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 other nodes are just one shared register away; think for instance about pointer jumping, which is not possible in a message passing system, or MST construction, where we where concerned about the strong diameter of components.

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>
A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds

- *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.*

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 extensively.

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 no updates have occurred to that register since the load-link. If any updates have occurred, 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 "successfully" execute the test-and-set, i.e., the result of the test-and-set 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 given by this algorithm. Even with only two nodes there are asynchronous executions where 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 so-called *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.

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,[1] 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. In the remarks, we discuss how it can be extended. 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.*

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

Algorithm 21 Mutual exclusion: Peterson's algorithm.

```
Given: shared registers $W_0, W_1, \Pi$, 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. Argueing 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), it either will stop trying to enter or 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 so-called "spinlock" or "busy-wait", similarly to the 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,[2] as only signal *transitions* are "expensive" (i.e., costly in terms of energy) in circuits.

Store & collect

We will now look at a similar shared memory problem. Informally, the problem can be stated as follows. There are n nodes $p_1, \dots, p_n$. 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 (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*, a function* $f : V \to VAL \cup \{\bot\}$ *from the set of nodes* V *to a set of values* VAL *or the symbol* $\bot$*, 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* $V(p_i) = \bot$*, then no* STORE *operation by* p_i *precedes* cop.
- *If* $V(p_i) = val \neq \bot$*, then val is the value of a* STORE *operation* sop *of* p_i *that does not follow* cop*, and 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 $\bot$. 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 as shown by Algorithm 22.

[2]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,[3] 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 costly. 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.

Splitters

Assume that up to a certain time t, $k \leq n$ nodes have finished or started at least one operation. We call an operation at time t *adaptive* to contention, if its step complexitydepends 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 characteristic. 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

[3]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: $X : \{\perp\} \cup \{1, \dots, n\}$; $Y$ : 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
```

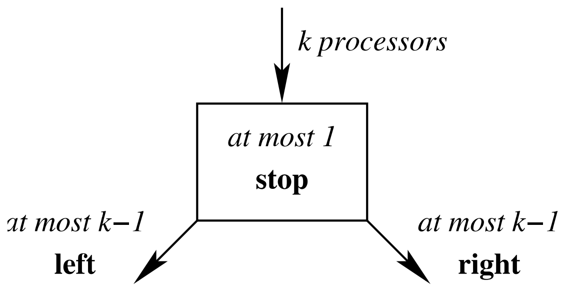

Figure 10.1: A Splitter

(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. The code implementing a splitter is given by Algorithm 23.

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**. □

Binary Splitter Tree

Assume that we are given $2^n - 1$ splitters and that for every splitter S, there is an additional shared variable $Z_S : \{\perp\} \cup \{1, \ldots, n\}$ that is initialized to $\perp$ and an additional shared variable M_S : **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:     $\alpha :=$ result of entering splitter $S(v)$;
5:     $M_{S(v)} :=$ true
6:     while $\alpha \neq$ stop do
7:        if $\alpha =$ left then
8:           $v :=$ left child of $v$
9:        else
10:          $v :=$ right child of $v$
11:       end if
12:       $\alpha :=$ 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 $Z_S \neq \perp$ then
19:       $i := Z_S$; $V(p_i) := R_i$                // read value of node $p_i$
20:    end if
21: end for                           // $V(p_i) = \perp$ 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 $\mathcal{O}(k)$ 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, where k nodes access the root. We claim that $x_k \leq 2k-1$, which is true for $k=1$ because a single node entering a splitter will always compute **stop**. Now assume the inequality holds for $1, \dots, k-1$. Not all k nodes may exit the splitter with **left** (or **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

$$x_k = x_{k_l} + x_{k_r} + 1 \leq (2k_l - 1) + (2k_r - 1) + 1 \leq 2k - 1,$$

concluding induction and proof. □

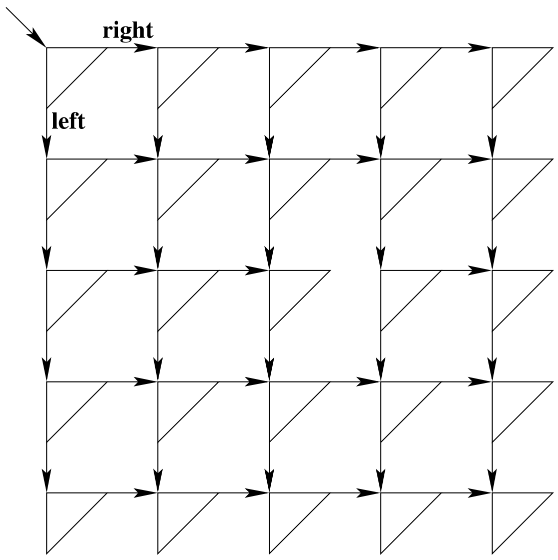

Figure 10.2: 5×5 Splitter Matrix

Remarks:

- The step complexities of Algorithm 24 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal.[4] 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.

10.0.1 Splitter Matrix

Instead of arranging splitters in a binary 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 that $x_i \leq k - i$. Clearly, $x_0 \leq k$. Let us therefore consider the case $i > 0$. Let j

[4]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.

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 $\mathcal{O}(k)$. $\square$

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. Petersons 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.