Chapter 5

Shared Memory

In distributed computing, various different models exist. 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).

5.1 Model

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

Definition 5.1 (Shared Memory). A shared memory system is a system that consists of asynchronous processes that access a common (shared) memory. A process can atomically access a register in the shared memory through a set of predefined operations. An atomic modification appears to the rest of the system instantaneously. Apart from this shared memory, processes can also have some local (private) memory.

Remarks:

- Various shared memory systems exist. A main difference is how they allow processes 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($R$): $t := R; R := 1; \text{return } t$
  - fetch-and-add($R, x$): $t := R; R := R + x; \text{return } t$
  - compare-and-swap($R, x, y$): if $R = x$ then $R := y; \text{return } \text{true}; \text{else return } \text{false};$ 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.

- The power of RMW operations can be measured with the so-called consensus-number: The consensus-number $k$ of a RMW operation defines whether one can solve consensus for $k$ processes. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processes, but not 3), 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 supporting weak RMW operations.

- 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 worry about exchanging messages correctly. 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: On one side you have the processes (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).

5.2 Mutual Exclusion

A classic problem in shared memory systems is mutual exclusion. 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 5.2 (Mutual Exclusion). We are given a number of processes, each executing the following code sections:
5.2. MUTUAL EXCLUSION 49

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

Sometimes we in addition ask for

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

Using RMW primitives one can build mutual exclusion algorithms quite easily. Algorithm 22 shows an example with the test-and-set primitive.

Algorithm 22 Mutual Exclusion: Test-and-Set

<table>
<thead>
<tr>
<th>Algorithm 22 Mutual Exclusion: Test-and-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Shared register ( R := 0 )</td>
</tr>
<tr>
<td>(&lt;\text{Entry}&gt;):</td>
</tr>
<tr>
<td>1. repeat</td>
</tr>
<tr>
<td>2. ( r := \text{test-and-set}(R) )</td>
</tr>
<tr>
<td>3. until ( r = 0 )</td>
</tr>
<tr>
<td>(&lt;\text{Critical Section}&gt;):</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>(&lt;\text{Exit}&gt;):</td>
</tr>
<tr>
<td>5. ( R := 0 )</td>
</tr>
<tr>
<td>(&lt;\text{Remaining Code}&gt;):</td>
</tr>
<tr>
<td>6.</td>
</tr>
</tbody>
</table>

**Theorem 5.3.** Algorithm 22 solves the mutual exclusion problem as in Definition 5.2.

**Proof.** Mutual exclusion follows directly from the test-and-set definition. Initially \( R = 0 \). Let \( p_i \) be the \( i^{th} \) process 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 \( t'_{i} \), process \( p_i \) resets the shared register \( R \) to 0. Between \( t_i \) and \( t'_{i} \), no other process can successfully test-and-set, hence no other process can enter the critical section concurrently.

Proving no deadlock works similar. One of the processes 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.

Remarks:

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

Algorithm 22 can be adapted to guarantee fairness (no lockout), essentially by ordering the processes 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 processes \( p_0 \) and \( p_1 \), only. In the remarks we discuss how it can be extended. The general idea is that process \( p_i \) 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 process is not interested \((W_0 := 0)\) 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 \( II \). See Algorithm 23.

Algorithm 23 Mutual Exclusion: Peterson’s Algorithm

**Initialization**: Shared registers \( W_i, W_\bar{i}, \Pi \) all initially 0.

**Code for process \( p_i \) \( i = \{0, 1\} \)**

<table>
<thead>
<tr>
<th>Algorithm 23 Mutual Exclusion: Peterson’s Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{Entry}&gt;):</td>
</tr>
<tr>
<td>1.</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3.</td>
</tr>
<tr>
<td>(&lt;\text{Critical Section}&gt;):</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>(&lt;\text{Exit}&gt;):</td>
</tr>
<tr>
<td>5.</td>
</tr>
<tr>
<td>(&lt;\text{Remaining Code}&gt;):</td>
</tr>
<tr>
<td>6.</td>
</tr>
</tbody>
</table>

Remarks:

- Note that line 3 in Algorithm 23 represents a “spinlock” or “busy-wait”, similarly to the lines 1-3 in Algorithm 22.

**Theorem 5.4.** Algorithm 23 solves the mutual exclusion problem as in Definition 5.2.

**Proof.** The shared variable \( II \) elegantly grants priority to the process that passes line 2 first. If both processes are competing, only process \( p_i \) can access the critical section because of \( II \). The other process \( p_{1-i} \) cannot access the critical section because \( W_{1-i} = 1 \) and \( ii \). The only other reason to access the critical section is because the other process is in the remainder code (that is, not interested). This proves mutual exclusion!

No deadlock comes directly with \( II \). Process \( p_1 \) gets direct access to the critical section, no matter what the other process does.
Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit. Thanks to the shared variable II also no lockout (fairness) is achieved: If a process \( p_i \) loses against its competitor \( p_{i-1} \) in line 2, it will have to wait until the competitor resets \( W_{i-1} := 0 \) in the exit section. If process \( p_i \) is unlucky it will not check \( W_{i-1} = 0 \) early enough before process \( p_{i-1} \) sets \( W_{i-1} := 1 \) again in line 1. However, as soon as \( p_{i-1} \) hits line 2, process \( p_i \) gets the priority due to II, and can enter the critical section.

**Remarks:**
- Extending Peterson's Algorithm to more than 2 processes can be done by a tournament tree, like in tennis. With \( n \) processes every process needs to win \( \log n \) matches before it can enter the critical section. More precisely, each process 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 II at each node of the binary tree, we inherit all the properties of Definition 5.2.

5.3 Store & Collect

5.3.1 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 \( n \) processes \( p_1, \ldots, p_n \). Every process \( p_i \) has a read/write register \( R_i \) in the shared memory where it can store some information that is destined for the other processes. Further, there is an operation by which a process can collect (i.e., read) the values of all the processes that stored some value in their register.

We say that an operation \( op_2 \) precedes an operation \( op_1 \) iff \( op_1 \) terminates before \( op_2 \) starts. An operation \( op_2 \) follows an operation \( op_1 \) iff \( op_2 \) precedes \( op_1 \).

**Definition 5.5 (Collect).** There are two operations: A \texttt{store(val)} by process \( p_i \) sets \( \mathbb{R}_i \) to be the latest value of its register \( R_i \). A collect operation returns a view, a partial function \( V \) from the set of processes to a set of values, where \( V(p_i) \) is the latest value stored by \( p_i \) for each process \( p_i \). For a collect operation \( \text{cop} \), the following validity properties must hold for every process \( p_i \):

- If \( V(p_i) = \bot \), then no store operation by \( p_i \) precedes \( \text{cop} \).
- If \( V(p_i) = v \neq \bot \), then \( v \) is the value of a store operation \( \text{sof} \) of \( p_i \) that does not follow \( \text{cop} \), and there is no store operation by \( p_i \) that follows \( \text{sof} \) and precedes \( \text{cop} \).

Hence, a collect operation \( \text{cop} \) should not read from the future or miss a preceding store operation \( \text{sof} \).

We assume that the read/write register \( R_i \) of every process \( p_i \) 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 24.

Algorithm 24 Collect: Simple (Non-Adaptive) Solution

\begin{algorithm}
\caption{Collect (by process \( p_i \))}
\begin{algorithmic}[1]
  \State \( R_i := \text{val} \)
  \Procedure{collect}{}\For {\( i = 1 \) to \( n \)} \State \( V(p_i) := R_i \) \EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

**Remarks:**
- Algorithm 24 clearly works. The step complexity of every store operation is 1, the step complexity of a collect operation is \( n \).
- At first sight, the step complexities of Algorithm 24 seem optimal. Because there are \( n \) processes, 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 processes \( p_i \) 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 \( t \), \( k \leq n \) processes 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.

5.3.2 Splitters

Algorithm 25 Splitter Code

\begin{algorithm}
\caption{Splitter Code}
\begin{algorithmic}[1]
\State \text{Shared Registers}: \( X : \{\bot\} \cup \{1, \ldots, n\} \); \( Y : \text{boolean} \)
\State \text{Initialization}: \( X := \bot \); \( Y := \text{false} \)
\State \text{Splitter access by process \( p_i \)}:
\begin{algorithmic}
  \State 1. \( X := i \)
  \State 2. \text{if} \( Y \) then \State 3. \text{return} \text{right} \State 4. \text{else} \State 5. \( Y := \text{true} \)
  \State 6. \text{if} \( X = i \) then \State 7. \text{return} \text{stop} \State 8. \text{else} \State 9. \text{return} \text{left} \State 10. \text{end if} \State 11. \text{end if}
\end{algorithmic}
\end{algorithmic}
\end{algorithm}
To obtain adaptive collect algorithms, we need a synchronization primitive, called a splitter.

Definition 5.6 (Splitter). A splitter is a synchronization primitive with the following characteristic. A process entering a splitter exists with either \text{stop}, \text{left}, or \text{right}. If \( k \) processes enter a splitter, at most one process exists with \text{stop} and at most \( k-1 \) processes exist with \text{left} and \text{right}, respectively.

Hence, it is guaranteed that if a single process enters the splitter, then it obtains \text{stop}, and if two or more processes enter the splitter, then there is at most one process obtaining \text{stop} and there are two processes that obtain different values (i.e., either there is exactly one \text{stop} or there is at least one \text{left} and at least one \text{right}). For an illustration, see Figure 5.1. The code implementing a splitter is given by Algorithm 25.

Lemma 5.7. Algorithm 25 correctly implements a splitter.

Proof. Assume that \( k \) processes enter the splitter. Because the first process that checks whether \( Y = \text{true} \) in line 2 will find that \( Y = \text{false} \), not all processes return \text{right}. Next, assume that \( i \) is the last process that sets \( X := i \). If \( i \) does not return \text{right}, it will find \( X = i \) in line 6 and therefore return \text{stop}. Hence, there is always a process that does not return \text{left}. It remains to show that at most 1 process returns \text{stop}. For the sake of contradiction, assume \( p_i \) and \( p_j \) are two processes that return \text{stop} and assume that \( p_i \) sets \( X := i \) before \( p_j \) sets \( X := j \). Both processes need to check whether \( Y = \text{true} \) before one of them sets \( Y = \text{true} \). 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, \( X \neq i \) (\( p_i \) and maybe some other processes have overwritten \( X \) by then). Therefore, \( p_j \) does not return \text{stop} and we get a contradiction to the assumption that both \( p_i \) and \( p_j \) return \text{stop}.

5.3.3 Binary Splitter Tree

Assume that we are given \( 2^k - 1 \) splitters and that for every splitter \( S \), there is an additional shared variable \( Z_S : \{\bot\} \cup \{1, \ldots, n\} \) that is initialized to \( \bot \) and an additional shared variable \( M_S : \text{boolean} \) that is initialized to \( \text{false} \). We call a splitter \( S \) marked if \( M_S = \text{true} \). The \( 2^k - 1 \) 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 \text{store} and \text{collect} operations are given by Algorithm 26.

Algorithm 26 Adaptive Collect: Binary Tree Algorithm

Operation\text{store}(val) (by process \( p_i \)):
1. \( R_i := \text{val} \)
2. if first \text{store} operation by \( p_i \) then
3. \( v := \text{root node of binary tree} \)
4. \( \alpha := \text{result of entering splitter } S(v) \)
5. \( M_S(v) := \text{true} \)
6. while \( \alpha \neq \text{stop} \) do
7. \( \text{if } \alpha = \text{left} \) then
8. \( v := \text{left child of } v \)
9. \( \text{else} \)
10. \( v := \text{right child of } v \)
11. \( \text{end if} \)
12. \( \alpha := \text{result of entering splitter } S(v) \)
13. \( M_S(v) := \text{true} \)
14. \( \text{end while} \)
15. \( Z_S(v) := i \)
16. \( \text{end if} \)

Operation\text{collect}:

 Traverse marked part of binary tree:
17. for all marked splitters \( S \) do
18. \( i := Z_S; V(p_i) := R_i \)
19. \( // \text{read value of process } p_i \)
20. \( \text{end if} \)
21. \( // V(p_i) = \bot \text{ for all other processes} \)

Theorem 5.8. Algorithm 26 correctly implements \text{store} and \text{collect}. Let \( k \) be the number of participating processes. The step complexity of the first \text{store} of a process \( p_i \) is \( O(k) \), the step complexity of every additional \text{store} of \( p_i \) is \( O(1) \), and the step complexity of \text{collect} is \( O(k) \).

Proof. Because at most one process can stop at a splitter, it is sufficient to show that every process stops at some splitter at depth at most \( k - i \leq n - 1 \) when invoking the first \text{store} operation to prove correctness. We prove that at most \( k - i \) processes 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 processes 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 \( k - i \) processes entering the splitter at the root of a depth \( i \) subtree, at most \( k - i - 1 \) obtain \text{left} and \text{right}, respectively. Hence, at the latest when reaching depth \( k - 1 \), a process is the only process entering a splitter and thus obtains \text{stop}. It thus also follows that the step complexity of the first invocation of \text{store} is \( O(k) \).

To show that the step complexity of \text{collect} is \( O(k) \), 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,
5.3. STORE & COLLECT

Figure 5.2: 5 × 5 Splitter Matrix

where \( k \) processes access the root. We claim that \( x_k \leq 2k - 1 \), which is true for \( k = 1 \) because a single process entering a splitter will always compute \textbf{stop}.

Now assume the inequality holds for \( 1, \ldots, k - 1 \). Not all \( k \) processes may exit the splitter with \textbf{left} (or \textbf{right}), i.e., \( k_i \leq k - 1 \) processes will turn left and \( k_i \leq \min\{k - k_i, k - 1\} \) turn right. The left and right children of the root are the roots of their subtrees, hence the induction hypothesis yields

\[
x_k = x_{k_1} + x_{k_2} + 1 \leq (2k_1 - 1) + (2k_2 - 1) + 1 \leq 2k - 1,
\]

concluding induction and proof.

Remarks:

- The step complexities of Algorithm 26 are very good. Clearly, the step complexity of the \textit{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 26 therefore is exponential in \( n \).

5.3.4 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 5.2. The algorithm is analogous to Algorithm 26. The matrix is entered at the top left. If a process receives \textbf{left}, it next visits the splitter in the next row of the same column. If a process receives \textbf{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 \textit{store} and \textit{collect}.

Theorem 5.9. Let \( k \) be the number of participating processes. The step complexity of the \textit{first store} of a process \( p_i \) is \( O(k) \), the step complexity of every additional \textit{store} of \( p_i \) is \( O(1) \), and the step complexity of \textit{collect} is \( O(k^2) \).

Proof. Let the top row be row 0 and the leftmost column be column 0. Let \( x_i \) be the number of processes 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 process visits the splitter in row \( i - 1 \) and column \( j \). By the properties of splitters, not all processes entering the splitter in row \( i - 1 \) and column \( j \) obtain left. Therefore, not all processes entering a splitter in row \( i - 1 \) move on to row \( i \). Because at least one process stays in every row, we get that \( x_i \leq k - i \). Similarly, the number of processes entering column \( j \) is at most \( k - j \). Hence, every process 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 \textit{collect} is at most \( O(k^2) \). Because the longest path in the splitter matrix is \( 2k \), the step complexity of \textit{store} is \( O(k) \).

Remarks:

- With a slightly more complicated argument, it is possible to show that the number of processes 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 \textbf{left} or \textbf{right} 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.

Chapter 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. For this work, Maurice Herlihy was awarded the Dijkstra Prize in Distributed Computing in 2003. Peterson’s Algorithm is due to [PET77, PET81], and adaptive collect was studied in the sequence of papers [MA95, AF02, AL05, AKP06].

Bibliography

[AKP+06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, 
and Roger Wattenhofer. Efficient adaptive collect using randomi-
a

[AL05] Yehuda Afek and Yaron De Levie. Space and Step Complexity Effi-

[Di65] Edsger W. Dijkstra. Solution of a problem in concurrent pro-


[MA95] Mark Moir and James H. Anderson. Wait-Free Algorithms for Fast, 

[Pet81] J.L. Peterson. Myths About the Mutual Exclusion Problem. Infor-

[PF77] G.L. Peterson and M.J. Fischer. Economical solutions for the crit-
ical section problem in a distributed system. In Proceedings of the