Chapter 13

Shared Memory

13.1 Introduction

Throughout this course we have already seen various models of distributed computing. So far, the focus of the course was on loosely-coupled distributed systems such as the Internet, where nodes asynchronously communicate by exchanging messages. The "opposite" model is a tightly-coupled parallel computer where nodes access a common memory totally synchronously—in distributed computing such a system is called a Parallel Random Access Machine (PRAM).

A third major model is somehow between these two extremes, the *shared memory* model. In a shared memory system, asynchronous processors communicate via a common memory area of shared variables or registers:

Definition 13.1 (Shared Memory)**.** *A shared memory system is a system that consists of asynchronous processors that access a common (shared) memory. A processor can atomically access a register in the shared memory through a set of predefined operations. Apart from this shared memory, processors can also have some local (private) memory.*

Remarks:

- Various shared memory systems exist. A main difference is how they allow processors to access the shared memory. All systems can atomically read or write a shared register. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations on registers r, 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): $t := r$; if $r = x$ then $r := y$ endif; return t
  - load-link/store-conditional: Load-link returns the current value of the specified register. A subsequent store-conditional to the same register will store a new value 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, even if the value read by the load-link has since been restored

- Maurice Herlihy suggested that the power of RMW operations can be measured with the so-called *consensus-number*: The consensus-number of a RMW operation defines whether one can solve consensus for k processors. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processors, but not 3), whereas the consensus-number of compare-and-swap is infinite. In his 1991 paper, Maurice Herlihy proved the "universality of consensus", i.e., the power of a shared memory system is determined by the consensus-number. This insight had a remarkable theoretical and practical impact. In practice for instance, hardware designers stopped developing shared memory systems supporting weak RMW operations. Consequently, Maurice Herlihy was awarded the Dijkstra Prize in Distributed Computing in 2003.

- Many of the results derived in the message passing model have an equivalent in the shared memory model. Consensus for instance is traditionally studied in the shared memory model.

- Whereas programming a message passing system is rather tricky (in particular if fault-tolerance has to be integrated), programming a shared memory system is generally considered easier, as programmers are given access to global variables directly and do not need to care about shared object support as discussed in Chapter 12. Because of this, even distributed systems which physically communicate by exchanging messages can often be programmed through a shared memory middleware, making the programmer's life easier.

- We will most likely find the general spirit of shared memory systems in upcoming multi-core architectures. As for programming style, the multi-core community seems to favor an accelerated version of shared memory, *transactional memory*.

- From a message passing perspective, the shared memory model is like a bipartite graph: One one side you have the processors (the nodes) which pretty much behave like nodes in the message passing model (asynchronous, maybe failures). On the other side you have the shared registers, which just work perfectly (no failures, no delay).

13.2 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion.[1] We are given a number of processes which occasionally need to access the same resource. The resource may be a shared variable, or a more general object such as a data structure or a shared printer. The catch is that only one process at the time is allowed to access the resource. More formally:

Definition 13.2 (Mutual Exclusion)**.** *We are given a number of processes, each executing the following code sections:*
<Entry> → <Critical Section> → <Exit> → <Remaining Code>

[1] In message passing systems mutual exclusion can be done by shared objects, see Chapter 12.

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 *processor is in the critical section.*
- *No deadlock: If some processor manages to get to the entry section, later* some *(possibly different) processor will get to the critical section.*

Sometimes we in addition ask for

- *No lockout: If some processor manages to get to the entry section, later* the same *processor will get to the critical section.*
- *Unobstructed exit: No processor can get stuck in the exit section.*

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 46 shows an example with the test-and-set primitive.

Algorithm 46 Mutual Exclusion: Test-and-Set

Initialization: Shared register $R = 0$

<**Entry**>

```
1: repeat
2:     r := test-and-set(R)
3: until r=0
```

<**Critical Section**>

```
4: ...
```

<**Exit**>

```
5: R := 0
```

<**Remainder Code**>

```
6: ...
```

Theorem 13.3. *Algorithm 46 solves the mutual exclusion problem as in Definition 13.2.*

Proof. Mutual exclusion follows directly from the test-and-set definition: Initially R is 0. Let p_i be the i^{th} processor to successfully execute the test-and-set, where successfully means that the result of the test-and-set is 0. This happens at time t_i. At time t_i' processor p_i resets the shared register R to 0. Between t_i and t_i' no other processor can successfully test-and-set, hence no other processor can enter the critical section concurrently.

Proofing no deadlock is similar: One of the processors loitering in the entry section will successfully test-and-set as soon as the process in the critical section exited.

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

No lockout, on the other hand, is not given by this algorithm. Even with only two processors there are asynchronous executions where always the same processor wins the test-and-set. □

Remarks:

- Algorithm 46 can be adapted to guarantee fairness (no lockout), essentially by ordering the processors in the entry section in a queue.
- A natural question is whether one can achieve mutual exclusion with only reads and writes, that is without advanced RMW operations. The answer is yes!

Our read/write mutual exclusion algorithm is for two processors p_0 and p_1 only. In the remarks we discuss how it can be extended. The general idea is that processor 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 processor is not interested ($w_{1-i} = 0$) access is granted. This however is too simple since we may run into a deadlock. This deadlock (and at the same time also lockout) is resolved by adding a priority variable π. See Algorithm 47.

Algorithm 47 Mutual Exclusion: Petersen's Algorithm

Initialization: Shared registers w_0, w_1, π, all $= 0$.
Code for processor p_i , i={0,1}
<**Entry**>
1: $w_i := 1$
2: $\pi := 1 - i$
3: **repeat until** $\pi = i$ or $w_{1-i} = 0$
<**Critical Section**>
4: ...
<**Exit**>
5: $w_i := 0$
<**Remainder Code**>
6: ...

Remark:

- Note that line 3 in Algorithm 47 represents a "spinlock" or "busy-wait", similarly to the lines 1-3 in Algorithm 46.

Theorem 13.4. *Algorithm 47 solves the mutual exclusion problem as in Definition 13.2.*

Proof. The shared variable π elegantly grants priority to the processor that passes line 2 first. If both processors are competing, only processor p_π can access the critical section because of π. The other processor $p_{1-\pi}$ cannot access the critical section because $w_\pi = 1$ (and $\pi \neq 1 - \pi$). The only other reason to access the critical section is because the other processor is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with π: Processor p_π gets direct access to the critical section, no matter what the other processor does.

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

Thanks to the shared variable π also no lockout (fairness) is achieved: If a processor p_i loses against its competitor p_{1-i} in line 2, it will have to wait until

the competitor resets $w_{1-i} := 0$ in the exit section. If processor p_i is unlucky it will not check $w_{1-i} = 0$ early enough before processor p_{1-i} sets $w_{1-i} := 1$ again in line 1. However, as soon as p_{1-i} hits line 2, processor p_i gets the priority register π, and can enter the critical section. □

Remark:

- Extending Petersen's Algorithm to more than 2 processors can be done by a tournament tree, like in tennis. With n processors every processor needs to win $\log n$ matches before it can enter the critical section. More precisely, each processor 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. Thanks to the priority variables π at each node of the binary tree, we inherit all the properties of Definition 13.2.

13.3 Store & Collect

13.3.1 Problem Definition

In this section, we will look at a second shared memory problem that has an elegant solution. Informally, the problem can be stated as follows. There are n processors $p_1, \ldots, p_n$. Every processor 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 processors. Further, there is an operation by which a processor can *collect* (i.e., read) the values of all the processors 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 13.5 (Collect)**.** *There are two operations: A* STORE(*val*) *by process p_i sets val to be the latest value of its register r_i. A* COLLECT *operation returns a* view*, a partial function V from the set of processors to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each processor p_i. For a* COLLECT *operation* cop*, the following validity properties must hold for every processor p_i:*

- *If $V(p_i) = \perp$, then no* STORE *operation by p_i precedes* cop*.*
- *If $V(p_i) = v \neq \perp$, then v 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*.*

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

We assume that the read/write register r_i of every processor p_i is initialized to $r_i = \perp$. 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 as shown by Algorithm 48.

Remarks:

- Algorithm 48 clearly works. The step complexity of every STORE operation is 1, the step complexity of a COLLECT operation is n.

Algorithm 48 Collect: Simple (Non-Adaptive) Solution

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

Algorithm 49 Splitter Code

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

Splitter access by processor 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**

- At first sight, the step complexities of Algorithm 48 seem optimal. Because there are n processors, there clearly 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 seems very costly. Assume that there are only two processors p_i and p_j that have stored a value in their registers r_i and r_j. In this case, a COLLECT in principle only needs to read the registers r_i and r_j and can ignore all the other registers.
- Assume that up to a certain time t, $k \leq n$ processors have finished or started at least one operation. We call an operation *op* at time t *adaptive* to contention if the step complexity of *op* only depends on k and is independent of n.
- In the following, we will see how to implement adaptive versions of STORE and COLLECT.

13.3.2 Splitters

To obtain adaptive collect algorithms, we need a synchronization primitive, called a *splitter*.

Definition 13.6 (Splitter)**.** *A splitter is a synchronization primitive with the following characteristic. A processor entering a splitter exits with either* ***stop****,* ***left****, or* ***right****. If k processors enter a splitter, at most one processor exits with* ***stop*** *and at most $k-1$ processors enter with* ***left*** *and* ***right****, respectively.*

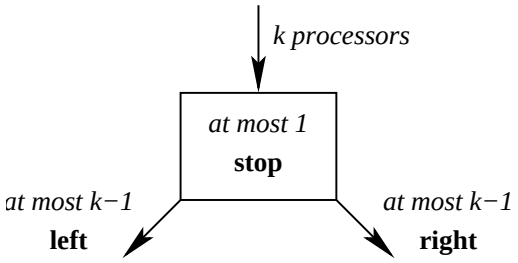

Figure 13.1: A Splitter

Hence, it is guaranteed that if a single processor enters the splitter, then it obtains **stop**, and if two or more processors enter the splitter, then there is at most one processor obtaining **stop** and there are two processors that obtain different values (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 13.1. The code implementing a splitter is given by Algorithm 49.

Lemma 13.7. *Algorithm 49 correctly implements a splitter.*

Proof. Assume that k processors enter the splitter. It is sufficient to show that at most 1 returns **stop**, at most $k-1$ return **left**, and at most $k-1$ return **right**. Because the first processor that checks whether $Y =$ **true** in line 2 will find that $Y =$ **false**, not all processors return **right**. Next, assume that i is the last processor 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 processor that does not return **left**. It remains to show that at most 1 processor returns **stop**. For the sake of contradiction, assume p_i and p_j are two processors that return **stop** and assume that p_i sets $X := i$ before p_j sets $X := j$. Both processors need to check whether Y is **true** before one of them sets $Y :=$ **true**. Hence, they both complete the assignment in line 1 before the first one of them checks the value of X in line 6. Hence, by the time p_i arrives at line 6, $X \neq i$ (p_j and maybe some other processors have overwritten X by then). Therefore, p_i does not return **stop** and we get a contradiction to the assumption that both p_i and p_j return **stop**. □

13.3.3 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 $Z_S = \perp$ and an additional shared variable $M_S :$ **boolean** that is initialized to $M_S =$ **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 50.

Theorem 13.8. *Algorithm 50 correctly implements* STORE *and* COLLECT. *Let k be the number of participating processes. The step complexity of the first* STORE *of a processor p_i is $O(k)$, the step complexity of every additional* STORE *of p_i is $O(1)$, and the step complexity of* COLLECT *is $O(k)$.*

Algorithm 50 Adaptive Collect: Binary Tree Algorithm

Operation STORE(val) (by processor 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 processor* p_i
20: **end if**
21: **end for** // $V(p_i) = \perp$ *for all other processors*

Proof. Because at most one processor can stop at a splitter, it is sufficient to show that every processor 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 $k-i$ processors enter a subtree at depth i (i.e., a subtree where the root has distance i to the root of the whole tree). By definition of k, the number of processors entering the splitter at depth 0 (i.e., at the root of the binary tree) is k. For $i > 1$, the claim follows by induction because of the at most $k-i$ processors entering the splitter at the root of a depth i subtree, at most $k-i-1$ obtain **left** and **right**, respectively. Hence, at the latest when reaching depth $k-1$, a processor is the only processor entering a splitter and thus obtains **stop**. It thus also follows that the step complexity of the first invocation of STORE is $O(k)$.

To show that the step complexity of COLLECT is $O(k)$, we show that at most $2k-1$ nodes of the binary tree are marked. Let x_k be the number of marked nodes in a tree, where k processors access the root. The splitter properties imply the following recursive equations:

$$x_k \leq x_i + x_{k-i-1} + 1, \qquad (i \geq 0) \qquad (13.1)$$

$$x_k \leq x_i + x_{k-i} + 1, \qquad (i > 0) \qquad (13.2)$$

Equation (13.1) holds if a process stops in the splitter; otherwise, Equation (13.2) holds.

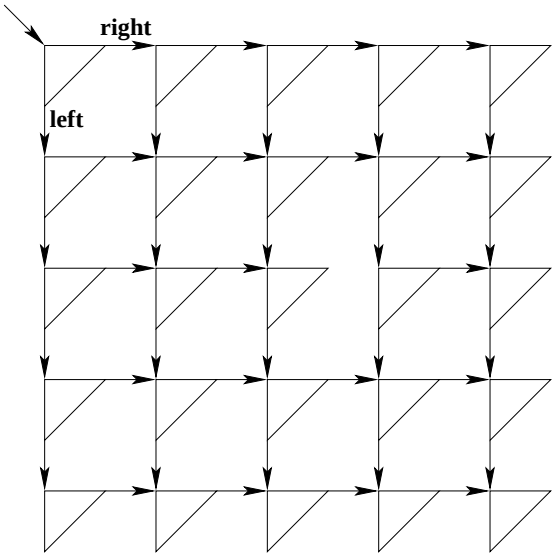

Figure 13.2: 5×5 Splitter Matrix

We prove the claim by induction; note that it trivially holds for $k = 1$. For the induction step, assume the claim is true for $j < k$, that is, $x_j \leq 2j - 1$. Then we can rewrite Equation (13.1):

$$x_k \leq (2i - 1) + (2(k - i - 1) - 1) + 1 \leq 2k - 1$$

and Equation (13.2) becomes:

$$x_k \leq (2i - 1) + (2(k - i) - 1) + 1 \leq 2k - 1.$$

□

Remark:

- The step complexities of Algorithm 50 are very good. Clearly, the step complexity of the COLLECT operation is asymptotically optimal. 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 50 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.

13.3.4 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 13.2. The algorithm is analogous to Algorithm 50. The matrix is entered at the top left. If a processor receives **left**, it next visits the splitter in the next row of the same column. If a processor receives **right**, it next visits the splitter in the next column of the same row. Clearly, the space complexity of this algorithm is $O(n^2)$. The following theorem gives bounds on the step complexities of STORE and COLLECT.

Theorem 13.9. *Let k be the number of participating processes. The step complexity of the first* STORE *of a processor p_i is $O(k)$, the step complexity of every additional* STORE *of p_i is $O(1)$, and the step complexity of* COLLECT *is $O(k^2)$.*

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

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of processors 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.
- The binary tree algorithm can be made space efficient by using a randomized version of a splitter. Whenever returning left or right, a randomized splitter returns left or right with probability 1/2. With high probability, it then suffices to allocate a binary tree of depth $O(\log n)$.
- Recently, it has been shown that with a considerably more complicated deterministic algorithm, it is possible to achieve $O(k)$ step complexity and $O(n^2)$ space complexity.