations. The process also checks whether it failed to participate in the most recently observed round of consensus. If so, it "catches up" by ascertaining that round's winner. In the second part of the protocol, the process undertakes a sequence of at most *n* consensus protocols, starting just above the highest observed round number. For each round, it merges its goal with the previous round's winner. If it wins, it returns immediately; otherwise, it returns after *n* unsuccessful rounds. Upon returning, it calls *trim* to return the suffix following its own most recent operation.

Define the *view* for a *fetch-and-cons* to be the list constructed by prepending the operation's argument to its result. A sequential list history consisting of *fetch-and-cons* operations is legal if and only if each operation's view is a suffix if its predecessor's view. A concurrent history is linearizable if (1) all operations' views are coherent: given any two views, one is a suffix of the other, and (2) if p and q are *fetch-and-cons* operations such that p completes before q starts, then p's view is a suffix of q's.

We use the following auxiliary variables: *lastRound*$_p$ is process P's value of the local variable lastRound, and *maxRound* is the current maximum value of round[P], for all P.

Lemma 23: If maxRound > 0, then consensus round *j* has a winner for $0 < j \le$ maxRound.

Proof: By induction on the number of times any process P has advanced round[P]. The result holds trivially in the initial state when maxRound is zero. Assume the result holds for the current value of maxRound. P advances round[P] to $j =$ lastRound$_P + 1$ when it completes consensus round *j*, ensuring that round *j* has a winner. Since lastRound$_P \le$ maxRound, however, advancing round[P] advances maxRound by at most 1.

Lemma 24: All views generated by the implementation shown in Figure 4-5 are coherent.

Proof: By construction, the winner's preference for consensus round *j* is a suffix of all preferences for round *j+1*, and hence it is a suffix of the winner's preference. The sequence of winner's preferences are thus coherent. To show that the operations' views are also coherent, we show that each operation's view is a suffix of some winner's preference.

Suppose process P invokes *fetch-and-cons* with item x. The result is immediate if P returns after winning a round of consensus, so assume P loses all *n* rounds of consensus. At each step in P's *fetch-and-cons,* let *unseen(P)* the set of processes Q such that P has not yet read round[Q], and *seen(P)* the complement of seen(P). Let *Pmax* be:

$$\text{Pmax} = \max(\text{lastRound}_P, \max_{Q \in \text{unseen}(P)}(\text{round}[Q]))$$

While P is scanning the array, Pmax is a lower bound on the eventual value of lastRound$_P$.

We claim the following is invariant:

$$\text{For all Q, round[Q]} \ge \text{Pmax} + n \Rightarrow x \in \text{Prefer[Q]}.$$

In other words, if consensus round Pmax + n has a winner, then x is an element of that winner's preference. When P invokes *fetch-and-cons*, the invariant holds trivially because Pmax $\ge$ round[Q]. If some Q in seen(P) advances round[Q] to Pmax + n, then maxRound $\ge$ Pmax + n, and Lemma 23 implies that consensus rounds Pmax + 1, ..., Pmax + n have winners. Since we assume that P didn't win any of these rounds, some other process Q must have won twice. If Q won twice, it must have executed two *fetch-and-cons* operations. During the second operation, it must have read x from announce[P], and therefore x must be an element of prefer[Q], and an element of the winner's preference for all subsequent rounds.

Lemma 25: If a *fetch-and-cons* by process P precedes a *fetch-and-cons* by Q, then P's view is a suffix of Q's.

Proof: By the previous lemma, one operation's view must be a suffix of the other's. Let P's item be x and Q's item be y. Clearly, y cannot be an element of P's view, hence P's view must be a suffix of Q's.

These lemmas imply that every history of *fetch-and-cons* operations permitted by our implementation is linearizable, and hence:

Theorem 26: In a system of *n* processes, an object X is universal if (and only if) it solves *n*-process consensus.

Corollary 27: Any polynomial-time consensus algorithm can be transformed into a polynomial-time *fetch-and-cons.*

Corollary 28: Each object at level *n* in Figure 1-1 is universal in a system of *n* processes.

5. Conclusions

We believe that 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 additional issues must be addressed if wait-free synchronization is to become useful in practice. For example, although we have characterized some circumstances under which wait-free synchronization is possible, little is known about practical techniques for achieving it. Also, the extent to which universal wait-free primitives can be implemented in hardware remains unclear. For example, it is known that *fetch-and-add* has a wait-free implementation in terms of combining networks [12]. Although we have shown that *fetch-and-add* itself is not universal, it is natural to ask whether more powerful primitives such as memory-to-memory *swap* have similar implementations. Finally, the use of randomization [1] for wait-free concurrent objects remains unexplored.

References

[1] K. Abrahamson.
On achieving consensus using a shared memory.
In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC).* August, 1988.
To appear.

[2] J.H. Anderson and M.G. Gouda.
The Virtue of Patience: Concurrent Programming With and Without Waiting.
Private Communication.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] W.D. Hillis.
The Connection Machine.
The MIT Press, Cambridge, MA, 1985.

[12] 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.

[13] L. Lamport.
Concurrent Reading and Writing.
Communications of the ACM 20(11):806-811, November, 1977.

[14] L. Lamport.
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.
IEEE Transactions on Computers C-28(9):690, September, 1979.

[15] L. Lamport.
Specifying Concurrent Program Modules.
ACM Transactions on Programming Languages and Systems 5(2):190-222, April, 1983.

[16] L. Lamport.
On Interprocess Communication, Parts I and II.
Distributed Computing 1:77-101, 1986.

[17] 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.

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

[19] N.A. Lynch and M. Merritt.
Introduction to the Theory of Nested Transactions.
Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1986.

[20] N.A. Lynch and M.R. Tuttle.
Hierarchical Correctness Proofs for Distributed Algorithms.
Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1987.

[21] 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.

[22] C.H. Papadimitriou.
The Serializability of Concurrent Database Updates.
Journal of the ACM 26(4):631-653, October, 1979.

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

[24] 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.

[25] G.H. Pfister et al.
The IBM research parallel processor prototype (RP3): introduction and architecture.
In *International Conference on Parallel Processing.* 1985.

[26] C.L. Seitz.
The Cosmic Cube.
Communications of the ACM 28(1), January, 1985.

[27] 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".

[28] H.S. Stone.
Database applications of the FETCH-AND-ADD instruction.
IEEE Transactions on Computers C-33(7):604-612, July, 1984.

[29] 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.