Lecture 10

Mutual Exclusion and Store & Collect

In the previous lectures, we've learned a lot about message passing systems. We've also seen that neither in shared memory nor message passing systems consensus can be solved deterministically. But what makes them *different?* Obviously, the key difference to message passing is the shared memory: Different processors can access the same register to store some crucial information, and anyone interested just needs to access this register. In particular, we don't suffer from locality issues, as nodes are just one shared register away. Think for instance about pointer jumping, which is not possible in a message passing system, or about MST construction, where the diameter of components matters.

Alas, great power comes with its own problems. One of them is to avoid that newly posted information is overwritten by other nodes before it's noticed.

Definition 10.1 (Mutual Exclusion)**.** *We are given a number of nodes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>,
where <Remaining Code> means that the node can access the critical section multiple times. A mutual exclusion *algorithm consists of code for entry and exit sections, such that the following holds*[1]

Mutual Exclusion *At all times* at most one *node is in the critical section.*

No Deadlock *If some node manages to get to the entry section, later* some *(possibly different) node will get to the critical section (in a fair execution).*

Sometimes we in addition ask for

No Lockout *If some node manages to get to the entry section, later* the same *node will get to the critical section.*

Unobstructed Exit *No node can get stuck in the exit section.*

[1] Assuming that nodes finish the <Critical Section> in finite time.

Remarks:

- We're operating in the asynchronous model today, as is standard for shared memory. The reason is that the assumption of strong memory primitives and organization of modern computing systems (multiple threads, interrupts, accesses to the hard drive, etc.) tend to result in unpredictable response times that can vary dramatically.

10.1 Strong RMW Primitives

Various shared memory systems exist. A main difference is how they allow nodes to access the shared memory. All systems can atomically read or write a shared register R. Most systems do allow for advanced atomic *read-modify-write (RMW)* operations, for example:

test-and-set(R): $t := R$; $R := 1$; return t

fetch-and-add(R, x): $t := R$; $R := R + x$; return t

compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;

load-link(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**) only if the register's content hasn't been modified in the meantime. Otherwise, the store-conditional is guaranteed to fail (and return **false**), even if the value read by the load-link has since been restored.

An operation being atomic means that it is only a single step in the execution. For instance, no other node gets to execute the "fetch" part of the fetch-and-add primitive while another already completed it, but hasn't executed the addition yet.

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 20 shows an example with the test-and-set primitive.

Algorithm 20 Mutual exclusion using test-and-set, code at node v.

Given: some shared register R, initialized to 0.

```
<Entry>
 1: repeat
 2:     r := test-and-set(R)
 3: until r = 0
<Critical Section>
 4: ...
<Exit>
 5: R := 0
<Remainder Code>
 6: ...
```

Theorem 10.2. *Algorithm 20 solves mutual exclusion and guarantees unobstructed exit.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} node to execute the test-and-set "successfully," i.e., such that the result is 0. Denote by t_i the time when this happens and by t'_i the time when p_i resets the shared register R to 0. Between t_i and t'_i no other node can successfully test-and-set, hence no other node can enter the critical section during $[t_i, t'_i]$.

Proving no deadlock works similar: One of the nodes loitering in the entry section will successfully test-and-set as soon as the node in the critical section exited.

Since the exit section only consists of a single instruction (no potential infinite loops), we have unobstructed exit. □

Remarks:

- No lockout, on the other hand, is not ensured by this algorithm. Even with only two nodes there are asynchronous executions in which always the same node wins the test-and-set.
- Algorithm 20 can be adapted to guarantee this, essentially by ordering the nodes in the entry section in a queue.
- The power of RMW operations can be measured with the *consensus number*. The consensus number k of an RMW operation is defined as the number of nodes for which one can solve consensus with k (crashing) nodes using basic read and write registers alongside the respective RMW operations. For example, test-and-set has consensus number 2, whereas the consensus number of compare-and-swap is infinite.
- It can be shown that the power of a shared memory system is determined by the consensus number ("universality of consensus"). This insight has a remarkable theoretical and practical impact. In practice, for instance, after this was known, hardware designers stopped developing shared memory systems that support only weak RMW operations.

10.2 Mutual Exclusion using only RW Registers

Do we actually need advanced registers to solve mutual exclusion? Or to solve it efficiently? It's not as simple as before,[2] but can still be done in a fairly straightforward way.

We'll look at mutual exclusion exclusion for two nodes p_0 and p_1 only. We discuss how it can be extended to more nodes in the remarks. The general idea is that node p_i has to mark its desire to enter the critical section in a "want" register W_i by setting $W_i := 1$. Only if the other node is not interested ($W_{1-i} = 0$) access is granted. To avoid deadlocks, we add a priority variable Π enabling one node to enter the critical section even when the "want" registers are saying that none shall pass.

Theorem 10.3. *Algorithm 21 solves mutual exclusion and guarantees both no lockout and unobstructed exit.*

[2] Who would have guessed, we're talking about a non-trivial problem here.

Algorithm 21 Mutual exclusion: Peterson's algorithm.

Given: shared registers W_0, W_1, Π, all initialized to 0.
Code for node p_i, $i \in \{0, 1\}$:
<**Entry**>
1: $W_i := 1$
2: $\Pi := 1 - i$
3: **repeat** *nothing* **until** $\Pi = i$ or $W_{1-i} = 0$ // "busy-wait"
<**Critical Section**>
4: ...
<**Exit**>
5: $W_i := 0$
<**Remainder Code**>
6: ...

Proof. The shared variable Π makes sure that one of the nodes can enter the critical section. Suppose p_0 enters the critical section first. If at this point it holds that $W_1 = 0$, p_1 has not yet executed Line 1 and therefore will execute Line 2 before trying to enter the critical section, which means that Π will be 0 and p_1 has to wait until p_0 leaves the critical section and resets $W_0 := 0$. On the other hand, if $W_1 = 1$ when p_0 enters the critical section, we already must have that $\Pi = 0$ at this time, i.e., the same reasoning applies. Arguing analogously for p_1 entering the critical section first, we see that mutual exclusion is solved.

To see that there are no lockouts, observe that once, e.g., p_0 is executing the spin-lock (i.e., is "stuck" in Line 3), the priority variable is not going to be set to 1 again until it succeeds in entering and passing the critical section. If p_1 is also interested in entering and "wins" (we already know that one of them will), afterwards it either will stop trying to enter or again set Π to 0. In any event, p_0 enters the section next.

Since the exit section only consists of a single instruction (no potential infinite loops), we have unobstructed exit. □

Remarks:

- Line 3 in Algorithm 21 is a *spinlock* or *busy-wait,* like Lines 1-3 in Algorithm 20. Here we have the extreme case that the node doesn't even try to do anything, it simply needs to wait for someone else to finish the job.

- Extending Peterson's Algorithm to more than 2 nodes can be done by a tournament tree, like in tennis. With n nodes every node needs to win $\lceil \log n \rceil$ matches before it can enter the critical section. More precisely, each node starts at the bottom level of a binary tree, and proceeds to the parent level if winning. Once winning the root of the tree it can enter the critical section.

- This solution inherits the additional nice properties: no lockouts, unobstructed exit.

- On the downside, more work is done than with the test-and-set operation, as the binary tree has depth $\lceil \log n \rceil$. One captures this by counting

asynchronous rounds or the number of actual changes of variables,[3] as only signal *transitions* are "expensive" (i.e., costly in terms of energy) in circuits.

10.3 Store & Collect

Informally, the STORE & COLLECT problem can be stated as follows. There are n nodes $p_1, \ldots, p_n$. Every node p_i has a read/write register R_i in the shared memory, where it can *store* some information that is destined for the other nodes. Further, there is an operation by which a node can *collect* (i.e., read) the values of all the nodes that stored some value in their register.

We say that an operation *op1 precedes* an operation *op2* iff *op1* terminates before *op2* starts. An operation *op2 follows* an operation *op1* iff *op1* precedes *op2*.

Definition 10.4 (Store and Collect)**.** *There are two operations: A* STORE(val) *by node p_i sets val to be the latest value of its register R_i. A* COLLECT *operation returns a* view*, i.e., a function $f\colon V \to VAL \cup \{\perp\}$ from the set of nodes V to a set of values VAL or the symbol $\perp$, which means "nothing written yet." Here, $f(p_i)$ is intended to be the latest value stored by p_i, for each node p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every node p_i:*

- *If $f(p_i) = \perp$, then no* STORE *operation by p_i precedes* cop*.*
- *If $f(p_i) = val \neq \perp$, then val is the value of a* STORE *operation* sop *of p_i that does not follow* cop *satisfying that there is no* STORE *operation by p_i that follows* sop *and precedes* cop*.*

Put simply, a COLLECT operation *cop* should not read from the future or miss a preceding STORE operation *sop*.

Attention: A COLLECT operation is not atomic, i.e., consists of multiple (atomic) operations! This means that there can be reads that neither precede nor follow a COLLECT. Such overlapping operations are considered *concurrent*. In general, also a write operation can be more involved, to simplify reads or achieve other properties, so the same may apply to them.

We assume that the read/write register R_i of every node p_i is initialized to $\perp$. We define the *step complexity* of an operation *op* to be the number of accesses to registers in the shared memory. There is a trivial solution to the *collect* problem shown in Algorithm 22.

[3] There may be an unbounded number of read operations due to the busy-wait, and it is trivial to see that this cannot be avoided in a (completely) asynchronous system.

Algorithm 22 Trivial COLLECT.

Operation STORE(val) (by node p_i) **:**
1: $R_i := val$
Operation COLLECT**:**
2: **for** $i := 1$ **to** n **do**
3: $f(p_i) := R_i$ // *read register R_i*
4: **end for**

Remarks:

- Obviously,[4] Algorithm 22 works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.
- The step complexities of Algorithm 22 is optimal: There are cases in which a COLLECT operation needs to read all n registers. However, there are also scenarios in which the step complexity of the COLLECT operation is unnecessarily large. Assume that there are only two nodes p_i and p_j that have stored a value in their registers R_i and R_j. Then, in principle, COLLECT needs to read the registers R_i and R_j only.

10.3.1 Splitters

Assume that up to a certain time t, $k \leq n$ nodes have started at least one operation. We call an operation completing at time t *adaptive* to contention, if its step complexity depends on k only.

To obtain adaptive COLLECT algorithms, we will use a symmetry breaking primitive called a *splitter*.

Definition 10.5 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristics. A node entering a splitter exits with either **stop**, **left**, or **right**. If k nodes enter a splitter, at most one node exits with **stop** and at most $k-1$ nodes exit with **left** and **right**, respectively.*

This definition guarantees that if a single node enters the splitter, then it obtains **stop**, and if two or more nodes enter the splitter, then there is at most one node obtaining **stop** and there are two nodes that obtain different values

[4] Be extra careful whenever such a word pops up. If it's not indeed immediately obvious, it may translate to "I believe it works, but didn't have the patience to check the details," which is an excellent source of (occasionally serious) blunders. One of my lecturers once said: "If it's trivial, then why don't we write it down? It should not take more than a line. If it doesn't, then it's not trivial!"

Algorithm 23 Splitter Code

Shared Registers: $X\colon \{\perp\} \cup \{1,\ldots,n\}$; $Y\colon$ **boolean**
Initialization: $X := \perp$; $Y :=$ **false**

Splitter access by node p_i:

```
1:  X := i;
2:  if Y then
3:    return right
4:  else
5:    Y := true
6:    if X = i then
7:      return stop
8:    else
9:      return left
10:   end if
11: end if
```

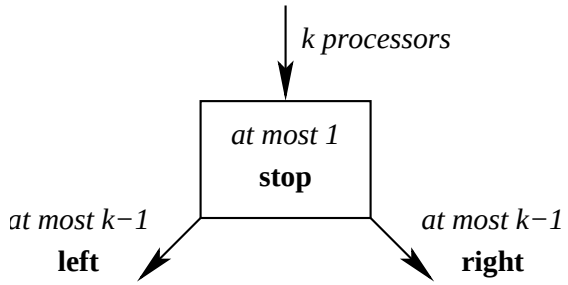

Figure 10.1: A Splitter

(i.e., either there is exactly one **stop** or there is at least one **left** and at least one **right**). For an illustration, see Figure 10.1. Algorithm 23 implements a splitter.

Lemma 10.6. *Algorithm 23 implements a splitter.*

Proof. Assume that k nodes enter the splitter. Because the first node that checks whether $Y =$ **true** in line 2 will find that $Y =$ **false**, not all nodes return **right**. Next, assume that i is the last node that sets $X := i$. If i does not return **right**, it will find $X = i$ in Line 6 and therefore return **stop**. Hence, there is always a node that does not return **left**.

It remains to show that at most 1 node returns **stop**. Suppose p_i decides to do this at time t, i.e., p_i reads that $X = i$ in Line 6 at time t. Then any p_j that sets $X := j$ after time t will (re)turn **right**, as already $Y =$ **true**. As any other node p_j will not read $X = j$ after time t (there is no other way to change X to j), this shows that at most one node will return **stop**. Finally, observe that if $k = 1$, then the result for the single entering node will be **stop**. □

10.3.2 Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S \colon \{\perp\} \cup \{1, \ldots, n\}$ that is initialized to $\perp$ and an additional shared variable $M_S \colon$ **boolean** that is initialized to **false**. We call a splitter S *marked* if $M_S =$ **true**. The $2^n - 1$ splitters are arranged in a complete binary tree of height $n - 1$. Let $S(v)$ be the splitter associated with a node v of the binary tree. The STORE and COLLECT operations are given by Algorithm 24.

Theorem 10.7. *Algorithm 24 implements* STORE *and* COLLECT*. Let k be the number of participating nodes. The step complexity of the first* STORE *of a node p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k)$.*

Proof. Because at most one node can stop at a splitter, it is sufficient to show that every node stops at some splitter at depth at most $k - 1 \leq n - 1$ when invoking the first STORE operation to prove correctness. We prove that at most $k - i$ nodes enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). This follows by induction from the definition of splitters, as not all nodes entering a splitter can proceed to the same subtree

Algorithm 24 Adaptive collect: binary tree algorithm

Operation STORE(val) (by node p_i) **:**

```
1:  R_i := val
2:  if first STORE operation by p_i then
3:    v := root node of binary tree
4:    α := result of entering splitter S(v);
5:    M_S(v) := true
6:    while α ≠ stop do
7:      if α = left then
8:        v := left child of v
9:      else
10:       v := right child of v
11:     end if
12:     α := result of entering splitter S(v);
13:     M_S(v) := true
14:   end while
15:   Z_S(v) := i
16: end if
```

Operation COLLECT**:**

Traverse marked part of binary tree:

```
17: for all marked splitters S do
18:   if Z_S ≠ ⊥ then
19:     i := Z_S; f(p_i) := R_i                    // read value of node p_i
20:   end if
21: end for                                        // f(p_i) = ⊥ for all other nodes
```

rooted at a child of the splitter. Hence, at the latest when reaching depth $k-1$, a node is the only node entering a splitter and thus obtains **stop**.

Note that the step complexity of executing a splitter is $\mathcal{O}(1)$. The bound of $k-1$ on the depth of the accessed subtree of the binary splitter tree thus shows that the step complexity of the initial STORE is $\mathcal{O}(k)$ for each node, and each subsequent STORE requires only $\mathcal{O}(1)$ steps.

To show that the step complexity of COLLECT is $\mathcal{O}(k)$, we first observe that the marked nodes of the binary tree are connected, and therefore can be traversed by only reading the variables M_S associated to them and their neighbors. Hence, showing that at most $2k-1$ nodes of the binary tree are marked is sufficient. Let x_k be the maximum number of marked nodes in a tree when $k \in \mathbb{N}_0$ nodes access the root. We claim that $x_k \leq \max\{2k-1, 0\}$, which is trivial for $k = 0$. Now assume the inequality holds for $0, \ldots, k-1$. Splitters guarantee that neither all nodes turn **left** nor all nodes turn **right**, i.e., $k_l \leq k-1$ nodes will turn left and $k_r \leq \min\{k-k_l, k-1\}$ turn right. The left and right children of the root are the roots of their subtrees, hence the induction hypothesis yields

$$x_k \leq x_{k_l} + x_{k_r} + 1 \leq \max\{2k_l - 1, 0\} + \max\{2k_r - 1, 0\} + 1 \leq 2k - 1,$$

concluding induction and proof. □

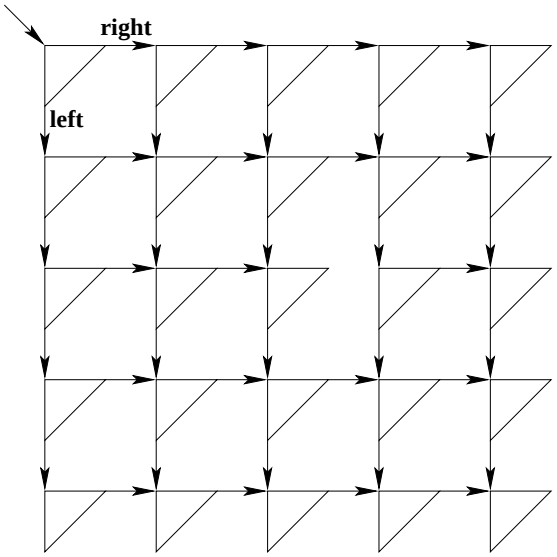

Figure 10.2: 5×5 Splitter Matrix

Remarks:

- The step complexities of Algorithm 24 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal.[5] In order for the algorithm to work, we however need to allocate the memory for the complete binary tree of depth $n-1$. The space complexity of Algorithm 24 therefore is exponential in n. We will next see how to obtain a polynomial space complexity at the cost of a worse COLLECT step complexity.

10.3.3 Splitter Matrix

In order to obtain quadratic memory consumption (instead of the exponential memory consumption of the splitter tree), we arrange n^2 splitters in an $n \times n$ matrix as shown in Figure 10.2. The algorithm is analogous to Algorithm 24. The matrix is entered at the top left. If a node receives **left**, it next visits the splitter in the next row of the same column. If a node receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $\mathcal{O}(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 10.8. *Let k be the number of participating nodes. The step complexity of the first* STORE *of a node p_i is $\mathcal{O}(k)$, the step complexity of every additional* STORE *of p_i is $\mathcal{O}(1)$, and the step complexity of* COLLECT *is $\mathcal{O}(k^2)$.*

Proof. Let the top row be row 0 and the left-most column be column 0. Let x_i be the number of nodes entering a splitter in row i. By induction on i, we show

[5] Here's another clearly to watch carefully. While the statement is correct, it's not obvious that we chose the performance measure wisely. We could refine our notion again and ask for the step complexity in terms of the number of writes that did not precede the most recent COLLECT operation of the collecting process. But let's not go there today.

that $x_i \leq k - i$. Clearly, $x_0 \leq k$. Let us therefore consider the case $i > 0$. Let j be the largest column such that at least one node visits the splitter in row $i-1$ and column j. By the properties of splitters, not all nodes entering the splitter in row $i-1$ and column j obtain **left**. Therefore, not all nodes entering a splitter in row $i-1$ move on to row i. Because at least one node stays in every row, we get that $x_i \leq k - i$. Similarly, the number of nodes entering column j is at most $k-j$. Hence, every node stops at the latest in row $k-1$ and column $k-1$ and the number of marked splitters is at most k^2. Thus, the step complexity of COLLECT is at most $\mathcal{O}(k^2)$. Because the longest path in the splitter matrix is $2k$, the step complexity of STORE is $\mathcal{O}(k)$. □

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of nodes entering the splitter in row i and column j is at most $k - i - j$. Hence, it suffices to only allocate the upper left half (including the diagonal) of the $n \times n$ matrix of splitters.

- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $\mathcal{O}(k)$ step complexity and $\mathcal{O}(n^2)$ space complexity.

What to take Home

- Obviously, more powerful RMW primitives are extremely useful. However, their implementation might be more costly than an implementation using read/write registers only. At the end of the day, RMW primitives solve mutual exclusion at some level of the system hierarchy.

- Naturally, atomic read/write registers do not fall out of the sky either. They are implemented from non-atomic registers using similar techniques.

Bibliographic Notes

Already in 1965 Edsger Dijkstra gave a deadlock-free solution for mutual exclusion [Dij65]. Later, Maurice Herlihy suggested consensus numbers [Her91], where he proved the "universality of consensus," i.e., the power of a shared memory system is determined by the consensus number. Peterson's Algorithm is due to [PF77, Pet81], and adaptive COLLECT was studied in the sequence of papers [MA95, AFG02, AL05, AKP+06].

Again, a big thanks goes to Roger Wattenhofer, whose lecture material today's topic is based on!

Bibliography

[AFG02] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[AKP+06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

[AL05] Yehuda Afek and Yaron De Levie. Space and Step Complexity Efficient Adaptive Collect. In *DISC*, pages 384–398, 2005.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[MA95] Mark Moir and James H. Anderson. Wait-Free Algorithms for Fast, Long-Lived Renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Pet81] J.L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

Lecture 11

Shared Counters

Maybe the most basic operation a computer performs is adding one, i.e., to count. In distributed systems, this can become a non-trivial task. If the events to be counted occur, e.g., at different processors in a multi-core system, determining the total count by querying each processor for its local count is costly. Hence, in shared memory systems, one may want to maintain a *shared counter* that permits to determine the count using a single or a few read operations.

11.1 A Simple Shared Counter

If we seek to implement such an object, we need to avoid that increments are "overwritten," i.e., two nodes increment the counter, but only one increment is registered. So, the simple approach of using one register and having a node incrementing the counter read the register and write the result plus one to the register is not good enough with atomic read/write registers only. With more powerful registers, things look differently.

Algorithm 25 Shared counter using compare-and-swap, code at node v.

Given: some shared register R, initialized to 0.
Increment:
1: **repeat**
2: $r := R$
3: success := compare-and-swap$(R, r, r + 1)$
4: **until** success = **true**
Read:
5: **return** R

11.1.1 Progress Conditions

Basically, this approach ensures that the read-write sequence for incrementing the counter behaves as if we applied mutual exclusion. However, there is a crucial difference. Unlike in mutual exclusion, no node obtains a "lock" and needs to release it before other nodes can modify the counter again. The algorithm is *lock-free*, meaning that it makes progress regardless of the schedule.

Definition 11.1 (Lock-Freedom)**.** *An operation is* lock-free*, if whenever any node is executing an operation, some node executing the same operation is guaranteed to complete it (in a bounded number of steps of that node). In asynchronous systems, this must hold even in (infinite) schedules that are not fair, i.e., if some of the nodes executing the operation may be stalled indefinitely.*

Lemma 11.2. *The increment operation of Algorithm 25 is lock-free.*

Proof. Suppose some node executes the increment code. It obtains some value r from reading the register R. When executing the compare-and-swap, it either increments the counter successfully or the register already contains a different value. In the latter case, some other node must have incremented the counter successfully. □

This condition is strong in the sense that the counter will not cease to operate because some nodes crash or are stalled for a long time. Yet, it is pretty weak with respect to read operations: It would admit that a node that just wants to read never completes this operation. However, as the read operations of this algorithm are trivial, they satisfy the strongest possible progress condition.

Definition 11.3 (Wait-Freedom)**.** *An operation is* wait-free *if whenever a node executes an operation, it completes if it is granted a bounded number of steps by the execution. In asynchronous systems, this must hold even in (infinite) schedules that are not fair, i.e., if nodes may be suspended indefinitely.*

Remarks:

- Wait-freedom is extremely useful in systems where one cannot guarantee reasonably small response times of other nodes. This is important in multi-core systems, in particular if the system needs to respond to external events with small delay.
- Consequently, wait-freedom is the gold standard in terms of progress. Of course, one cannot always afford gold.
- From the FLP theorem, we know that wait-free consensus is not possible without advanced RMW primitives.

11.1.2 Consistency Conditions

Progress is only a good thing if it goes in the right direction, so we need to figure out the direction we deem right. Even for such a simple thing as a counter, this is not as trivial as it might appear at first glance. If we require that the counter always returns the "true" value when read, i.e., the sum of the local event counts of all nodes, we cannot hope to implement this distributedly in any meaningful fashion: whatever is read at a single location may already be outdated, so we cannot satisfy the "traditional" sequential specification of a counter. Before we proceed to relaxing it, let us first formalize it.

Definition 11.4 (Sequential Object)**.** *A* sequential object *is given by a tuple* (S, s_0, R, O, t)*, where*

- S *is the set of states the object can attain,*

- s_0 *is its initial state,*
- R *is the set of values that can be read from the object,*
- O *is the set of operations that can be performed on the object, and*
- $t\colon O \times S \to S \times R$ *is the transition function of the object.*

A sequential execution *of the object is a sequence of operations* $o_i \in O$ *and states* $s_i \in S$*, where* $i \in \mathbb{N}$ *and* $(s_i, r_i) = t(o_i, s_{i-1})$*; operation* o_i returns *value* $r_i \in R$*.*

Definition 11.5 (Sequential Counter)**.** *A* counter *is the object given by* $S = \mathbb{N}_0$*,* $s_0 = 0$*,* $R = \mathbb{N}_0 \cup \{\perp\}$*,* $O = \{\mathit{read}, \mathit{increment}\}$*, and, for all* $i \in \mathbb{N}_0$*,* $t(\mathit{read}, i) = (i, i)$ *and* $t(\mathit{increment}, i) = (i + 1, \perp)$*.*

We could now "manually" define a distributed variant of a counter that we can implement. Typically, it is better to apply a generic *consistency condition*. In order to do this, we first need "something distributed" we can relate the sequential object to.

Definition 11.6 (Implementation)**.** *A (distributed)* implementation *of a sequential object is an algorithm*[1] *that enables each node to access the object using the operations from* O*. A node completing an operation obtains a return value from the set of possible return values for that operation.*

So far, this does not say anything about whether the returned values make any sense in terms of the behavior of the sequential object; this is addressed by the following definitions.

Definition 11.7 (Precedence)**.** *Operation* o precedes *operation* o' *if* o *completes before* o' *begins.*

Definition 11.8 (Linearizability)**.** *An execution of an implementation of an object is* linearizable*, if there is a sequential execution of the object such that*

- *there is a one-to-one correspondence between the performed operations,*
- *if* o *precedes* o' *in execution of the implementation, the same is true for their counterparts in the sequential execution, and*
- *the return values of corresponding operations are identical.*

An implementation of an object is linearizable if all its executions are linearizable.

Theorem 11.9. *Algorithm 25 is a linearizable counter implementation. Its read operations are wait-free and its increment operations are lock-free.*

Proof. All claims but linearizability are easily verified from the definitions and the algorithm. For linearizability, note that read operations are atomic, so we only need to worry about when we let a write operation take place in the linearization. This is easy, too: we choose the point in time when the successful compare-and-swap actually incrementing the value stored by R occurs. □

[1] Or rather a suite of subroutines that can be called, one for each possible operation.

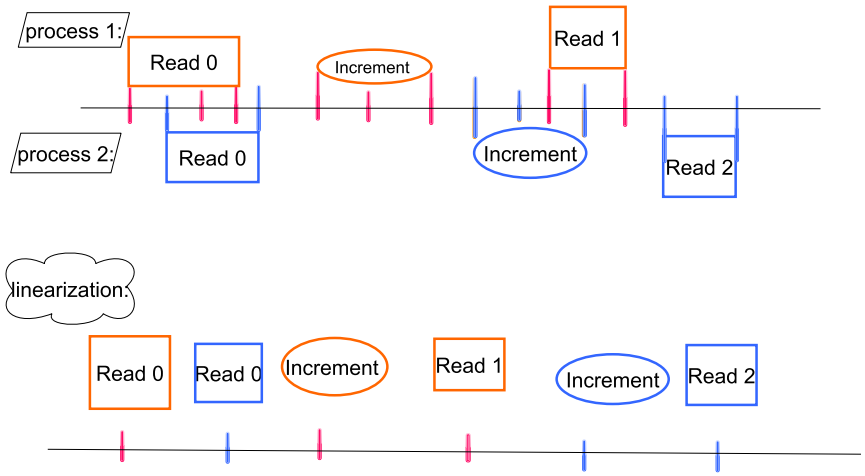

Figure 11.1: Top: An execution of a distributed counter implementation. Each mark is one atomic step of the respective node. Bottom: A valid linearization of the execution. Note that if the second read of node 1 would have returned 2, it would be ordered behind the increment by node 2. If it had returned 0, the execution would not be linearizable.

Remarks:

- Linearizability is extremely useful. It means that we can treat a (possibly horribly complicated) distributed implementation of an object as if it was accessed atomically.
- This makes linearizability the gold standard in consistency conditions. Unfortunately, also this gold has its price.
- Put simply, linearizability means "simulating sequential behavior," but not just any behavior – if some operation completed in the past, it should not have any late side effects.
- There are many equivalent ways of defining linearizability:
  - Extend the partial "precedes" order to a total order such that the resulting list of operation/return value pairs is a (correct) sequential execution of the object.
  - Assign strictly increasing times to the (atomic) steps of the execution of the implementation. Now each operation is associated with a time interval spanned by its first and last step. Assign to each operation a *linearization point* from its interval (such that no two linearization points are identical). This induces a total order on the operations. If this can be done in a way consistent with the specification of the object, the execution is linearizable.
- One can enforce linearizability using mutual exclusion.

- In the store & collect problem, we required that the "precedes" relation is respected. However, our algorithms/implementations were *not* linearizable. Can you see why?
- Coming up with a linearizable, wait-free, and efficient implementation of an object can be seen as creating a more powerful shared register out of existing ones.
- Shared registers are linearizable implementations of conventional registers.
- There are many weaker consistency conditions. For example one may just ask that the implementation behaves like its sequential counterpart only during times when a single node is accessing it.

11.2 No Cheap Wait-Free Linearizable Counters

There's a straightforward wait-free, linearizable shared counter using atomic read/write registers only: for each node, there's a shared register to which it applies increments locally; a read operation consists of reading all n registers and summing up the result.

This clearly is wait-free. To see that it is linearizable, observe that local increments require only a single write operation (as the node knows its local count), making the choice of the linearization point of the operation obvious. For each read, there must be a point in time between when it started and when it completes at which the sum of all registers equals the result of the read; this is a valid linearization point for the read operation.

Here's the problem: this seems very inefficient. It requires $n-1$ accesses to shared registers just to *read* the counter, and it also requires n registers. We start with the bad news. Even with the following substantially weaker progress condition, this is optimal.

Definition 11.10 (Solo-Termination)**.** *An operation is* solo-terminating *if it completes in finitely many steps provided that only the calling node takes steps (regardless of what happened before).*

Note that wait-freedom implies lock-freedom and that lock-freedom implies solo-termination.

Theorem 11.11. *Any linearizable deterministic implementation of a counter that guarantees solo-termination of all operations and uses only atomic read/write shared registers requires at least $n-1$ registers and has step complexity at least $n-1$ for read operations.*

Proof. We construct a sequence of executions $\mathcal{E}_i = \mathcal{I}_i\mathcal{W}_i\mathcal{R}_i$, $i \in \{0, \ldots, n-1\}$, where $\mathcal{E}_i$ is the concatenation of $\mathcal{I}_i$, $\mathcal{W}_i$, and $\mathcal{R}_i$. In each execution, the nodes are $\{1, \ldots, n\}$, and execution $\mathcal{E}_i$ is going to require i distinct registers; node n is the one reading the counter.

1. In $\mathcal{I}_i$, nodes $j \in \{1, \ldots, i\}$ increment the counter (some of these increments may be incomplete).

2. In $\mathcal{W}_i$, nodes $j \in \{1, \ldots, i\}$ each write to a different register R_j once and no other steps are taken.

3. In $\mathcal{R}_i$, node n reads the registers $R_1, \ldots, R_i$ as part of a (single) read operation on the counter.

As in $\mathcal{E}_{n-1}$ node n accesses $n-1$ different registers, this shows the claim.

The general idea is to "freeze" nodes $j \in \{1, \ldots, i\}$ just before they write to their registers R_j. This forces node $i+1$ to write to another register if it wants to complete many increments (which wait-freedom enforces) in a way that is not

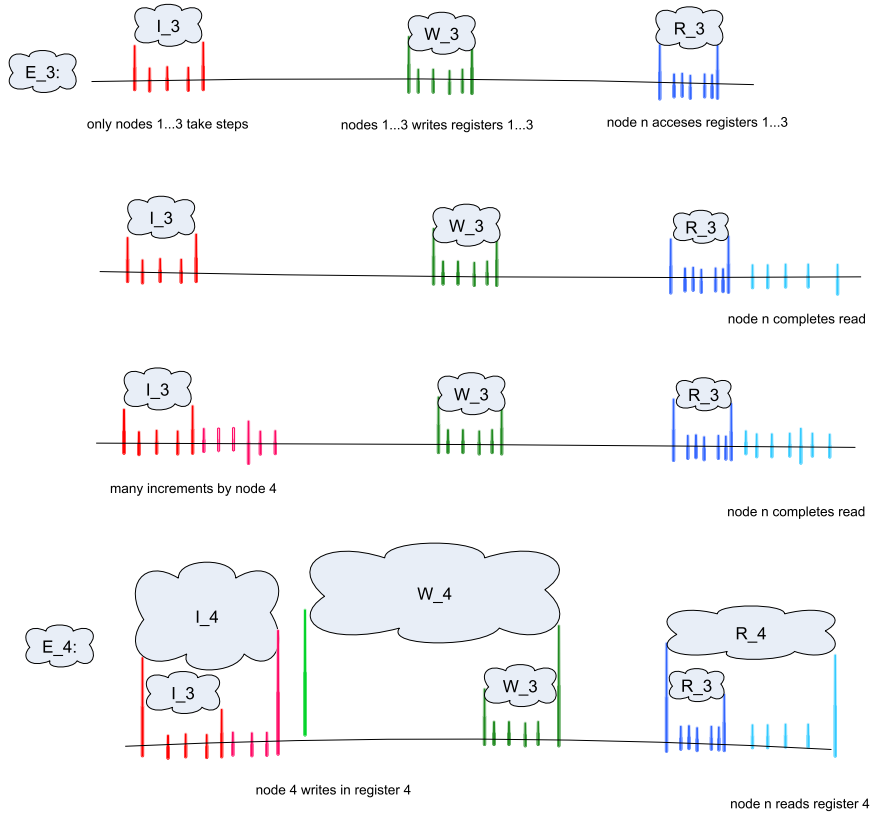

Figure 11.2: Example for the induction step from 3 to 4. Top: Execution $\mathcal{E}_3$. Second: We extend $\mathcal{E}_3$ by letting node n complete its read operation. Third: We consider the execution where we insert many increment operations by some unused node between $\mathcal{I}_3$ and $\mathcal{W}_3$. This might change how node n completes its read operation. However, if node n would not read a register not overwritten by $\mathcal{W}_3$ to which the new node writes, the two new executions would be indistinguishable to node n and its read would return a wrong (i.e., not linearizable) value in at least one of the them. Bottom: We let the new node execute until it writes the new register first and node n perform its read until it accesses the register first, yielding $\mathcal{E}_4$.

overwritten when we let nodes $1, \ldots, i$ perform their stalled write steps. This is necessary for node n to be able to complete a read operation without waiting for nodes $1, \ldots, i$; otherwise n wouldn't be able to distinguish between $\mathcal{E}_i$ and $\mathcal{E}_{i+1}$, which require different outputs if $i+1$ completed more increments than have been started in $\mathcal{E}_i$.

The induction is trivially anchored at $i = 0$ by defining $\mathcal{E}_0$ as the empty execution. Now suppose we are given $\mathcal{E}_i$ for $i < n-1$. We claim that node n must access some new register R_{i+1} before completing a read operation (its single task in all executions we construct). Assuming otherwise, consider the following execution. We execute $\mathcal{E}_i$ and then let node n complete its read operation. As the implementation is solo-terminating, this must happen in finitely many steps of n, and, by linearizability, the read operation must return at most the number k of increments that have been started in $\mathcal{E}_i$; otherwise, we reach a contradiction by letting these operations complete (one by one, using solo-termination) and observing that there is no valid linearization.

On the other hand, consider the execution in which we run $\mathcal{I}_i$, then let some node $j \in \{i+1, \ldots, n-1\}$ complete $k+1$ increments running alone (again possible by solo-termination), append $\mathcal{W}_i$, and let node n complete its read operation. Observe that the state of all registers $R_1, \ldots, R_i$ before node n takes any steps is the same as after $\mathcal{I}_i\mathcal{W}_i$, as any possible changes by node j were overwritten. Consequently, as n does not access any other registers, it cannot distinguish this execution from the previous run and thus must return a value of at most k. However, this contradicts linearizability of the new execution, in which already $k+1$ increments are complete. We conclude that when extending $\mathcal{E}_i$ by letting node n run alone, n will eventually access some new register R_{i+1}.

Define $\mathcal{R}_{i+1}$ as the sequence of steps n takes in this setting up to and including the first access to register R_{i+1}. W.l.o.g., assume that there exists an extension of $\mathcal{I}_i$ in which only nodes $i+1, \ldots, n-1$ take steps and eventually some $j \in \{i+1, \ldots, n-1\}$ writes to R_{i+1}. Otherwise, R_{i+1} is never going to be written (by a node different from n) in any of the executions we construct, i.e., node n cannot distinguish any of the executions we construct by reading R_{i+1}; hence it must read another register by repetition of the above argument. Eventually, there must be a register it reads that is written by some node $i+1 \leq j \leq n-1$ (if we extend $\mathcal{I}_i$ such that only node j takes steps), and we can apply the reasoning that follows.

W.l.o.g., assume that $j = i+1$ (otherwise we just switch the indices of nodes j and $i+1$ for the purpose of this proof) and denote by $\mathcal{I}_{i+1}w_{i+1}$ such an extension of $\mathcal{I}_i$, where w_{i+1} is the write of $j = i+1$ to R_{i+1}. Setting $W_{i+1} := w_{i+1}W_i$ and $\mathcal{E}_{i+1} := \mathcal{I}_{i+1}\mathcal{W}_{i+1}\mathcal{R}_{i+1}$ completes the induction and therefore the proof. □

Remarks:

- There was some slight cheating, as the above reasoning applies only to unbounded counters, which we can't have in practice anyway. Arguing more carefully, one can bound the number of increment operations required in the construction by $2^{\mathcal{O}(n)}$.
- The technique is far more general:
  - It works for many other problems, such as modulo counters, fetch-and-add, or compare-and-swap. In other words, using powerful RMW

registers just shifts the problem.

  - This can also seen by using reductions. Algorithm 25 shows that compare-and-swap cannot be easy to implement, and load-link/store-conditional can be used in the very same way. A fetch-and-add register is even better: it trivially implements a wait-free linearizable counter.
  - The technique works if one uses *historyless* objects in the implementation, not just RW registers. An object is historyless, if the resulting state of any operation that is not just a read (i.e., does never affect the state) does not depend on the current state of the object.
  - For instance, test-and-set registers are historyless, or even registers that can hold arbitrary values and return their previous state upon being written.
  - It also works for *resettable consensus* objects. These support the operations propose(i), $i \in \mathbb{N}$, reset, and read, and are initiated in state $\perp$. A propose(i) operation will result in state i if the object is in state $\perp$ and otherwise not affect the state. The reset operation brings the state back to $\perp$. This means that the hardness of the problem is not originating in an inability to solve consensus!
  - The space bound also applies to randomized implementations. Basically, the same construction shows that there is a positive probability that node n accesses $n-1$ registers, so these registers must exist. However, one can hope to achieve a small step complexity (in expectation or w.h.p.), as the probability that such an execution occurs may be very small.

- By now you might already expect that we're going to "beat" the lower bound. However, we're not going to use randomization, but rather exploit another loophole: the lower bound crucically relies on the fact that the counter values can become very large.

11.3 Efficient Linearizable Counter from RW Registers

Before we can construct a linearizable counter, we first need to better understand linearizability.

11.3.1 Linearizability "=" Atomicity

As mentioned earlier, a key feature of linearizability is that we can pretend that linearizable objects are atomic. In fact, this is the reason why it is standard procedure to assume that atomic shared registers are available: one simply uses a linearizable implementation from simpler registers. Let's make this more clear.

Definition 11.12 (Base objects)**.** *The* base objects *of an implementation of an object* **O** *are all the registers and (implementations of) objects that nodes may access when executing any operations of* **O**.

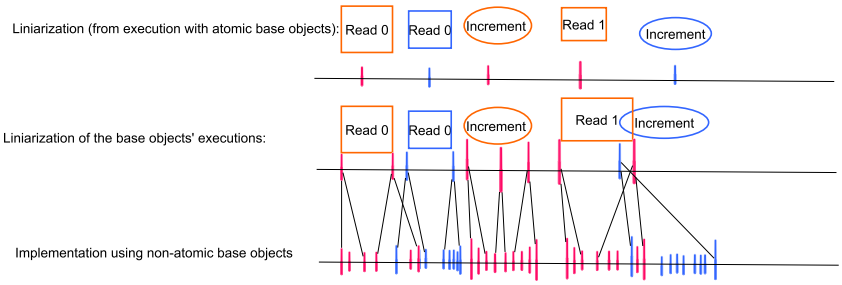

Figure 11.3: Bottom: An execution of an implementation using linearizable base objects. Center: Exploiting linearizability of each base object, we obtain an execution of a corresponding implementation from atomic base objects. Top: By linearizability of the assumed implementation from atomic base objects, this execution can be linearized, yielding a linearization of the original execution at the bottom.

Lemma 11.13. *Suppose some object* $\mathbf{O}$ *has a linearizable implementation using atomic base objects. Then replacing any atomic base object by a linearizable implementation (where each atomic access is replaced by calling the respective operation and waiting for it to complete) results in another linearizable implementation of* $\mathbf{O}$*.*

Proof. Consider an execution $\mathcal{E}$ of the constructed implementation of $\mathbf{O}$ from linearizable implementations of its base objects. By definition of linearizability, we can map the (sub)executions comprised of the accesses to (base objects of) the implementations of base objects to sequential executions of the base objects that preserve the partial order given by the "precedes" relation.

We claim that doing this for all of the implementations of base objects of $\mathbf{O}$ yields a valid execution $\mathcal{E}'$ of the given implementation of $\mathbf{O}$ from atomic base objects. To see this, observe that the view of a node in (a prefix of) $\mathcal{E}$ is given by its initial state and the sequence of return values from its previous calls to the atomic base objects. In $\mathcal{E}$, the node calls an operation once all its preceding calls to operations are complete. As, by definition of linearizability, the respective return values are identical, the claim holds true.

The rest is simple. We apply linearizability to $\mathcal{E}'$, yielding a sequential execution $\mathcal{E}''$ of $\mathbf{O}$ that preserves the "precedes" relation on $\mathcal{E}'$. Now, if operation o precedes o' in $\mathcal{E}$, the same holds for their counterparts in $\mathcal{E}'$, and consequently for *their* counterparts in $\mathcal{E}''$; likewise, the return values of corresponding operations match. Hence $\mathcal{E}''$ is a valid linearization of $\mathcal{E}$. □

Remarks:

- Beware side effects, as they break this reasoning! If a call to an operation affects the state of the node (or anything else) beyond the return value, this can mess things up.
- For instance, one can easily extend this reasoning to randomized implementations. However, in practical systems, randomness is usually not

"true" randomness, and the resulting dependencies can be... interesting.

- Lemma 11.13 permits to abstract away the implementation details of more involved objects, so we can reason hierarchically. This will make our live much, *much* easier!

- This result is the reason why it is common lingo to use the terms "atomic" and "linearizable" interchangably.

- We're going to exploit this to the extreme now. Recursion time!

11.3.2 Counters from Max Registers

We will construct our shared counter using another, simpler object.

Definition 11.14 (Max Register)**.** *A* max register *is the object given by* $S = \mathbb{N}_0$, $s_0 = 0$, $R = \mathbb{N}_0 \cup \{\bot\}$, $O = \{read, write(i) \mid i \in \mathbb{N}_0\}$, *and, for all* $i, j \in \mathbb{N}_0$, $t(read, i) = (i, i)$ *and* $t(write(i), j) = (\max\{i, j\}, \bot)$. *In words, the register always returns the maximum previously written value on a read.*

Max registers are not going to help us, as the lower bound applies when constructing them. We need a twist, and that's requiring a bound on the maximum value the counter – and thus the max registers – can attain.

Definition 11.15 (Bounded Max Register)**.** *A* max register with maximum value $M \in \mathbb{N}$ *is the object given by* $S = \{0, \ldots, M\}$, $s_0 = 0$, $R = S \cup \{\bot\}$, $O = \{read, write(i) \mid i \in S\}$, *and, for all* $i, j \in S$, $t(read, i) = (i, i)$ *and* $t(write(i), j) = (\max\{i, j\}, \bot)$.

Definition 11.16 (Bounded Counter)**.** *A* counter with maximum value $M \in \mathbb{N}$ *is the object given by* $S = \{0, \ldots, M\}$, $s_0 = 0$, $R = S \cup \{\bot\}$, $O = \{read, increment\}$, *and, for all* $i \in S$, $t(read, i) = (i, i)$ *and* $t(increment, i) = (\min\{i + 1, M\}, \bot)$.

Before discussing how to implement bounded max registers, let's see how we obtain an efficient wait-free linearizable bounded counter from them.

Lemma 11.17. *Suppose we are given two atomic counters of maximum value M that support k incrementing nodes (i.e., no more than k different nodes have the ability to use the increment operation) and an atomic max register of maximum value M. Then we can implement a counter with maximum value M and the following properties.*

- *It supports $2k$ incrementing nodes.*

- *It is linearizable.*

- *All operations are wait-free.*

- *The step complexity of reads is 1, a read of a max register.*

- *The step complexity of increments is 4, where only one of the steps is a counter increment.*

Proof. Denote the counters by C_1 and C_2 and assign k nodes to each of them. Denote by R the max register. To read the new counter C, one simply reads R. To increment C, a node increments its assigned counter, reads both counters, and writes the sum to R. Obviously, we now support $2k$ incrementing nodes, all operations are wait-free, and their step complexity is as claimed. Hence, it remains to show that the counter is linearizable.

Fix an execution of this implementation of C. We need to construct a corresponding execution of a counter of maximum value M. At each point in time, we rule that the state of C is the state of R.[2] Thus, we can map the sequence of read operations to the same sequence of read operations; all that remains is to handle increments consistently. Suppose a node applies an increment. Denote by σ the sum of the two counter values right after it incremented its assigned counter. At this point, $r < \sigma$, where r denotes the value stored in R, as no node ever writes a value larger than the sum it read from the two counters to R. As the node reads C_1 and C_2 after incrementing its assigned counter, it will read a sum of at least σ and subsequently write it to R. We conclude that at some point during the increment operation the node performs on C, R will attain a value of at least σ, while before it was smaller than σ. We map the increment of the node to this step.

To complete the proof, we need to check that the result is a valid linearization. For each operation o, we have chosen a linearization point $l(o)$ during the part of the execution in which the operation is performed. Thus, if o precedes o', we trivially have that $l(o) < l(o')$. As only reads have return values different from $\perp$ and clearly their return values match the ones they should have for a max register whose state is given by R, we have indeed constructed a valid linearization. □

Corollary 11.18. *We can implement a counter with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of reads is* 1*.*
- *The step complexity of each increment operation is* $3\lceil \log n \rceil + 1$*.*
- *Its base objects are $\mathcal{O}(n)$ atomic read/write registers and max registers of maximum value M.*

Proof. W.l.o.g., suppose $n = 2^i$ for some $i \in \mathbb{N}_0$. We show the claim by induction on i, where the bound on the step complexity of increments is $3i + 1$. For the base case, observe that a linearizable wait-free counter with a single node that may increment it is given by a read/write register that is written by that node only, and it has step complexity 1 for all operations.

Now assume that the claim holds for some $i \in \mathbb{N}_0$. By the induction hypothesis, we have linearizable wait-free counters supporting 2^i incrementing nodes

[2]This is a slight abuse of notation, as it means that multiple increments may take effect at the same instant of time. Formally, this can be handled by splitting them up into individual increments that happen right after each other.

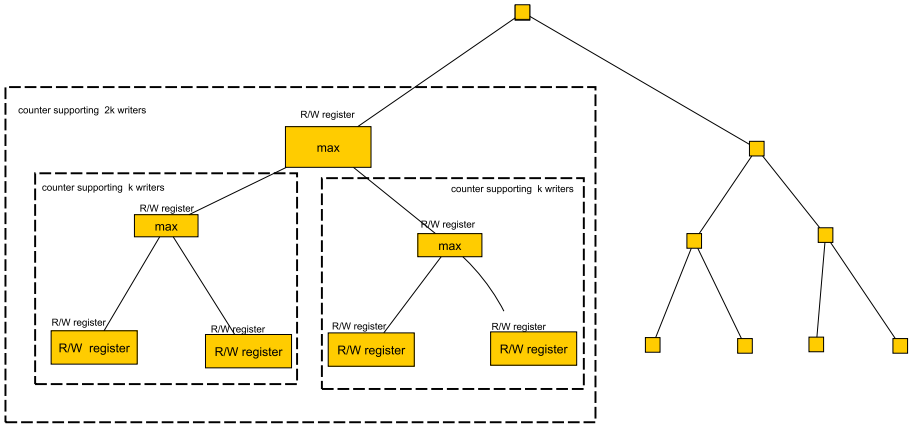

Figure 11.4: The recursive construction from Corollary 11.18, resulting in a tree. The leaves are simple read/write registers, which can be used as atomic counters with a single writer. A subtree of depth d implements a linearizable counter supporting 2^d writers, and by Lemma 11.13 it can be treated as atomic. Using a single additional max register, Lemma 11.19 shows how construct a counter supporting 2^{d+1} writers using 2 counters supporting 2^d writers.

(with the "right" step complexities and numbers of registers). If these were atomic, Lemma 11.17 would immediately complete the induction step. Applying Lemma 11.13, it suffices that they are linearizable implementations, i.e., the induction step succeeds. □

Remarks:

- This is an application of a reliable recipe: Construct something linearizable out of atomic base objects, "forget" that it's an implementation, pretend its atomic, rinse and repeat.
- Doing it without Lemma 11.13 would have meant to unroll the argument for the entire tree construction of Corollary 11.18, which would have been cumbersome and error-prone at best.

11.3.3 Max Registers from RW Registers

The construction of max registers with maximum value M from basic RW registers is structurally similar.

Lemma 11.19. *Suppose we are given two atomic max registers of maximum value M and an atomic read/write register. Then we can implement a max register with maximum value $2M$ and the following properties from these.*

- *It is linearizable.*
- *All operations are wait-free.*
- *Each read operation consists of one read of the RW register and reading one of the max registers.*

- *Each write operation consists of at most one read of the RW register and writing to one of the max registers.*

The construction is given in Algorithm 26. The proof of linearizability is left for the exercises.

Algorithm 26 Recursive construction of a max register of maximum value $2M$ from two max registers of maximum value M and a read/write register.

Given: max registers $R_<$ and $R_\geq$ of maximum value M, and RW register switch, all initialized to 0.

read

```
 1: if switch = 0 then
 2:    return R_<.read
 3: else
 4:    return M + R_≥.read
 5: end if
```

write(i)

```
 6: if i < M then
 7:    if switch = 0 then
 8:       R_<.write(i)
 9:    end if
10: else
11:    R_≥.write(i − M)
12:    switch := 1
13: end if
14: return ⊥
```

Corollary 11.20. *We can implement a max register with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of all operations is $\mathcal{O}(\log M)$.*
- *Its base objects are $\mathcal{O}(M)$ atomic read/write registers.*

Proof sketch. Like in Corollary 11.18, we use Lemmas 11.13 and 11.19 inductively, where in each step of the induction the maximum value of the register is doubled. The base case of $M = 1$ is given by a read/write register initialized to 0: writing 0 requires no action, and writing 1 can be safely done, since no other value is ever (explicitly) written to the register; since both reads and writes require at most one step, the implementation is trivially linearizable. □

Theorem 11.21. *We can implement a counter with maximum value M and the following properties.*

- *It is linearizable.*
- *All operations are wait-free.*
- *The step complexity of reads is $\mathcal{O}(\log M)$.*

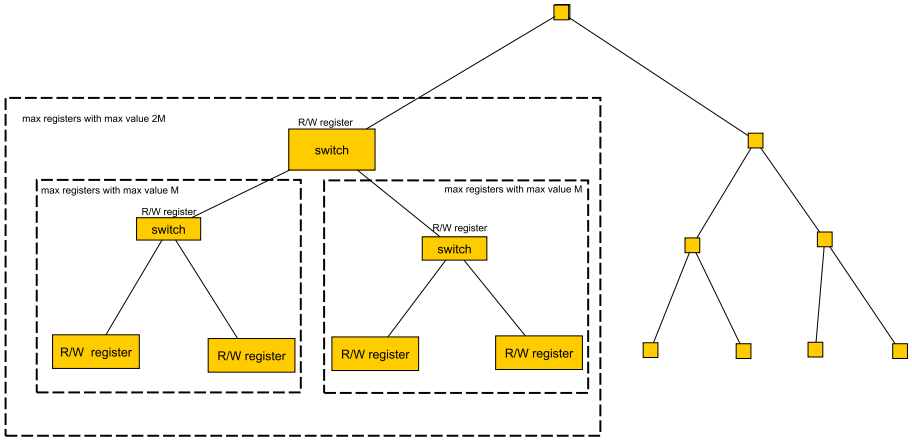

Figure 11.5: The recursive construction from Corollary 11.20, resulting in a tree. The leaves are simple read/write registers, which can be used as atomic max registers with maximum value 1. A subtree of depth d implements a linearizable max register of maximum value 2^d, and by Lemma 11.13 it can be treated as atomic. Using a single read/write register "switch," Lemma 11.19 shows how to control access to two max registers with maximum value 2^d to construct one with maximum value 2^{d+1}.

- *The step complexity of each increment operation is* $\mathcal{O}(\log M \log n)$.
- *Its base objects are* $\mathcal{O}(nM)$ *atomic read/write registers.*

Proof. We apply Lemmas 11.13 and 11.18 to the implementations of max registers of maximum value M given by Corollary 11.20. The step complexities follow, as we need to replace each access to a max register by the step complexity of the implementation. Similarly, the total number of registers is the number of read/write registers per max register times the number of used max registers (plus an additive $\mathcal{O}(n)$ read/write registers for the counter implementation that gets absorbed in the constants of the $\mathcal{O}$-notation). □

Remarks:

- As you will show in the exercises, writing to $R_<$ only if switch reads 0 is crucial for linearizability.
- If M is $n^{\mathcal{O}(1)}$, reads and writes have step complexities of $\mathcal{O}(\log n)$ and $\mathcal{O}(\log^2 n)$, respectively, and the total number of registers is $n^{\mathcal{O}(1)}$. As for many algorithms and data structures only polynomially many increments happen, this is a huge improvement compared to the linear step complexity the lower bound seems to imply!
- If one has individual caps c_i on the number of increments a node may perform, one can use respectively smaller registers. This improves the space complexity to $\mathcal{O}(\log n \sum_{i=1}^{n} c_i)$, as on each of the $\lceil \log n \rceil$ hierarchy levels (read: levels of the tree) of the counter construction, the max registers must be able to hold $\sum_{i=1}^{n} c_i$ *in total*, but not individually.

- For instance, if one wants to know the number of nodes participating in some algorithm or subroutine, this becomes $\mathcal{O}(n \log n)$.

- One can generalize the construction to cap the step complexity at n. However, at this point the space complexity is already exponential.

What to take Home

- This is another example demonstrating how lower bounds do more than just giving us a good feeling about what we've done. The lower bound was an essential guideline for the max register and counter constructions, as it told us that the bottleneck was the possibility of a very large number of increments!

- Advanced RMW registers are very powerful. At the same time, this means they are very expensive.

- Understanding and proving consistency for objects that should behave like they are accessed sequentially is challenging. One may speculate that we're not seeing the additional computational power of multi-processor systems with many cores effectively used in practice, due to the difficulty of developing scalable parallel software.

Bibliographic Notes

This lecture is based largely on two papers: Jayanti et al. [JTT00] proved the lower bounds on the space and step complexity of counters and, as discussed in the remarks, many other shared data structures. The presented counter implementation was developed by Aspnes et al. [AACH12]. In this paper, it is also shown how to use the technique to implement general *monotone* circuits (i.e., things only "increase," like for a counter), though the result is not linearizable, but satisfies a weaker consistency condition. Moreover, the authors show that randomized implementations of max registers with maximum value n must have step complexity $\Omega(\log n / \log \log n)$ for read operations, assuming that write operations take $\log^{\mathcal{O}(1)} n$ steps. In this sense their deterministic implementation is almost optimal!

Randomization [AC13] or using (deterministic) linearizable snapshots that support only polynomially many write operations of the underlying data structure [AACHE12] are other ways to circumvent the linear step complexity lower bound. Finally, any implementation of a max register of maximum value M from historyless objects requires $\Omega(\min\{M, n\})$ space [AACHH12].

Bibliography

[AACH12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, 2012.

[AACHE12] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster Than Optimal Snapshots (for a While): Preliminary Version. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, pages 375–384, 2012. Journal preprint at http://cs-www.cs.yale.edu/homes/aspnes/papers/limited-use-snapshots-abstract.html.

[AACHH12] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Danny Hendler. Lower bounds for restricted-use objects. In *Twenty-Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 172–181, 2012.

[AC13] James Aspnes and Keren Censor-Hillel. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 254–268, 2013.

[JTT00] P. Jayanti, K. Tan, and S. Toueg. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.