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 $R := y$; 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.

• 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. This insight had practical impact, as 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:

$\langle \text{Entry} \rangle \to \langle \text{Critical Section} \rangle \to \langle \text{Exit} \rangle \to \langle \text{Remaining Code} \rangle$

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 5.3 shows an example with the test-and-set primitive.

Algorithm 5.3 Mutual Exclusion: Test-and-Set

Input: Shared register \( R \) := 0

\begin{verbatim}
<Entry>
1. repeat
2. \( r := \text{test-and-set}(R) \)
3. until \( r = 0 \)
<Critical Section>
4. 
<Exit>
5. \( R := 0 \)
<Remainder Code>
6. ...
\end{verbatim}

Theorem 5.4. Algorithm 5.3 solves the mutual exclusion problem as in Definition 5.2.

Proof. Mutual exclusion follows directly from the test-and-set definition. Initially \( R \) is 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 5.3 can be adapted to guarantee fairness (no lockout), essentially by ordering the processes in the entry section in a queue.

Algorithm 5.5 Mutual Exclusion: Peterson’s Algorithm

Initialization: Shared registers \( W_0, W_1, II \), all initially 0.

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

\begin{verbatim}
<Entry>
1. \( W_i := 1 \)
2. \( II := 1 - i \)
3. repeat until \( II = i \) or \( W_1, i = 0 \)
<Critical Section>
4. ...
<Exit>
5. \( W_i := 0 \)
<Remainder Code>
6. ...
\end{verbatim}

Remarks:
• Note that line 3 in Algorithm 5.5 represents a “spinlock” or “busy-wait”, similarly to the lines 1-3 in Algorithm 5.3.

Theorem 5.6. Algorithm 5.5 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 \( p_0 \) 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 \( II \neq 1 - II \)). 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 \( p_{1-i} \) in line 2, it will have to wait until the competitor resets \( W_{1-i} := 0 \) in the exit section. If process \( p_i \) is unlucky it will not check \( W_{1-i} = 0 \) early enough before process \( p_{1-i} \) sets \( W_{1-i} := 1 \) again in line 1. However, as soon as \( p_{1-i} \) hits line 2, process \( p_i \) gets the priority due to \( II \), and can enter the critical section.
5.3 STORE & COLLECT

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 \Pi 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 \( \text{op1} \) precedes an operation \( \text{op2} \) iff \( \text{op1} \) terminates before \( \text{op2} \) starts. An operation \( \text{op2} \) follows an operation \( \text{op1} \) iff \( \text{op1} \) precedes \( \text{op2} \).

Definition 5.7 (Collect). There are two operations: A \( \text{store(val)} \) by process \( p_i \) sets \( \text{val} \) to be the latest value of its register \( R_i \). A \( \text{collect} \) 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 \( \text{collect} \) operation \( \text{cop} \), the following validity properties must hold for every process \( p_i \):

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

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

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 \( \text{op} \) to be the number of accesses to registers in the shared memory. There is a trivial solution to the \( \text{collect} \) problem as shown by Algorithm 5.8.

Remarks:
• Algorithm 5.8 clearly works. The step complexity of every \( \text{store} \) operation is 1, the step complexity of a \( \text{collect} \) operation is \( n \).
• At first sight, the step complexities of Algorithm 5.8 seem optimal. Because there are \( n \) processes, there clearly are cases in which a \( \text{collect} \) operation needs to read all \( n \) registers. However, there are also scenarios in which the step complexity of the \( \text{collect} \) operation seems very costly. Assume that there are only two processes \( p_1 \) and \( p_2 \) that have stored a value in their registers \( R_1 \) and \( R_2 \). In this case, a \( \text{collect} \) in principle only needs to read the registers \( R_1 \) and \( R_2 \) 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 \( \text{op} \) at time \( t \) adaptive to contention if the step complexity of \( \text{op} \) only depends on \( k \) and is independent of \( n \).
• In the following, we will see how to implement adaptive versions of \( \text{store} \) and \( \text{collect} \).

5.3.2 Splitters

Algorithm 5.9 Splitter Code

Shared Registers: \( X : \{\bot\} \cup \{1, \ldots, n\} \); \( Y : \text{boolean} \)
Initialization: \( X := \bot \); \( Y := \text{false} \)

Splitter access by process \( p_i \):
1. \( X := i \)
2. if \( Y \) then
3. \( \text{return right} \)
4. else
5. \( Y := \text{true} \)
6. if \( X = i \) then
7. \( \text{return stop} \)
8. else
9. \( \text{return left} \)
10. end if
11. end if

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

Definition 5.11 (Splitter). A splitter is a synchronization primitive with the following characteristic. A process entering a splitter exits with either \( \text{stop} \), \( \text{left} \), or \( \text{right} \). If \( k \) processes enter a splitter, at most one process exits with \( \text{stop} \) and at most \( k - 1 \) processes exit with \( \text{left} \) and \( \text{right} \), respectively.
Algorithm 5.13 Adaptive Collect: Binary Tree Algorithm

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

Operation collect:
Traverse marked part of binary tree:
17. for all marked splitters $S$ do
18. if $Z_S \neq \bot$ then
19. $i := Z_S$; $V(p_i) := R_i$ \hspace{1cm} // read value of process $p_i$
20. end if
21. end for \hspace{1cm} // $V(p_i) = \bot$ for all other processes

invoking the first 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 left and right, respectively. Hence, at the latest when reaching depth $k - 1$, a process is the only process 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 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 $k$ 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 stop. Now assume the inequality holds for $1, \ldots, k - 1$. Not all $k$ processes may exit the splitter with left (or 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 5.13 are very good. Clearly, the step complexity of the collect operation is asymptotically optimal.

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 5.15. The algorithm is analogous to Algorithm 5.13. The matrix is entered at the top left. If a process receives left, it next visits the splitter in the next row of the same column. If a process 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 5.16.** Let \( k \) be the number of participating processes. The step complexity of the first store of a process \( p_i \) is \( O(k) \), the step complexity of every additional store of \( p_i \) is \( O(1) \), and the step complexity of collect is \( O(n^2) \).


