Chapter 18

The wait-free hierarchy

In a shared memory model, it may be possible to solve some problems using **wait-free** protocols, in which any process can finish the protocol in a bounded number of steps, no matter what the other processes are doing (see Chapter 26 for more on this and some variants).

The **wait-free hierarchy** h^r_m classifies asynchronous shared-memory object types T by **consensus number**, where a type T has consensus number n if with objects of type T and atomic registers (all initialized to appropriate values[1]) it is possible to solve wait-free consensus (i.e., agreement, validity, wait-free termination) for n processes but not for $n+1$ processes. The consensus number of any type is at least 1, since 1-process consensus requires no interaction, and may range up to ∞ for particularly powerful objects.

The general idea is that a type T with consensus number c can't simulate at type T' with a higher consensus number c', because then we could use the simulation to convert a c'-process consensus protocol using T' into a c'-process consensus protocol using T. The converse claim, that objects with the same or higher consensus numbers can simulate those with lower ones, is not necessarily true: even though n-process consensus can implement any object for n processes (see §18.2), it may be that for more than n

[1]The justification for assuming that the objects can be initialized to an arbitrary state is a little tricky. The idea is that if we are trying to implement consensus from objects of type T that are themselves implemented in terms of objects of type S, then it's natural to assume that we initialize our simulated type-T objects to whatever states are convenient. Conversely, if we are using the ability of type-T objects to solve n-process consensus to show that they can't be implemented from type-S objects (which can't solve n-process consensus), then for both the type-T and type-S objects we want these claims to hold no matter how they are initialized.

processes there is some object that has consensus number n but that cannot be implemented from an arbitrary n-consensus object.[2]

The wait-free hierarchy was suggested by work by Maurice Herlihy [Her91b] that classified many common (and some uncommon) shared-memory objects by consensus number, and showed that an unbounded collection of objects with consensus number n together with atomic registers gives a wait-free implementation of any object in an n-process system.

Various subsequent authors noticed that this did not give a **robust hierarchy** in the sense that combining two types of objects with consensus number n could solve wait-free consensus for larger n, and the hierarchy h_m^r was proposed by Prasad Jayanti [Jay97] as a way of classifying objects that might be robust: an object is at level n of the h_m^r hierarchy if having unboundedly many objects plus unboundedly many registers solves n-process wait-free consensus but not $(n+1)$-process wait-free consensus.[3]

Whether or not the resulting hierarchy is in fact robust for arbitrary deterministic objects is still open, but Ruppert [Rup00] subsequently showed that it is robust for RMW registers and objects with a read operation that returns the current state, and there is a paper by Borowsky, Gafni, and Afek [BGA94] that sketches a proof based on a topological characterization of computability[4] that h_m^r is robust for deterministic objects that don't discriminate between processes (unlike, say, single-writer registers). So for well-behaved shared-memory objects (deterministic, symmetrically accessible, with read operations, etc.), consensus number appears to give a real classification that allows us to say for example that any collection of read-write registers (consensus number 1), fetch-and-increments (2), test-and-set bits (2), and queues (2) is not enough to build a compare-and-swap (∞).[5]

A further complication is that all of these results assumed that objects are "combined" by putting them next to each other. If we can combine two objects by constructing a single object with operations of both—which is essentially what happens when we apply different machine language instructions to the same memory location—then the object with both operations may have a

[2]The existence of such objects was eventually demonstrated by Afek, Ellen, and Gafni [AEG16].

[3]The r in h_m^r stands for the registers, the m for having many objects of the given type. Jayanti [Jay97] also defines a hierarchy h_1^r where you only get finitely many objects. The h stands for "hierarchy," or, more specifically, $h(T)$ stands for the level of the hierarchy at which T appears [Jay11].

[4]See Chapter 28.

[5]Ruppert's paper is particularly handy because it gives an algorithm for computing the consensus number of the objects it considers. However, for infinite-state objects, this requires solving the halting problem (as previously shown by Jayanti and Toueg [JT92].

higher consensus number than the object with either operation individually.

This was observed by Ellen *et al.* [EGSZ16]. A simple example would be a register than supports increment (+1) and doubling (×2) operations. A register with only one of these operations is equivalent to a counter and has consensus number 1. But a register with both operations has consensus number at least 2, since if it is initialized to 2, we can tell which of the two operations went first by looking at the final value: $3 = 2 + 1, 4 = 2 \times 2, 5 = (2 \times 2) + 1, 6 = (2 + 1) \times 2$.

We won't attempt to do the robustness proofs of Borowsky *et al.* [BGA94] or Ruppert [Rup00]. Instead, we'll concentrate on Herlihy's original results and show that specific objects have specific consensus numbers when used in isolation. The procedure in each case will be to show an upper bound on the consensus number using a variant of Fischer-Lynch-Paterson (made easier because we are wait-free and don't have to worry about fairness) and then show a matching lower bound (for non-trivial upper bounds) by exhibiting an n-process consensus protocol for some n. Essentially everything below is taken from Herlihy's paper [Her91b], so reading that may make more sense than reading these notes.

18.1 Classification by consensus number

Here we show the position of various types in the wait-free hierarchy. The quick description is shown in Table 18.1; more details (mostly adapted from [Her91b]) are given below.

18.1.1 Level 1: atomic registers, counters, other interfering RMW registers that don't return the old value

First observe that any type has consensus number at least 1, since 1-process consensus is trivial.

We'll argue that a large class of particularly weak objects has consensus number exactly 1, by running FLP with 2 processes. Recall from Chapter 11 that in the Fischer-Lynch-Paterson [FLP85] proof we classify states as bivalent or univalent depending on whether both decision values are still possible, and that with at least one failure we can always start in a bivalent state (this doesn't depend on what objects we are using, since it depends only on having invisible inputs). Since the system is wait-free there is no constraint on adversary scheduling, and so if any bivalent state has a bivalent successor we can just do it. So to solve consensus we have to reach a bivalent configuration C that has only univalent successors, and in particular has a

Consensus number	Defining characteristic	Examples
1	Read with interfering no-return RMW.	Registers, counters, generalized counters, max registers, atomic snapshots.
2	Interfering RMW; queue-like structures.	Test-and-set, fetch-and-add, queues, process-to-memory swap.
m		m-process consensus objects.
$2m-2$		Atomic m-register write.
∞	First write-like operation wins.	Queue with peek, fetch-and-cons, sticky bits, compare-and-swap, memory-to-memory swap, memory-to-memory copy.

Table 18.1: Position of various types in the wait-free hierarchy

0-valent and a 1-valent successor produced by applying operations x and y of processes p_x and p_y.

Assuming objects don't interact with each other behind the scenes, x and y must be operations of the same object. Otherwise $Cxy = Cyx$ and we get a contradiction.

Now let's suppose we are looking at atomic registers, and consider cases:

- x and y are both reads, Then x and y commute: $Cxy = Cyx$, and we get a contradiction.
- x is a read and y is a write. Then p_y can't tell the difference between Cyx and Cxy, so running p_y to completion gives the same decision value from both Cyx and Cxy, another contradiction.
- x and y are both writes. Now p_y can't tell the difference between Cxy and Cy, so we get the same decision value for both, again contradicting that Cx is 0-valent and Cy is 1-valent.

There's a pattern to these cases that generalizes to other objects. Suppose that an object has a read operation that returns its state and one or more read-modify-write operations that don't return anything (perhaps we could call them "modify-write" operations). We'll say that the MW operations are **interfering** if, for any two operations x and y, either:

- x and y **commute**: $Cxy = Cyx$.
- One of x and y **overwrites** the other: $Cxy = Cy$ or $Cyx = Cx$.

Then no pair of read or modify-write operations can get us out of a bivalent state, because (a) reads commute; (b) for a read and MW, the non-reader can't tell which operation happened first; (c) and for any two MW operations, either they commute or the overwriter can't detect that the first operation happened. So any MW object with uninformative, interfering MW operations has consensus number 1.

For example, consider a counter that supports operations read, increment, decrement, and write: a write overwrites any other operation, and increments and decrements commute with each other, so the counter has consensus number 1. The same applies to a generalized counter that supports an atomic $x \leftarrow x + a$ operation; as long as this operation doesn't return the old value, it still commutes with other atomic increments.

Max registers [AACH12], which have read operations that return the largest value previously written, also have commutative updates, so they also have consensus number 1. This gives an example of an object not invented at the time of Herlihy's paper that are still covered by Herlihy's argument.

18.1.2 Level 2: interfering RMW objects that return the old value, queues (without peek)

Suppose now that we have a RMW object that returns the old value, and suppose that it is *non-trivial* in the sense that it has at least one RMW operation where the embedded function f that determines the new value is not the identity (otherwise RMW is just read). Then there is some value v such that $f(v) \neq v$. To solve two-process consensus, have each process p_i first write its preferred value to a register r_i, then execute the non-trivial RMW operation on the RMW object initialized to v. The first process to execute its operation sees v and decides its own value. The second process sees $f(v)$ and decides the first process's value (which it reads from the register). It follows that non-trivial RMW object has consensus number *at least* 2.

In many cases, this is all we get. Suppose that the operations of some RMW type T are non-interfering in a way analogous to the previous definition, where now we say that x and y commute if they leave the object in the same state (regardless of what values are returned) and that y overwrites x if the object is always in the same state after both x and xy (again regardless of what is returned). The two processes p_x and p_y that carry out x and y know what happened, but a third process p_z doesn't. So if we run p_z to completion we get the same decision value after both Cx and Cy, which means that Cx and Cy can't be 0-valent and 1-valent. It follows that no collection of RMW registers with interfering operations can solve 3-process consensus, and thus all such objects have consensus number 2. Examples of these objects include **test-and-set** bits, **fetch-and-add** registers, and **swap** registers that support an operation `swap` that writes a new value and returns the previous value.

There are some other objects with consensus number 2 that don't fit this pattern. Define a **wait-free queue** as an object with enqueue and dequeue operations (like normal queues), where dequeue returns $\perp$ if the queue is empty (instead of blocking). To solve 2-process consensus with a wait-free queue, initialize the queue with a single value (it doesn't matter what the value is). We can then treat the queue as a non-trivial RMW register where a process wins if it successfully dequeues the initial value and loses if it gets empty.[6]

However, enqueue operations are non-interfering: if p_x enqueues v_x and p_y enqueues v_y, then any third process can detect which happened first; similarly we can distinguish $\texttt{enq}(x)\texttt{deq}()$ from $\texttt{deq}()\texttt{enq}(x)$. So to show we can't do three process consensus we do something sneakier: given a bivalent state C with allegedly 0- and 1-valent successors $C\texttt{enq}(x)$ and $C\texttt{enq}(y)$, consider both $C\texttt{enq}(x)\texttt{enq}(y)$ and $C\texttt{enq}(y)\texttt{enq}(x)$ and run p_x until it does a $\texttt{deq}()$ (which it must, because otherwise it can't tell what to decide) and then stop it. Now run p_y until it also does a $\texttt{deq}()$ and then stop it. We've now destroyed the evidence of the split and poor hapless p_z is stuck. In the case of $C\texttt{deq}()\texttt{enq}(x)$ and $C\texttt{enq}(x)\texttt{deq}()$ on a non-empty queue we can kill the initial dequeuer immediately and then kill whoever dequeues x or the value it replaced, and if the queue is empty only the dequeuer knows. In either case we reach indistinguishable states after killing only 2 witnesses, and the queue has consensus number at most 2.

Similar arguments work on stacks, deques, and so forth—these all have consensus number exactly 2.

[6]But wait! What if the queue starts empty?

This turns out to be a surprisingly annoying problem, and was one of the motivating examples for h_m^r as opposed to Herlihy's vaguer initial definition.

With one empty queue and nothing else, Jayanti and Toueg [JT92, Theorem 7] show that there is no solution to consensus for two processes. This is also true for stacks (Theorem 8 from the same paper). But adding a register (Theorem 9) lets you do it. A second empty queue also works.

18.1.3 Level ∞: objects where the first write wins

These are objects that can solve consensus for any number of processes. Here are a bunch of level-∞ objects:

Queue with peek Has operations $\texttt{enq}(x)$ and $\texttt{peek}()$, which returns the first value enqueued. (Maybe also $\texttt{deq}()$, but we don't need it for consensus). Protocol is to enqueue my input and then peek and return the first value in the queue.

Fetch-and-cons Returns old cdr and adds new car on to the head of a list. Use preceding protocol where $\texttt{peek}() = \texttt{tail}(\mathsf{car} :: \mathsf{cdr})$.

Sticky bit Has a `write` operation that has no effect unless register is in the initial $\perp$ state. Whether the `write` succeeds or fails, it returns nothing. The consensus protocol is to write my input and then return result of a read.

Compare-and-swap Has $\texttt{CAS}(\mathsf{old}, \mathsf{new})$ operation that writes new only if previous value is old. Use it to build a sticky bit.

Load-linked/store-conditional Like compare-and-swap split into two operations. The **o**peration reads a memory location and marks it. The **o**peration succeeds only if the location has not been changed since the preceding load-linked by the same process. Can be used to build a sticky bit.

Memory-to-memory swap Has $\texttt{swap}(r_i, r_j)$ operation that atomically swaps contents of r_i with r_j, as well as the usual read and write operations for all registers. Use to implement fetch-and-cons. Alternatively, use two registers $\mathsf{input}[i]$ and $\mathsf{victory}[i]$ for each process i, where $\mathsf{victory}[i]$ is initialized to 0, and a single central register prize, initialized to 1. To execute consensus, write your input to $\mathsf{input}[i]$, then swap $\mathsf{victory}[i]$ with prize. The winning value is obtained by scanning all the $\mathsf{victory}$ registers for the one that contains a 1, then returning the corresponding input value.)

Memory-to-memory copy Has a $\texttt{copy}(r_i, r_j)$ operation that copies r_i to r_j atomically. Use the same trick as for memory-to-memory swap, where a process copies prize to $\mathsf{victory}[i]$. But now we have a process follow up by writing 0 to prize. As soon as this happens, the $\mathsf{victory}$ values are now fixed; take the leftmost 1 as the winner.[7]

[7] Or use any other rule that all processes apply consistently.

Herlihy [Her91b] gives a slightly more complicated version of this procedure, where there is a separate $\mathsf{prize}[i]$ register for each i, and after doing its copy a process writes 0 to all of the prize registers. This shows that memory-to-memory copy solves consensus for arbitrarily many processes even if we insist that copy operations can never overlap. The same trick also works for memory-to-memory swap, since we can treat a memory-to-memory swap as a memory-to-memory copy given that we don't care what value it puts in the $\mathsf{prize}[i]$ register.

Bank accounts A **bank account** object stores a non-negative integer, and supports a `read` operation that returns the current value and a `withdraw`(k) operation that reduces the value by k, unless this would reduce the value below 0, in which case it has no effect.

To solve (binary) consensus with a bank account, start it with 3, and have each process with input b attempt to withdraw $3 - b$ from the account. After the first withdrawal, the object will hold either 0 or 1, and no further withdrawals will have any effect. So the bank account acts exactly like a sticky bit where 3 represents $\perp$.[8]

18.1.4 Level $2m - 2$: simultaneous m-register write

Here we have a (large) collection of atomic registers augmented by an m-register write operation that performs all the writes simultaneously. The intuition for why this is helpful is that if p_1 writes r_1 and r_{shared} while p_2 writes r_2 and r_{shared} then any process can look at the state of r_1, r_2 and r_{shared} and tell which write happened first. Code for this procedure is given in Algorithm 18.1; note that up to 4 reads may be necessary to determine the winner because of timing issues.[9]

The workings of Algorithm 18.1 are straightforward:

- If the process reads $r_1 = r_2 = \perp$, then we don't care which went first, because the reader (or somebody else) already won.
- If the process reads $r_1 = 1$ and then $r_2 = \perp$, then p_1 went first.

[8]If you have more money, you can extend this construction to any fixed set of values. For example, to choose among values v in $0 \dots m - 1$, start with $2m$ and have a process with input v subtract $2m - v$.

[9]The main issue is that processes can only read the registers one at a time. An alternative to running Algorithm 18.1 is to use a double-collect snapshot (see §19.1) to simulate reading all three registers at once. However, this might require as many as twelve read operations, since a process doing a snapshot has to re-read all three registers if any of them change.

```
$v_1 \leftarrow r_1$
$v_2 \leftarrow r_2$
if $v_1 = v_2 = \perp$ then
    return no winner
if $v_1 = 1$ and $v_2 = \perp$ then
    // $p_1$ went first
    return 1
// read $r_1$ again
$v_1' \leftarrow r_1$
if $v_2 = 2$ and $v_1' = \perp$ then
    // $p_2$ went first
    return 2
// both $p_1$ and $p_2$ wrote
if $r_{\mathsf{shared}} = 1$ then
    return 2
else
    return 1
```

Algorithm 18.1: Determining the winner of a race between 2-register writes. The assumption is that p_1 and p_2 each wrote their own ids to r_i and r_{shared} simultaneously. This code can be executed by any process (including but not limited to p_1 or p_2) to determine which of these 2-register writes happened first.

- If the process reads $r_2 = 2$ and then $r_1 = \bot$, then p_2 went first. (This requires at least one more read after checking the first case.)

- Otherwise the process saw $r_1 = 1$ and $r_2 = 2$. Now read r_{shared}: if it's 1, p_2 went first; if it's 2, p_1 went first.

Algorithm 18.1 requires 2-register writes, and will give us a protocol for 2 processes (since the reader above has to participate somewhere to make the first case work). For m processes, we can do the same thing with m-register writes. We have a register $r_{pq} = r_{qp}$ for each pair of distinct processes p and q, plus a register r_{pp} for each p; this gives a total of $\binom{m}{2} + m = O(m^2)$ registers. All registers are initialized to $\bot$. Process p then writes its initial preference to some single-writer register pref_p and then simultaneously writes p to r_{pq} for all q (including r_{pp}). It then attempts to figure out the first writer by applying the above test for each q to r_{pq} (standing in for r_{shared}), r_{pp} (r_1) and r_{qq} (r_2). If it won against all the other processes, it decides its own value. If not, it repeats the test recursively for some p' that beat it until it finds a process that beat everybody, and returns its value. So m-register writes solve m-process wait-free consensus.

A further tweak gets $2m - 2$: run two copies of an $(m - 1)$-process protocol using separate arrays of registers to decide a winner for each group. Then add a second phase where processes contend across the groups. This involves each process p from group 1 writing the winning id for its group simultaneously into s_p and s_{pq} for each q in the other group. To figure out who won in the end, build a graph of all victories, where there is an edge from p to q if and only if p beat q in Phase 1 or p's id was written before q's id in Phase 2. The winner is the (unique) process with at least one outgoing edge and no incoming edges, which will be the process that won its own group (by writing first) and whose value was written first in Phase 2.

One thing to note about the second phase is that, unlike mutex, we can't just have the winners of the two groups fight each other, since this would not give the wait-free property for non-winners. Instead, we have to allow a non-winner p to pick up the slack for a slow winner and fight on behalf of the entire group. This requires an m-process write operation to write s_p and all s_{pq} at once.

18.1.4.1 Matching impossibility result

It might seem that the technique used to boost from m-process consensus to $(2m-2)$-process consensus could be repeated to get up to at least $\Theta(m^2)$, but this turns out not to be the case. The essential idea is to show that in order to escape bivalence, we have to get to a configuration C where *every* process is about to do an m-register write leading to a univalent configuration (since reads don't help for the usual reasons, and normal writes can be simulated by m-register writes with an extra $m-1$ dummy registers), and then argue that these writes can't overlap too much. So suppose we are in such a configuration, and suppose that Cx is 0-valent and Cy is 1-valent, and we also have many other operations $z_1 \dots z_k$ that lead to univalent states. Following Herlihy [Her91b], we argue in two steps:

1. There is some register that is written to by x alone out of all the pending operations. Proof: Suppose not. Then the 0-valent configuration $Cxyz_1 \dots z_k$ is indistinguishable from the 1-valent configuration $Cyz_1 \dots z_k$ by any process except p_x, and we're in trouble.
2. There is some register that is written to by x and y but not by any of the z_i. Proof:: Suppose not. The each register is written by at most one of x and y, making it useless for telling which went first; or it is overwritten by some z_i, hiding the value that tells which went first. So $Cxyz_1 \dots z_k$ is indistinguishable from $Cyxz_1 \dots z_k$ for any process other than p_x and p_y, and we're still in trouble.

Now suppose we have $2m-1$ processes. The first part says that each of the pending operations (x, y, all of the z_i) writes to 1 single-writer register and at least k two-writer registers where k is the number of processes leading to a different univalent value. This gives $k+1$ total registers simultaneously written by this operation. Now observe that with $2m-1$ process, there is some set of m processes whose operations all lead to a b-valent state; so for any process to get to a $(\neg b)$-valent state, it must write $m+1$ registers simultaneously. It follows that with only m simultaneous writes we can only do $(2m-2)$-consensus.

18.1.5 Level m: m-process consensus objects

An m-process **consensus object** has a single `consensus` operation that, the first m times it is called, returns the input value in the first operation, and thereafter returns only $\perp$. Clearly this solves m-process consensus. To show that it doesn't solve $(m+1)$-process consensus even when augmented with registers, run a bivalent initial configuration to a configuration C where any further operation yields a univalent state. By an argument similar to the m-register write case, we can show that the pending operations in C must all be consensus operations on the same consensus object (anything else commutes or overwrites). Now run $Cxyz_1 \dots z_k$ and $Cyxz_1 \dots z_k$, where x and y lead to 0-valent and 1-valent states, and observe that p_k can't distinguish the resulting configurations because all it got was $\bot$. (Note: this works even if the consensus object isn't in its initial state, since we know that before x or y the configuration is still bivalent.)

So the m-process consensus object has consensus number m. This shows that h^r_m is nonempty at each level.

A natural question at this point is whether the inability of m-process consensus objects to solve $(m+1)$-process consensus implies robustness of the hierarchy. One might consider the following argument: given any object at level m, we can simulate it with an m-process consensus object, and since we can't combine m-process consensus objects to boost the consensus number, we can't combine any objects they can simulate either. The problem here is that while m-process consensus objects can simulate any object in a system with m processes (see below), it may be that some objects can do more in a system with $m+1$ objects while still not solving $(m+1)$-process consensus. A simple way to see this would be to imagine a variant of the m-process consensus object that doesn't fail completely after m operations; for example, it might return one of the first two inputs given to it instead of $\bot$. This doesn't help with solving consensus, but it might (or might not) make it too powerful to implement using standard m-process consensus objects.

18.2 Universality of consensus

Universality of consensus says that any type that can implement n-process consensus can, together with atomic registers, give a wait-free implementation of any object in a system with n processes. That consensus is universal was shown by Herlihy [Her91b] and Plotkin [Plo89]. Both of these papers spend a lot of effort on making sure that both the cost of each operation and the amount of space used is bounded. But if we ignore these constraints, the same result can be shown using a mechanism similar to the replicated state machines of §12.3.

Here the processes repeatedly use consensus to decide between candidate histories of the simulated object, and a process successfully completes an operation when its operation (tagged to distinguish it from other similar operations) appears in a winning history. A round structure avoids too much confusion.

Details are given in Algorithm 18.2.

There are some subtleties to this algorithm. The first time that a process

```
1  procedure apply($\pi$)
       // announce my intended operation
2      op[$i$] $\leftarrow \pi$
3      while true do
           // find a recent round
4          $r \leftarrow \max_j$ round[$j$]
           // obtain the history as of that round
5          if $h_r = \bot$ then
6              $h_r \leftarrow$ consensus($c[r], \bot$)
7          if $\pi \in h_r$ then
8              return value $\pi$ returns in $h_r$
           // else attempt to advance
9          $h' \leftarrow h_r$
10         for each $j$ do
11             if op[$j$] $\notin h'$ then
12                 append op[$j$] to $h'$
13         $h_{r+1} \leftarrow$ consensus($c[r+1], h'$)
14         round[$i$] $\leftarrow r + 1$
```

Algorithm 18.2: A universal construction based on consensus

calls consensus (on $c[r]$), it may supply a dummy input; the idea is that it is only using the consensus object to obtain the agreed-upon history from a round it missed. It's safe to do this, because no process writes r to its round register until $c[r]$ is complete, so the dummy input can't be accidentally chosen as the correct value.

It's not hard to see that whatever h_{r+1} is chosen in $c[r+1]$ is an extension of h_r (it is constructed by appending operations to h_r), and that all processes agree on it (by the agreement property of the consensus object $c[r+1]$. So this gives us an increasing sequence of consistent histories. We also need to show that these histories are linearizable. The obvious linearization is just the most recent version of h_r. Suppose some call to apply(π_1) finishes before a call to apply(π_2) starts. Then π_1 is contained in some h_r when apply(π_1) finishes, and since π_2 can only enter h by being appended at the end, we get π_1 linearized before π_2.

Finally, we need to show termination. The algorithm is written with a loop, so in principle it could run forever. But we can argue that no process after executes the loop more than twice. The reason is that a process p puts its operation in op$[p]$ before it calculates r; so any process that writes $r' > r$ to round sees p's operation before the next round. It follows that p's value gets included in the history no later than round $r + 2$. (We'll see this sort of thing again when we do atomic snapshots in Chapter 19.)

Building a consistent shared history is easier with some particular objects that solve consensus. For example, a **fetch-and-cons** object that supplies an operation that pushes a new head onto a linked list and returns the old head trivially implements the common history above without the need for helping. One way to implement fetch-and-cons is with a swap object; to add a new element to the list, create a cell with its next pointer pointing to itself, then swap the next field with the head pointer for the entire list.

The solutions we've described here have a number of deficiencies that make them impractical in a real system (even more so than many of the algorithms we've described). If we store entire histories in a register, the register will need to be very, very wide. If we store entire histories as a linked list, it will take an unbounded amount of time to read the list. For solutions to these problems, see [AW04, 15.3] or the papers of Herlihy [Her91b] and Plotkin [Plo89].