Chapter 5

The Relative Power of Synchronization Methods

So far, we have been addressing questions of the form:

> Given objects X and Y, is there a wait-free implementation of X from one or more instances of Y?

It is clear how to show that such a wait-free implementation exists: just show the implementation, and then argue why it works. This strategy has worked out well, yielding a number of non-trivial wait-free constructions:

1. a regular register from safe registers,
2. an atomic register from regular registers
3. a multi-writer atomic register from single-reader atomic registers,
4. a single-enqueuer, single-dequeuer FIFO queue from atomic registers, and
5. a snapshot object from single-writer atomic registers

These observations invite even more questions. For example, can we construct a wait-free FIFO queue, using only atomic registers, such that the queue supports multiple dequeuers?

So far, we have no logical tools for showing claims of the form "there is no wait-free implementation of X by Y." In this chapter, we introduce a simple, yet powerful technique for showing that such implementations do *not* exist. We will show that there is an infinite hierarchy of objects such that no object at one level can implement any object at higher levels. The basic idea is simple: each class has an associated *consensus number*, which is the maximum number of threads for which objects of the class can solve an elementary synchronization

[0] *This chapter is part of the Manuscript* MULTIPROCESSOR SYNCHRONIZATION *by Maurice Herlihy and Nir Shavit copyright © 2003, all rights reserved.*

```
public abstract class Consensus {

  private Object[] proposed = new Object[n];

  // announce my input value to the world
  public void propose(Object value) {
    proposed[Thread.myIndex()] = value;
  }

  // figure out which thread was first
  abstract public Object decide();
}
```

Figure 5.1: Generic Consensus Protocol

problem, called *consensus*. We will see that in a system of n or more concurrent threads, is impossible to construct a wait-free implementation of an object with consensus number n from an object with a lower consensus number.

5.1 Consensus Numbers

Consensus is an innocuous-looking, somewhat abstract problem that will have enormous consequences for everything from algorithm design to hardware architecture. A *Consensus Object* provides two methods. Each thread first *proposes* a value (by calling the `propose` method), and then *decides* a value (through the `decide` method). A Consensus implementation must be:

- *consistent*: all threads decide the same value,
- *valid*: the common decision value is some thread's input, and
- *wait-free*: each thread decides after a finite number of steps.

It is convenient to establish a stylized form for consensus protocols, illustrated in Figure 5.1. Each thread first calls the `propose` method, which "announces" the thread's input value by writing it to a shared `proposed` array. Each thread then calls the `decide` method, which (in essence) figures out which thread "went first", and each thread then decides the value proposed by that thread.

We want to understand whether a particular class of objects is powerful enough to solve consensus. How can we make this notion more precise? If we think of such objects as supported by a lower level of the system, perhaps the operating system, or even the hardware, then we care about the properties of the class, not about the number of objects. (If the system can provide one object of this class, it can undoubtedly provide more.) Second, it is reasonable to suppose fill this in

Figure 5.2: Execution Tree for Two Threads

fill this in

Figure 5.3: Bivalent, Univalent, and 0-Valent States

that any modern system can provide a generous amount of read/write memory for bookkeeping. These two observations suggest the following definition.

Definition A class C *solves* n-thread consensus if there exist a consensus protocol which uses one or more objects of class C and any number of atomic registers.

Definition The *consensus number* of a class C is the largest n for which that class solves n-thread consensus. If no largest n exists, we say the class's consensus number is *infinite.*

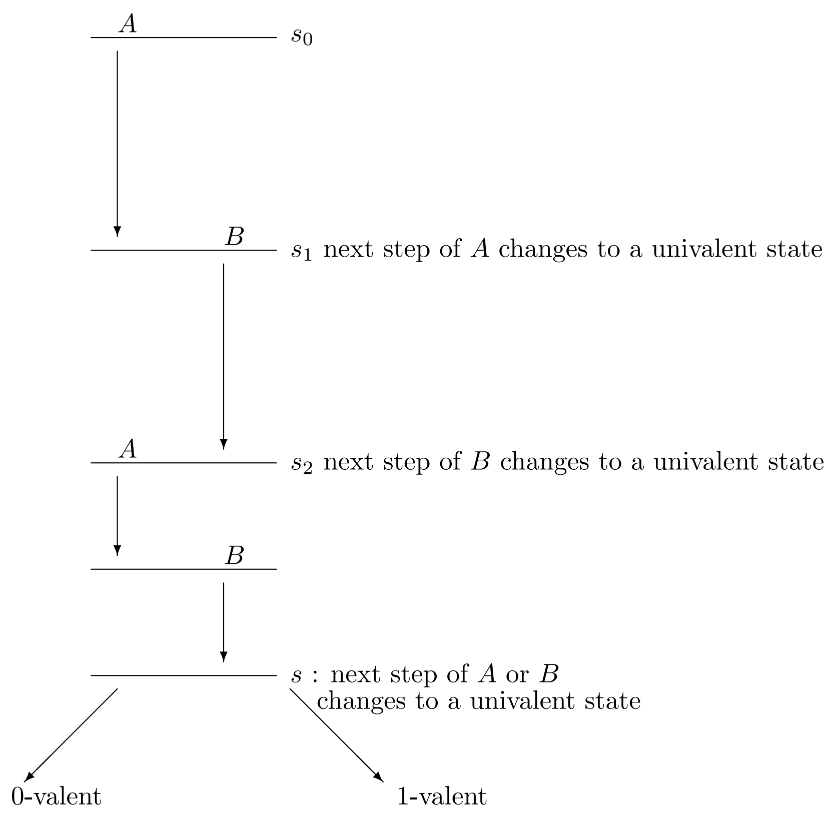
Corollary 5.1.1 *Suppose one can implement an object of class C from one or more objects of class D, together with some number of atomic registers. If class C solves n-consensus, then so does class D.*

We will now reason about consensus protocols in general. Later on, we will consider consensus protocol based on specific types of objects.

Imagine we have a consensus protocol for n threads, indexed 0 through $n-1$, with possible input values 0 and 1. While the protocol is running, the *protocol state* consists of the threads' states and the registers' states. We say that a thread *moves* when it calls a shared register's `read` or `write` method. An *initial protocol state* is a protocol state before any thread has moved. Because the protocol is wait-free, each thread moves only a finite number of times before it decides on a value and leaves the protocol.

We say that a protocol state is *bivalent* if the decision value is not yet fixed: there is some execution starting from that state in which the threads decide 0, and one in which they decide 1. By contrast, the protocol state is *univalent* if the outcome is fixed: the threads decide the same value in every execution. A protocol state is 1-*valent* if it is univalent, and the decision value will be 1, and similarly for 0-*valent*

It is convenient to visualize the set of possible protocol states as a tree. Each node of the tree represents a possible protocol state, and each edge of the tree is labeled with a thread identifier. Leaf nodes are labeled with decision values. An edge labeled A from node s to node s' means that if A moves in protocol state s, then the new protocol state is s'. We refer to s' as a *successor state* to s. Because the protocol is wait-free, any such tree is finite. We can label each leaf with the protocol's decision value. The execution tree for two threads that



A
s_0
B
s_1 next step of A changes to a univalent state
A
s_2 next step of B changes to a univalent state
B
s : next step of A or B changes to a univalent state
0-valent
1-valent

Figure 5.4: Reaching a critical state s

each move twice is shown in Figure 5.2. A bivalent protocol state corresponds to the root of a subtree whose leaves are labeled with both decision values, wile a univalent node's tree is labeled with only one value.

Our next lemma says that bivalent states exist. This observation means that the outcome of the protocol cannot be fixed in advance, but must depend on how `read` and `write` calls are interleaved.

Lemma 5.1.2 *Every two-thread consensus protocol has a bivalent initial state.*

Proof: Consider the initial state where A has input 0 and B has input 1. If A finishes the protocol before B takes a step, then A must decide 0, because it must decide some thread's input, and 0 is the only input it has seen. Symmetrically, If B finishes the protocol before A takes a step, then B must decide 1, because it must decide some thread's input, and 1 is the only input it has seen. If follows that the initial state where A has input 0 and B has input 1 is bivalent. ■

Lemma 5.1.3 *Every n-thread consensus protocol has a bivalent initial state.*

Proof: Left as an exercise. ■

A protocol state is *critical* if

- It is bivalent, and
- if any thread moves, the protocol state becomes univalent.

We leave as an exercise to the reader to show that one successor state must be 0-valent, and the other 1-valent.

Lemma 5.1.4 *Every consensus protocol has a critical state.*

Proof: Suppose not. By Lemma 5.1.3, the protocol has a bivalent initial state. Start the protocol in this state. As long as there is some thread that can move without making the protocol state univalent, let that thread move. If the protocol runs forever, then it is not wait-free. Otherwise, the protocol eventually enters a state where no such move is possible, which must be a critical state. ■

Everything we have proved so far applies to any consensus protocol, no matter what class (or classes) of shared objects it uses. Now we turn our attention to specific classes of objects.

5.2 Read/Write Registers

The obvious place to begin is to ask whether we can solve consensus using atomic registers. Surprisingly, perhaps, the answer is no.

Theorem 5.2.1 *Read/Write registers have consensus number 1.*

Proof: Suppose there exists such a protocol for two threads A and B. As usual, we will reason about the properties of such a protocol and derive a contradiction.

By Lemma 5.1.4, it is possible to run the protocol until it reaches a critical state. Suppose A's next move carries the protocol to an x-valent state, and B's next move carries the protocol to a y-valent state, where (of course) x and y are distinct.

What methods are A and B about to call? Perhaps

- one thread, say A, is about to read a register, or
- A and B are about to write to different registers, or
- A and B are about to write to the same register

The rest is a case analysis.

1. Suppose A is about to read a shared register (Figure 5.5).

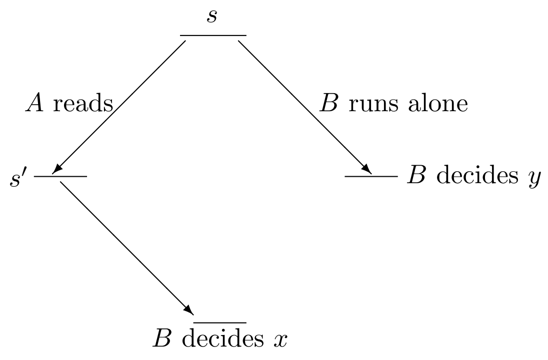


Figure 5.5: Case: A reads register r

After A reads, the new state s' is x-valent. If B runs by itself starting from s', it must also decide x. On the other hand, if B runs by itself starting from s, it reaches a y-valent state (by assumption) and decides y. Since the states s and s' look the same to B, B must decide the same value in both cases, a contradiction.

2. Suppose the threads write to different registers(Figure 5.6).

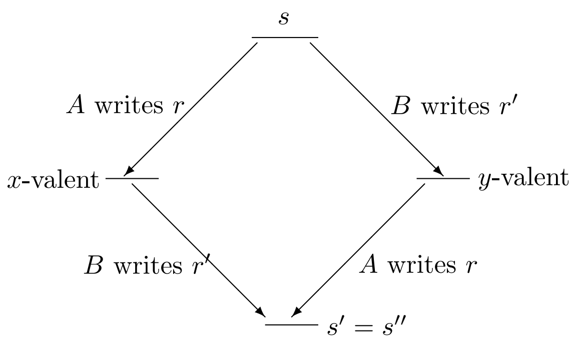


Figure 5.6: A and B about to write to distinct registers r and r'

The protocol state that results if A's write is immediately followed by B's write is the same as the protocol state that results if the writes occur in the opposite order. One order leads to an x-valent state and the other to y-valent state, a contradiction.

3. Suppose both threads write to the same register (Figure 5.7).

   Suppose A writes first. The resulting protocol state s' is an x-valent state, so if B runs uninterrupted from s' it must decide x. Suppose instead that B writes first. The resulting protocol state s'' is a y-valent state, so if B runs uninterrupted from s'' it must decide y. The problem is that s' and s look the same to B, so B must decide the same value starting from s' or s'', a contradiction.

■

Corollary 5.2.2 *It is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic read/write registers.*

5.3 Consensus Protocols

The impossibility of solving two-thread consensus using read/write registers has far-reaching implications, affecting areas ranging from high-level algorithm design to low-level hardware architectures. We will investigate these implications one-by-one, but our task will be easier if we settle on a standard interface for consensus protocols.

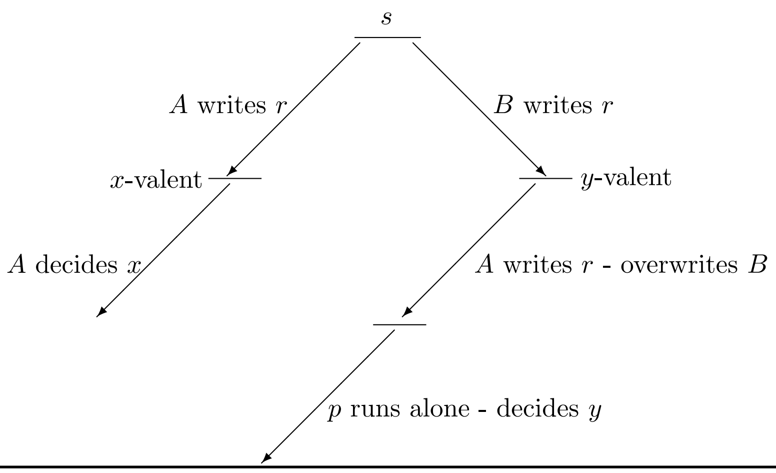


Figure 5.7: A and B both about to write to r

Recall that we assume that each of n threads has a unique index i, $0 \leq i < n$. Most (but not all) consensus protocols we consider can be split into two phases.

1. Thread i *proposes* its own input value by writing it into a shared `proposed` array at index i.

2. The threads decide among themselves which one went first. If they elect thread j, then they all decide the value stored in `proposed[j]`.

The first phase is the same for every consensus protocol, while the second will be different each time.

5.4 FIFO Queues

In the previous chapter, we saw a wait-free FIFO Queue implementation using only atomic registers, subject to the limitation that only one thread could dequeue from the queue, and only one thread could dequeue from the queue. It is natural to ask whether one can implement a wait-free FIFO queue that supports multiple enqueuers and dequeuers. For now, let us focus on a more specific problem: can we implement a wait-free two-dequeuer FIFO queue using atomic registers?

Theorem 5.4.1 *The two-dequeuer FIFO queue class has consensus number at least 2.*

```
public  class QueueConsensus extends Consensus {

  Queue queue;

  // initialize queue with two elements
  //  first:  "win"
  //  second: "lose"
  public QueueConsensus() {
    queue = new Queue();
    queue.enq("win");
    queue.enq("lose");
  }

  // figure out which thread was first
  public Object decide() {
    String status = queue.deq();
    int i = Thread.myIndex();
    if (status == "win")
      return proposed[i];
    else
      return proposed[1-i];
  }
}
```

Figure 5.8: Two-Thread Consensus using a FIFO Queue

Proof: Figure 5.8 shows a two-thread consensus protocol using a single FIFO queue. Here, the queue stores strings. The queue is initialized by enqueuing the value "win" followed by the value "lose". As in all the consensus protocol considered here, each thread first calls the superclass `propose` method, which stores the input value in a shared array. It then calls the `decide` method, which dequeues the next item from the queue. If that item is the string "win", then the calling thread was first, and it decides its own value. If that item is the string "lose", then the other thread was first, so the calling thread returns the other thread's input, as declared in the `proposed` array.

Each thread dequeues a string from the queue, returning its own preference if it dequeues "win", and the other's preference if it dequeues "lose".

The protocol is wait-free, since it contains no loops. If each thread returns its own input, then they must both have dequeued "win", violating the FIFO queue specification. If each returns the others' input, then they must both have dequeued "lose", also violating the queue specification.

The validity condition follows from the observation that the thread that dequeues "win" stores its input in the `proposed` array before the first queue method call. ■

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

Corollary 5.4.2 *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-thread consensus, they cannot solve three-thread consensus.

Theorem 5.4.3 *FIFO queues have consensus number 2.*

Proof: By contradiction. Assume we have a consensus protocol for threads A, B, and C. By Lemma 5.1.4, the protocol has a critical state. Without loss of generality, we can assume that A's next move takes the protocol to an x-valent state, and B's next move takes the protocol to a y-valent state. The rest is a case analysis.

First, suppose A and A both call `deq` methods. Let s be the protocol state if A dequeues and then B dequeues, and let s' be the state if the dequeues occur in the opposite order. Since s is x-valent, then if C runs uninterrupted from s, then it decides x. Since s' is y-valent, then if C runs uninterrupted from s, then it decides y. We have a problem: s and s' look the same to C, so C must decide the same value in both cases, a contradiction.

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

Finally, suppose both A and B do `enq` methods. Let s be the state at the end of the following execution:

1. A and B enqueue items a and b in that order.
2. Run A until it dequeues a. (Since the only way to observe the queue's state is via the `deq` method, A cannot decide before it observes one of a or b.)
3. Run B until it dequeues b.

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

1. B and A enqueue items b and a in that order.
2. Run A until it dequeues b.
3. Run B until it dequeues a.

Clearly, s is x-valent and s' is y-valent. Both of A's executions are identical until it dequeues a or b. Since A is halted before it can modify any other objects, B's executions are also identical until it dequeues a or b. By a now-familiar argument, a contradiction arises because s and s' are indistinguishable to C. ■

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.

5.5 Multiple Assignment

The *multiple assignment* problem

Theorem 5.5.1 *There is no wait free implementation of multiple assignment by atomic registers.*

Proof: It is enough to show that given two threads, an array of three registers, and a two-assignment method, then we can solve two-thread consensus.

As usual, the `decide` method must figure out which thread went first. Suppose thread 0 writes (atomically) to array elements 0 and 1, while thread 1 writes (atomically) to elements 1 and 2.

The threads perform their respective assignments, and then try to determine who went first. ¿From A's point of view, there are three cases:

- If A's assignment was ordered first, and B has not not happened, then positions 0 and 1 have A's value, and position 2 is null. A decides its own input.
- If A's assignment was ordered first, and B's second, then position 0 has A's value, and positions 1 and 2 have B's. A decides its own input.

  If B's assignment was ordered first, and A's second, then positions 0 and 1 have A's value, and 2 has B's. A decides B's input.

A similar analysis holds for B. ■

```
class MultiConsensus {

  MultiAssign multi;

  public Object decide() {
    int i = Thread.myIndex();   // my index
    int j = 1 - i;              // other index
    // double assignment
    multi.assign(i, input, i+1, input);
    // snapshot results
    Object[] result  = multi.scan();
    if (result[(j + 1) % 3] == null ||
        (result[j] == result[(j + 1) % 3]))
      return proposed[i];
    else
      return proposed[j];
  }
```

Figure 5.9: Two-Thread Consensus from Double Assignment

5.6 Read Modify Write (RMW) Methods

Many, if not all, of the classical synchronization primitives can be expressed as *read-modify-write* methods, defined as follows. Let r be a register, and f a function. The method `r.rmw`(r, f) is defined by the synchronized method shown in Figure 5.10.

```
class RMWRegister extends Register {

  synchronized int RMW(function f) {
    int prior = this.value;
    this.value = f(this.value);
    return prior;
  }
}
```

Figure 5.10: The *rmw* method

Note that if f is the identity function, then `r.rmw`(f, r) is simply a *read* method. Examples of well-known read-modify-write methods include `test-and-set` (Figure 5.11, `swap` (Figure 5.12 `fetch-and-increment` (Figure 5.13, and `fetch-and-ad` (Figure 5.14. Each of these methods has been implemented in hardware. For example, `swap` is called XCHG on Intel's x86 architecture.

```
int TAS() {
  int prior = this.value;
  this.value = 1;
  return prior;
}
```

Figure 5.11: Test-and-Set

```
int Swap(int x) {
  int prior = this.value;
  this.value = x;
  return prior;
}
```

Figure 5.12: Swap

```
int FetchAndInc() {
  int prior = this.value;
  this.value = this.value + 1;
  return prior;
}
```

Figure 5.13: Fetch-and-Increment

```
int FetchAndAdd(int x) {
  int prior = this.value;
  this.value = this.value + x;
  return prior;
}
```

Figure 5.14: Fetch-and-Add

```
class RMWConsensus extends Consensus {

  private RMWregister r = new RMWRegister(v); // initialize to v

  int decide() {
    int i = Thread.myIndex();   // my  index
    int j = 1 - i;      // other's index
    if (r.rmw(f) == v)      // I am first
      return proposed[i];
    else            // I am second
      return proposed[j];
  }
}
```

Figure 5.15: Two-Thread Consensus using read-modify-write

We say that a read-modify-write method is *non-trivial* if it is not a read.

Theorem 5.6.1 *A register with any non-trivial read-modify-write method has consensus number at least 2.*

Proof: We construct a two-thread protocol. Since f is not the identity, there exists a value v such that $f(v) \neq v$. The register is initialized to such a v.

As usual, the `propose` method writes the thread's input into a private `proposed` array, and the `decide` method simply determines which thread is ordered first. Each thread applies `rmw`(f) to a shared read-modify-write register. The first thread will get back the value v, and decides its own value, and the second gets back $f(v) \neq v$, and decides the other's value.

■

Corollary 5.6.2 *It is impossible to construct a wait free implementation of any nontrivial read-modify-write method from a set of atomic read/write registers in a system of two or more threads.*

5.7 Interfering Read-Modify Write Methods

Although read-modify-write registers are more powerful than read/write registers, many common read-modify-write methods are still computationally weak. In particular, there is no wait-free consensus protocol for three or more threads using any combination of `read`, `write`, `test-and-set`, `swap`, and `fetch-and-add` methods.

Definition Let F be a set of functions indexed by an arbitrary set S. F is *interfering* if for all values v and all i and j in S, either:

- f_i and f_j commute: $f_i(f_j(v)) = f_j(f_i(v))$

- One function "overwrites" the other: $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$.

For example, the `test-and-set` and `swap` methods use constant functions, which overwrite any other functions. The remaining `fetch-and-inc` and `fetch-and-add` methods use functions that commute with one another.

Very informally, here is the problem with three threads. The first thread (the "winner") can typically tell it was first, and the second and third threads (the "losers") can each tell that they are losers, but cannot identify which of the others was the winner. If the functions commute, then a loser thread cannot tell which of the others went first, and because the protocol is wait-free, it cannot wait to find out. If the functions overwrite, then a loser thread cannot tell whether it was preceded by a single winner, or by a winner followed by an overwriting loser. Let us make this argument more precise.

Theorem 5.7.1 *Any non-trivial read-modify-write method with functions taken from an interfering set has consensus number (exactly) 2.*

Proof: Theorem 5.6.1 states that any such method has consensus number at least 2. It suffices to show that one cannot use such an method to solve consensus among three threads.

Assume instead that such a protocol exists, where threads A, B, and C reach consensus through interfering read-modify-write registers. By Lemma 5.1.4, this protocol has a critical state, in which the protocol is bivalent, but any method call by any thread will cause the protocol to enter a univalent state.

We now do a case analysis, examining each possible method call. The kind of reasoning used in the proof of theorem 5.2.1 shows that the pending methods cannot be reads or writes to atomic read/write registers, nor can the threads be about to call methods of different objects.

It follows that the threads are about to call `rmw` methods of a single register r. Suppose A is about to call `rmw`(r, f_A), sending the protocol to an x-valent state, and B is about to call `rmw`(r, f_B), sending the protocol to a y-valent state, for $x \neq y$. There are two possible cases:

1. The functions commute: $f_A(f_B(v)) = f_B(f_A(v))$. Let s' be the state that results if A calls `r.rmw`(f_A) and then B calls `r.rmw`(f_B). Because s' is x-valent, C will decide x if runs alone until it finishes the protocol. Let s'' be the state that results if A and B perform their calls in the reverse order. Because s'' is y-valent, C will decide y if runs alone until it finishes the protocol. The problem is that the two possible register states $f_A(f_B(v))$ and $f_B(f_A(v))$ are the same, so s' and s'' differ only in the internal states of A and B. Since C completes the protocol without communicating with A or B, these two states look identical to C, so it cannot possibly decide different values from the two states.

2. One function overwrites the other: $f_B(f_A(v)) = f_B(v)$. Let s' be the state that results if A calls `r.rmw`(f_A) and then B executes `r.rmw`(f_B).

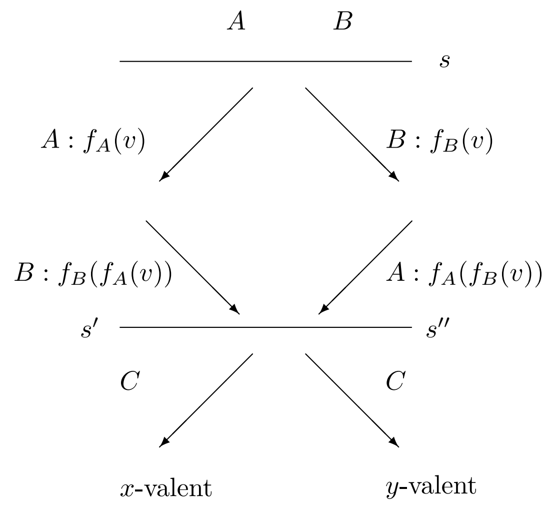


Figure 5.16: commuting functions

Because s' is x-valent, C will decide x if runs alone until it finishes the protocol. Let s'' be the state that results if B calls its method. Because s'' is y-valent, C will decide y if runs alone until it finishes the protocol. The problem is that the two possible register states $f_B(f_A(v))$ and $f_B(v))$ are the same, so s' and s'' differ only in the internal states of A and B. Since C completes the protocol without communicating with A or B, these two states look identical to C, so it cannot possibly decide different values from the two states. See Figure 5.17

■

5.8 Compare and Swap

We consider another read-modify-write method `compare-and-swap` (CAS), that is supported by several contemporary architectures. (For example, it is called `CMPXCHG` on the Intel Pentium(tm).) As illustrated in Figure 5.18, CAS takes two arguments: an *old* value and a *new* value. If the current register value is equal to the old value, then it is replaced by the new value, and otherwise the value is left unchanged. Either way, it returns the prior value. The reader should check that CAS is a read-modify-write method.

Theorem 5.8.1 *CAS has an unbounded consensus number.*

Proof: We display a consensus protocol for n threads $A_0, \ldots, A_n$ using CAS. As usual, the `propose` method writes the thread's input into a private `proposed`

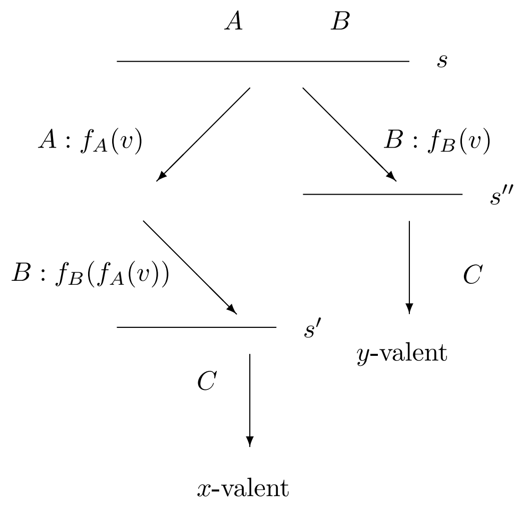


Figure 5.17: overwriting functions

```
synchronized int CAS(int old, int new) {
  int prior = this.value;
  if (this.value == old)
    this.value = new;
  return prior;
}
```

Figure 5.18: Compare and Swap (CAS)

array, and the `decide` method simply determines which thread is ordered first. The register value is initialized to -1, and each thread calls CAS with -1 as the old value, and its own index as the new value. Suppose thread i gets gets back prior value -1. It is linearized first, so it decides its own value. The other threads all get back prior value i, so they each decide `proposed[i]`.

■

5.9 Bibliography

- M. J. Fischer, N. A. Lynch, and M. S. Paterson. *Impossibility of distributed consensus with one faulty process.* Journal of the ACM, 32(2):374-382, April 1985.

Fischer, Lynch, and Paterson were the first to prove that consensus is impossible in a message-passing system where a single thread can halt. They introduced

```
class CASConsensus extends Consensus {

  private CASregister r = new CASRegister(-1); // initialize to -1

  int decide() {
    int i = Thread.myIndex();   // my  index
    int j = 1 - i;      // other's index
    if (r.CAS(-1,i) == -1)  // I am first
      return proposed[i];
    else            // I am second
      return proposed[j];
  }
}
```

the "critical-state" kind of impossibility argument.

- M. Herlihy, J. M. Wing. *Linearizability: A Correctness Condition for Concurrent Objects.* TOPLAS 12 (3): 463-492 (1990).

Herlihy and Wing formulated the notion of linearizability, and proved non-blocking and locality.

- C. Kruskal, L. Rudolph and M. Snir, *Efficient Synchronization of multi-processors with shared memory.* In Proceeding of 5'th PODC, Aug 1986.

Kruskal, Rudolph, and Snir developed the notion of a read-modify-write operation as a spinoff of the NYU Ultracomputer project

- M. Herlihy. *Wait-free Synchronization.* ACM Trans. on Programming Languages and Systems, Vol. 11, No.1, Jan 1991, pp. 124-149.

This paper introduced the notion of a consensus number as a measure of computational power, and first showed most of the impossibility and universality results presented in this chapter.