4 Synchronization and Communication Primitives

Interaction between concurrently executing fragments is an essential characteristic of parallel programs and the major source of difference between sequential programming and parallel programming. Synchronization and communication are the two ways in which fragments directly interact, and these are the subjects of this chapter. We begin with a brief review of basic operating system concepts, particularly in the context of parallel and concurrent execution. If you already have a good knowledge of operating systems concepts, browse lightly or skip ahead.

Question: Who controls the executing fragments? How do different executing fragments interact and impact each other's execution?

4.1 *Threads and Processes*

Computing systems are managed by a program: an operating system. *Process* is the mechanism that operating systems use to start and control the execution of other programs. A process provides one or more ranges of addresses for the executing program to use. Each address has a value (which remains undefined until it is defined). Each range is mapped to a block of memory (which may be associated with an attached device). These blocks of memory are under direct management of the operating system. A range of addresses and the locations that they map to are collectively called an *address space*. An address space is divided into fixed-size units called *pages*. Address space and pages provide a logical or a *virtual* view of the memory. This view is also called *virtual memory*. The operating system maintains a mapping between pages and their locations in the real memory. One advantage of virtual memory is that not all pages need to be resident in the memory – some may be relegated to slower storage (not unlike the cache strategy), while others that remain undefined need not be mapped to any storage at all.

Being an executing program, the operating system comprises a set of processes, which start and schedule other processes. For example, an application starts with some running process launching a new process to execute that application's code. These processes may execute concurrently, sharing the available hardware by turn. An executing process may be forced to turn over to a waiting process via a mechanism of hardware interrupts. Some types of interrupts may terminate the process altogether. Others allow it to await a new turn. Processes may also directly request termination.

A process may create or *fork* child processes, with each executing its own instructions. The parent may share none, some, or all of its address space with the child.[1] Additionally, at the time of the fork, a copy of the parent's address space may be created for the child. The child then owns the copy, which is hidden from the parent. Creating such copies is time-consuming. Hence non-copy variants are sometimes called light-weight processes.

[1] There are page-mapping mechanisms for unrelated processes to also share address space with each other. We will not discuss their detais.

There is thus a spectrum of relationships between processes, but we will use this broad distinction: each process has its own address space, *threads* within a process all share that address space. A process comprises one or more threads that share that process's address space. We say that a process that only executes sequentially is a single thread, and its code shares the address space with no other thread. In this sense, a single-threaded process may be conveniently called a thread. Hence, we will commonly use the term thread when referring to a sequential execution. In other words, an executing fragment is a part of some thread's execution. Two threads from different processes do not share an address space, but through page-mapping mechanisms, they may yet be able to share memory.

There are intricacies we will not delve into. For example, the execution of *kernel* threads are scheduled directly and separately by the operating system, whereas *user* threads may be scheduled as a single unit, leaving them to coordinate each other's scheduling among themselves. Be aware though that different operating systems may differ slightly in their use of these terminologies. Generally, a programming platform, which includes the hardware, the operating systems, compilers, and any middleware, determines the schedulable unit of program. It tries to schedule as many units at a time as the number of cores available for execution. Different execution engines within a core can then be used in parallel only when the unit's code explicitly has multiple threads, or one thread is implicitly parallelized by the system architecture. In this book, we will assume that a thread is schedulable program unit. (When discussing cases where a set of threads are scheduled together, e.g., in GPUs, we will make the distinction clearer.)

There exist simple interfaces to start processes by providing, say, an executable file to the operating system. Already executing processes can, in turn, start other processes, sometimes remotely at another operating system. Such remote start requires communicating with a running process at that operating system. We will discuss these later in Chapter 6.

4.2 Race-condition and Consistency of State

Using terminology introduced in the previous section, threads can share addresses or variables with other threads, and hence may read values written by other threads. Each thread executes its sequence of instructions at its own pace. Recall that in a parallel system, no universal clock may be available. We will assume that there is a universal time that continually increases, but threads may not have any way to know this universal time at any instant. Rather, threads have their own local clocks, possibly ticking at a rate different from other threads' clocks. Even within a thread, there may be an arbitrary lag between its two consecutive instructions (e.g., if the execution is interrupted after the execution of the first). Thus, events occurring in concurrent threads at independent times impact the shared state and progress of a parallel program.

The order in which these events occur is non-deterministic.[2] Consequently, the behavior of the program may be non-deterministic. The program must always produce the expected result even in the presence of such non-determinism. If this non-determinism can lead to incorrect results, we call this *race-condition*. A race condition happens when the relative order of events impacts correctness.[3] Here is a simple example of a race condition:

[2] ***Defined :*** Non-determinism implies not knowing in advance. For example, non-deterministic order of two events means the order in which they occur changes unpredictably from execution to execution.

[3] Recall from Chapter 1 that events are not necessarily instantaneous and the order may not even be well defined. More generally, we say the relative timing of two events impact correctness

Listing 4.1: Race Condition

```
counter$ = counter$ + 1;
```

Suppose multiple threads execute this code, each incrementing the shared variable `counter$`. We will suffix $ to a variable name to highlight that it is shared by multiple threads. (As an aside, even though we emblematically use shared-memory terminology for ease of explanation, the discussion generally applies to any shared resource or state including those accessed through message-passing.) There are three parts to this instruction:

1. Fetch the value of shared resource `counter$`
2. Add one to the value
3. Send the result to shared resource `counter$`

We assume here that the intent of the variable `counter$` is to maintain the number of increments by all threads. But as different threads' steps occur in a non-deterministic order, counting may suffer errors. For example, if step 1 for thread i occurs between steps 1 and 2 of another thread j, both threads would increment the same value and write the same value. One of the two increments is lost. Race conditions can occur due to non-deterministic order of fetch and update of shared location, or other events. For example, two threads sending a message to a third thread could also race each other and lead to non-deterministic behavior, and possibly error.

In such cases, enforcing a relationship among certain parts of the code execution may solve the problem. For example, thread i may be prevented from starting this sequence during the interval any other thread j executes the three-step sequence. This eliminates the overlap and the race condition. The threads' relative order no longer impacts the correctness. In other words, in the middle of thread i incrementing `count$` no other thread accesses[4] it. Thus, if thread j follows thread i, it necessarily sees the value stored by thread i, and vice versa. Any non-determinism in the order in which threads i and j execute their three steps does not impact the final result. In general, a thread may be allowed to cause temporarily inconsistent or transient state within its view, but no other thread should be privy to that view, meaning they should not be able to see or operate on that inconsistent state. Let us understand the broader notion of consistency next.

[4] Access refers to a fetch or store of value at an address

Sequential Consistency

Recall that a CPU core may execute several instructions of a code fragment in parallel, but it ensures that they appear to execute in the sequence in which they are presented – the *program order*. For example, if instructions numbered i and $i+1$ in some code fragment do not depend on each other (as inferred by the compiler/hardware logic), instruction $i+1$ may be completed before i is. On the other hand, if there is a dependency, even if parts of their execution do overlap, the results of the instruction are the same as they would be if instruction $i+1$ started only after instruction i completed. A way to reconcile parallel execution with strict ordering is that execution of instructions may well overlap with others, but each 'takes effect' instantaneously and these instants are in an expected order. For example, in the following instructions:

Listing 4.2: Taking Effect

```
x = 5;
R1 = x;
```

the first instruction may take effect when the value 5 has appeared in all cache instances of address x. The second takes effect when the value 5 appears in Register R1. We will see that this notion of taking effect is too strict in the context of parallel programs, and may require unabated serialization to ensure that effects are instantaneous. Every access to a shared address may need to wait until all other accesses that have begun before it have taken effect. Furthermore, a global mechanism would be required to determine which ones began 'before' that access. Some controlled relaxation of the order could eliminate some serialization and yet be 'justifiable.' Let us discuss some examples.

We start by defining the notion of sequential equivalence, or sequential consistency in the shared state. Consider the following listing, assuming the values in A$ are 0 initially, and two threads with `threadID` 0 and 1, respectively, execute:

Listing 4.3: Sequential Expectation

```
A$[threadID] = 1
print A$[1-threadID]
```

With sequential reasoning, it would be natural to expect that at least one of the threads prints 1. After all, whichever thread writes into A$ 'later' expects the other thread's location in A$ to already be 1. Hence, its subsequent fetch of that value should yield 1. A parallel platform that fails to meet this expectation is not sequentially consistent. Let us formalize this idea.

We first assign a more practical meaning to taking effect. A fetch or read operation takes effect when it completes, meaning, *e.g.*, that a value at that address at some unknown time in the past arrives in a register ready for use in a subsequent operation. A store or write operation takes effect when a reader fetches the new value – we call that fetch the writer's *read-effect*. This definition allows different readers to record different read-effects of the same store operation, which could occur at different times and in different order from each other.

Concurrent execution of multiple threads is said to be *sequentially consistent*[5] if there exists a global sequence of all operations that access the shared addresses, which is consistent with the order of operation executed by each thread. The global sequence includes operations by every thread. The global sequence is consistent with thread i only if:

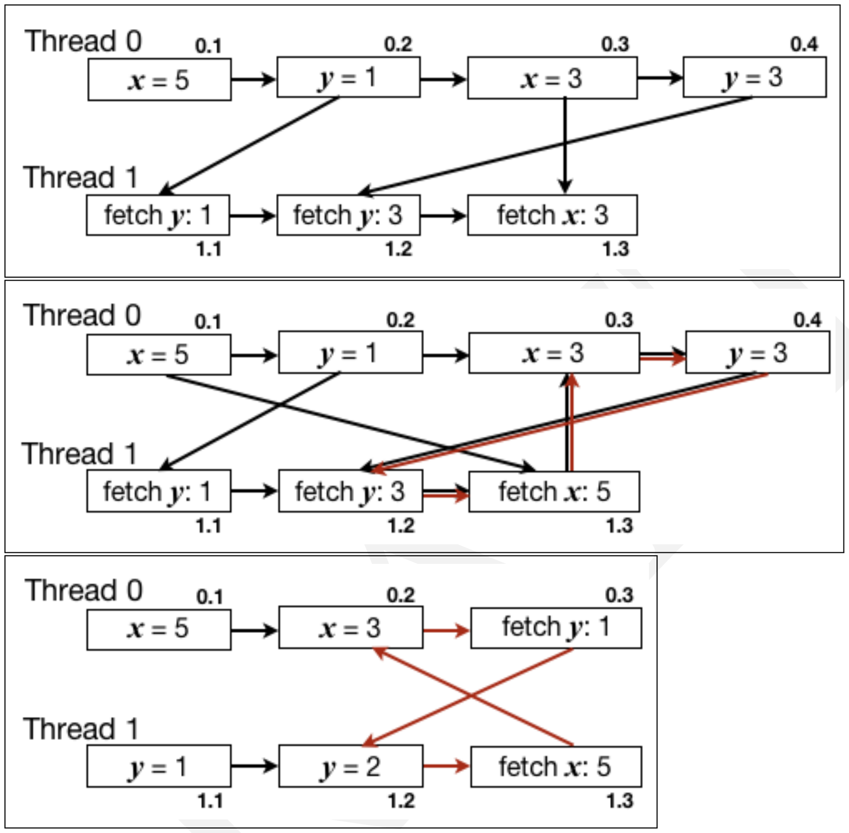
[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979. ISSN 0018-9340

1. If thread i executes operation o_1 before o_2, o_1 appears before o_2 also in the global sequence.

2. If operation o_i (Read from x) of thread i is the read effect of operation o_j (write to x) of thread j, o_i must appear after o_j in the

global sequence, and no other instance of (write to x) may appear between o_i and o_j in it.

This defines that a read of address x in every thread returns the value of the most recent write to x in that global sequence. In this sequence, we do not worry about the time when an operation takes effect but only their order. Operations of different threads are allowed to fully interleave. However, every thread's view of the order in which its operations take effect is consistent with every other thread's view. The following are examples of sequentially consistent and inconsistent executions.



(a) Sequentially consistent execution

(b) Sequentially inconsistent execution

(c) Sequentially inconsistent execution

Figure 4.1: Sequential consistency of Read/Write shared-memory variables

Figure 4.1(a) shows a sequentially consistent execution; (b) and (c) show two inconsistent executions. Note that the assignments imply write operations on the shared-memory locations. Values actually obtained by the fetch operations on the listed variables in the given execution are also shown after the colon. We refer to operation number j of thread i by the symbol $i.j$. The arrows show some of the orders that can explain the execution result. For example, 1.2 is the read-effect of 0.4 and must come after it (the value 3 was observed in *y* by 1.2 and written by 0.4).

There exist several global sequences of operations that the observed behavior is consistent with, *e.g.*, (0.1 $\rightarrow$ 0.2 $\rightarrow$ 0.3 $\rightarrow$ 1.1 $\rightarrow$ 0.4 $\rightarrow$ 1.2 $\rightarrow$ 1.3). Another sequence which has all operations of thread 0 in sequence followed by those of thread 1 is also consistent. An execution consistent with one such global sequence is sequentially consistent. If all executions of a program are guaranteed to be sequentially consistent, the program is said to be sequentially consistent. If all programs executable on a programming platform are sequentially consistent, the platform is sequentially consistent.

Figure 4.1(b) and (c) are both inconsistent executions because no consistent global order exists. This can be seen by following the arrows in red. These arrows form a cycle, meaning there is no way to order them in a sequence. For example, in Figure 4.1(b) 1.3 occurred before 0.3; otherwise, it would have read the value 3 in *x*. Of course, 0.3 always must occur before 0.4, which occurred before 1.2 in this execution, because 1.2 is the read-effect of 0.4. This could occur in an execution if the update to variable *y* becomes quickly visible to other threads, while updates to *x* travel slower.

This variance in the update speed can also occur due to caches in case of shared-memory – even if the caches are coherent. Updates indeed reach all threads, just not in the same order. Keeping all updates in order implies slower updates hinder the faster ones. Moreover, in-order updates do not guarantee sequential consistency, as Figure 4.1(c) shows. Each thread updates different variables – *x* and *y*, respectively. So, there is no inherent order between 0.2 and 1.2. Still, there exists a cycle as red arrows show, and hence no consistent sequential order. In this case, both updates are just slow. Should we simply block the execution until a *write* becomes visible to all potential readers? Or, is the lack of sequential consistency inconsequential?

There certainly are real-world consequences, as seen in the following situation.

Listing 4.4: Single Producer Consumer

```
     Thread 0                    Thread 1
while(! ready$);            data$ = generate();
  x = data$;                ready$ = true;
```

Thread 1 produces data that thread 0 consumes. This is a simple instance of the *producer-consumer* problem, where one or more threads may produce a sequence of data and one or more threads consume them one at a time. In this example, a single piece of data is produced and consumed. Thread 1 sets `ready$` after ensuring that `data$` is indeed ready. We assume `ready$` is initially `false`. Accordingly, thread 0 continues checking the value of `ready$` until it becomes true. At this point, it reads `data$`. What if the execution is not sequentially consistent? Thread 0 could read stale data even after finding the updated value in `ready$`.

There are situations, however, when certain pairs of operations may be swapped in the global sequence. For example, if thread i reads x and then reads y, it is possible that the results are the same even of those two operations are in the reverse order in the global sequence. We next discuss a few common relaxations to the requirement of complete sequential equivalence. The general idea is to allow the platform to guarantee somewhat relaxed constraints, thus supporting higher performance. The programmer is then responsible for enforcing any other ordering constraints if required, using synchronization techniques discussed later in this chapter.

Causal Consistency

The idea of causal consistency is to limit the consistent ordering constraint only to what are called *causally* related operations. In particular, there is no requirement of a consistent global sequence of all operations to exist. Rather, each thread views the write operations of other threads in a causally consistent order, meaning two causally related operations are viewed by every thread in the order of their causality, which is defined as follows:

1. All writes of one thread are causally related after that thread's earlier reads and writes

2. A read is causally related after the write whose value it gets

3. Causality is transitive, *i.e.*, $op_a \rightarrow op_b$ and $op_b \rightarrow op_c$ imply $op_a \rightarrow op_c$.

In particular, two writes (from different threads) that are not causally related may be observed in different orders by different threads: they are truly concurrent. In a given thread's view, its own operations must always appear in its program order. Figure 4.1(c) shows the example of a causally consistent execution. In thread 0's view, only 1.1 needs to happen before 1.2. It sees no evidence to the contrary. For example, the order (0.1 $\rightarrow$ 0.2 $\rightarrow$ 1.1 $\rightarrow$ 0.3 $\rightarrow$ 1.2) is thread 0's causally consistent view. It does not need to find a consistent place for 1.3 in this ordering. Similarly, the order (1.1 $\rightarrow$ 1.2 $\rightarrow$ 0.1 $\rightarrow$ 1.3 $\rightarrow$ 0.2) is thread 0's causally consistent view of thread 1. The example in Figure 4.1(b) remains causally inconsistent, however. In thread 1's view, all of thread 0's operations 0.4 happens before 1.2, but 0.3 happens after 1.3, when 0.3 is causally before 0.4.

FIFO and Processor Consistency

Guarantee of causal consistency requires a potentially complex evaluation of transitive relationships. On the other hand, further relaxation of certain order constraints provides more opportunity for performance optimization. FIFO[6] consistency only enforces a consistent order of writes operations of all threads. In other words, writes from a given thread are seen to be in that thread's order by every thread. Read operations and ensuing transitive causalities need not be consistently ordered. The example in figure 4.1(b) remains FIFO inconsistent, but the example in Figure 4.2(a) exhibits FIFO consistency, even though it violates causal consistency, and hence also sequential consistency. In this figure, the red arrows demonstrate transitive causality, which forces a relationship between otherwise concurrent writes 0.2 and 1.3. Note that 1.3 must occur before 0.1, as 0.1 is its read-effect, and similarly, 1.2 must occur after 0.2. We may assume in these examples that the initial value of variables is, say, 0. Figure 4.2(a) is a FIFO consistent execution because there is only a single write by thread 0, and it can be viewed anywhere before 1.2 in thread 2's view. The two writes to y in thread 1 must appear in that same order in thread 0's view. (1.1 $\rightarrow$ 1.3 $\rightarrow$ 0.1 $\rightarrow$ 0.2) is a FIFO consistent order.

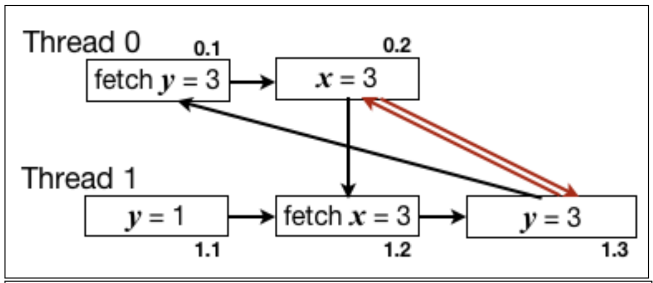
[6] FIFO consistency model is also known as PRAM consistency.

The notion of processor consistency is a slight tightening of constraints. In addition to a consistent ordering of all writes by a given thread, all threads must also view all writes to the same variable in the same order. FIFO and processor consistency are both weaker than causal consistency and allow the execution in Figure 4.2(a). Figure 4.2(b) shows an execution that is FIFO consistent but not processor consistent. It is FIFO consistent because writes to y by thread 1 (*i.e.*, 1.1 $\rightarrow$ 1.3) are consistent in thread 0's view: (1.1 $\rightarrow$ 1.3 $\rightarrow$ 0.1 $\rightarrow$ 0.2 $\rightarrow$ 0.3). Writes from thread 0 (*i.e.*, 0.2 $\rightarrow$ 0.3) remains consistent in thread 1's view: (0.2 $\rightarrow$ 1.1 $\rightarrow$ 0.3 $\rightarrow$ 1.2 $\rightarrow$ 1.3).

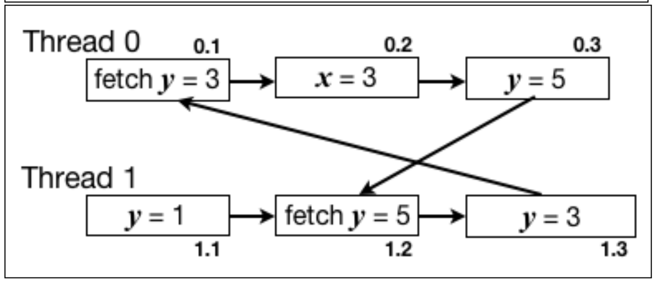
However, processor consistency additionally requires that all writes to y are seen in by the same order by all threads. This means that the two orders above, which switch 0.3 and 1.3 are not processor consistent. In fact, no consistent order exists for this execution because thread 0 must see 1.3 occur before 0.1 and hence before 0.3. On the other hand, thread 1 must see 0.3 occur before 1.2 and hence before 1.3.

Figure 4.1(c) is neither FIFO consistent nor processor consistent. Thread 1 must see all writes from thread 0 in order (0.1 $\rightarrow$ 0.2 $\rightarrow$ 0.3 $\rightarrow$ 0.4). However, it sees 0.4 before 1.2 but 0.3 after 1.3.

Note that a guarantee of FIFO consistency is sufficient to prove the correctness of Listing 4.4 in its every execution.



(a) Causally inconsistent execution



(b) Processor inconsistent execution

Figure 4.2: FIFO consistent executions

Weak consistency

Finally, there is a practical notion of consistency called weak consistency, under which minimal guarantees are made by the programming platform. The responsibility of maintaining consistency is instead left to the programmer. This follows the principle 'programmer knows best' and allows the system to make aggressive optimizations. A programmer in need of enforcing order between two operations then must employ special primitives; an example is *flush*. Another possibility is to enforce sequential or some other form of consistency only on specially designated variables or resources, called *synchronization variables*. Additionally, writes to non-synchronization shared variables are allowed to be reordered but only so long as there is no synchronization operation between them. In other words, synchronization operation must all be in sequentially consistent order, and non-synchronization operations between two synchronization operations must remain between those two in any reordering.

Flush is a synchronization operation. This means all memory operations in flight (*i.e.*, started but not completed) when flush is called must be completed before operations after the flush can begin. This operation is also called *memory fence* – fences that memory operations cannot cross in any re-ordering. The example in Listing 4.4 will need to be re-written thus to ensure correctness if even FIFO consistency is not supported by the programming platform.

Listing 4.5: Single Producer Consumer with Weak consistency

```
     Thread 0                      Thread 1
while(! ready$)              data$ = generate();
```

```
  memory_fence();          memory_fence();
x = data$;                 ready$ = true;
```

Fences slow down memory operations and the code above may be an overkill but it guarantees correctness even if caches are not coherent. A fence ensures that caches are flushed and an updated value of `ready$` is indeed fetched by thread 0. Further, the memory fence of thread 1 ensures that the data read by thread 0 is indeed the updated data written by thread 1.

Linearizability

There are several other flavors of the notion of consistency. Linearizability is stronger than sequential consistency. It guarantees not only that all operations have a global order consistent with all threads' execution, but also that each operation completes within a known time interval. In particular, the operation is supposed to take global effect at some specific instant between the invocation and completion of each operation by its thread. It thus requires the notion of a real-time central clock and requries arguments about an operation having completed before a certain real time t. As one consequence, if an execution is linearizable with respect to each variable, the overall execution also becomes linearizable. (We will not prove this statement here.) Sequentially consistent sub-executions are not able to be composed in this manner to produce a longer sequentially consistent execution.

There is a related notion of serializability, mostly used in the context of databases. It's an ordering constraint on *transactions*. A transaction is a set of operations on a set of shared resources. Serializability guarantees that the transactions appear atomic. In other words, there is a sequential order of transactions that produce the same result. Sequential order there means that transactions are performed one at a time without any interleaving.

Now we turn our attention to the general notion of synchronization.

4.3 *Synchronization*

It is not necessary to synchronize individual thread clocks to the universal time. Indeed perfect synchronization is impossible, given that signals may travel only at a finite speed and there may be variable delays. There is a distance even between the clock generator and the execution engines.

However, as discussed in the previous section, programs only care about the order of events. A read-effect is so, whether the write completed immediately before the read or somewhat earlier. Hence, we focus on enforcing consistency using synchronization to impose order between selected events of two or more threads. We use two types of synchronization: *exclusion* and *inclusion*. Exclusion synchronization precludes mutual execution of two or more threads – rather two or more specific events within those threads. (An event is a sequence of execution steps.) Inclusion synchronization, on the other hand, ensures co-occurrence of events. We will now examine a few important synchronization concepts and tools. Later, we will see some examples.

Synchronization Condition

With some support from the hardware, operating systems provide several basic synchronization primitives. However, their context is only the system controlled by the operating system. If multiple operating systems are involved, additional primitives must be built, possibly using these basic primitives. In any such primitive, once the execution of a thread encounters a *synchronization event*, it requires certain conditions to be satisfied before it may proceed further. Other fragment executions may impact those conditions. There are two types of conditions: *shared* conditions and *exclusive* conditions. Multiple threads waiting for a shared condition may all see it when the condition becomes satisfied, and therefore all continue their execution. Only one of the waiting threads may continue if the condition is exclusive. There are also hybrids, which allow a fixed number of waiting threads to continue. Usually, the choice of continuing threads is random, but it is also possible to allow continuation based on some priority.

Protocol Control

There is usually a coordination protocol involving multiple synchronization events a thread must follow before it can complete the synchronized *activity*. There are two classes of protocols: *centralized protocol* and *distributed protocol*. In centralized protocols, there is a coordinating entity, *e.g.*, another thread, an operating system, or some piece of hardware. This centralized controller flags a thread ahead or stops it, not unlike what a traffic signal does. In parallel computation involving a large number of synchronizing threads, such a centralized controller is often a source of bottleneck. Failure of the coordinator also can be disastrous for the entire program. In distributed protocols, there is no centralized controller. Rather the threads themselves follow a set of steps synchronizing each other.

This may involve the use of multiple passive shared resources, *e.g.*, memory locations.

As a simple example, concurrent operations on a queue (*e.g.*, insertion or removal) by multiple threads may be activities. Checking if a queue is full is a synchronization event. Checking if there are ongoing removals could be another event. A protocol is the set of events designed to ensure that multiple threads may safely add and remove elements without being misled by any transient variables (set by another thread).

Progress

Synchronization event is nothing but a sequence of instructions executed by a thread, often via a function call. There are two parts to this call: checking if the condition is satisfied and then waiting or (eventually) proceeding past the event, depending on the result. Atomicity is required for exclusive conditions because two different threads must not view, and both proceed on the same condition. This requires some coordination among competing threads, and even the test for the condition may itself be impacted by the state of a different thread. Still, it is possible to implement synchronization in a way that allows the test to safely complete independent of action by other threads. Of course, the synchronization protocol still applies, and actions to be taken when the condition is satisfied may still be taken only if the condition is satisfied. Such methods are called *non-blocking*. In particular, a non-blocking function completes in finite time even in the presence of indefinite delays, or failure, in other threads' execution.

This notion of non-blocking functions applies to contexts other than synchronization as well, *e.g.*, data communication or file IO. Although similar, this notion is slightly different from that of non-blocking network topology discussed in chapter 1. There, messages between one pair of nodes could progress without being blocked by messages between a different pair, *i.e.*, one message was not blocked by another as long as the communicators were separate.

A blocking function merely does not return until the synchronization condition is satisfied. The execution proceeds to the next instruction after this return, just as it would after every other function. A non-blocking function, on the other hand, returns even when the condition is not satisfied. The protocol must then account for such unsuccessful return and may, *e.g.*, choose to perform some other computation that does not require the condition to hold. We will see examples of such protocols later in this section.

Note that the notion of blocking is a thread-level progress criterion. There are also system-wide progress criteria. A synchronization protocol that guarantees that at least one thread (among all synchronizing threads) always completes its activity irrespective of other threads' behavior is called *lock-free*. In particular, if multiple threads can operate on a shared data structure such that some operation or the other continues to succeed, it is called a lock-free data structure. If every thread is guaranteed to complete its activity eventually, it is called *wait-free*. A wait-free data structure ensures that each operation eventually completes. We will study examples of lock-free and wait-free synchronization later in this chapter.

Separate from whether a function is non-blocking is the issue of how the condition-checking and progress are managed. In *busy-wait* based implementation, the fragment repetitively checks until the condition turns favorable. A busy-wait loop can result in blocking if the condition checking depends on action by other threads. The other alternative is *signal-wait*. The calling thread is suspended until the conditions become favorable again, after which an external entity like the operating system wakes the thread and makes it eligible for execution. The signal-wait mechanism is blocking by definition, as the thread can make no progress in the absence of action by the external entity.

When there are many more threads than the number of cores available to execute them, the busy-wait strategy can waste computing cycles in repetitively testing and failing – particularly if the synchronization event involves a large number of threads or long synchronized activities. On the other hand, the latency of such tests is usually much lower than that of signal-wait. In any case, synchronization overhead is not trivial. This overhead includes the time spent in the synchronization primitive as well as the time a thread waits for action by other threads. Hence frequent or fine-grained synchronization is not advisable in parallel programming. A well-designed parallel program tries to reduce synchronization in the first place. We will see that certain busy-wait strategies are suitable for parallel execution.

Synchronization Hazards

In any synchronization protocol, there are two hazards to guard against: *deadlock* and *starvation*.

Two or more thread deadlock if none of them can ever complete the synchronization activity because the condition for each remains permanently unfavorable. Each such unfavorable condition could only be turned favorable by one or more of the other deadlocked threads. But, they cannot, for they are themselves waiting, likely for some other condition. Effectively, they all indefinitely wait for each other. There is a famous abstraction called the dining philosopher's problem demonstrating deadlocks. A modified version goes like this.

Consider five philosophers sitting around a table with five forks alternately laid between them. Philosophers meditate and eat alternately, but they may eat only with two forks. After they eat, they clean both forks and put them back in their original setting. Each philosopher eats and meditates for arbitrarily long periods. Their eat-meditate lifecycle goes on indefinitely. No more than two philosophers may eat at the same time (maybe because the food cannot be supplied quickly enough).

Consider the following protocol. Philosophers pick any available fork on their left and then their right when hungry. If both are picked, they eat. Once full, they put the forks down one at a time and go back to meditating. If only one fork is available, they pick it up. If they do not have two, they meditate some more before checking again. They do so repeatedly until they get both forks. They then eat before replacing the forks.

This protocol ensures that no philosopher eats with only a single fork. It also ensures that only up to two philosophers can be eating at any given time. Note that a philosopher who is using two forks ensures that neither neighbor may have two forks. A synchronization protocol that guarantees the required behavior at all times, as this eating protocol does, is called *safe*. What happens in this protocol, however, if all philosophers pick up the forks to their left almost simultaneously, and then wait for the fork to their right to become free? Since no one got two, no one eats, and no fork is set down, no matter how many times they check to their right. This is deadlock. Note that if the philosophers could simultaneously pick two forks, the deadlock could be avoided. Sometimes, such complex atomic instructions may be available for synchronizations. At other times, only simple building blocks, like 'pick one fork' are available.

Starvation is the situation when a thread waiting for a condition to become favorable fails to find it so even if the condition does intermittently become favorable. It either fails to check in time before the condition turns unfavorable again (if it is busy-waiting), or it remains sleeping for a long time, even indefinitely, and some other thread is repeatedly woken up instead. Protocols that guarantee a lack of starvation are called *fair*. Notice that in the previous example, a dining philosopher could starve in the listed protocol, for there is no guarantee that they check during the period their neighbors left the fork down on the table. The neighbor is allowed to pick the fork back up.

Considering the traffic light example, the basic purpose of syn-

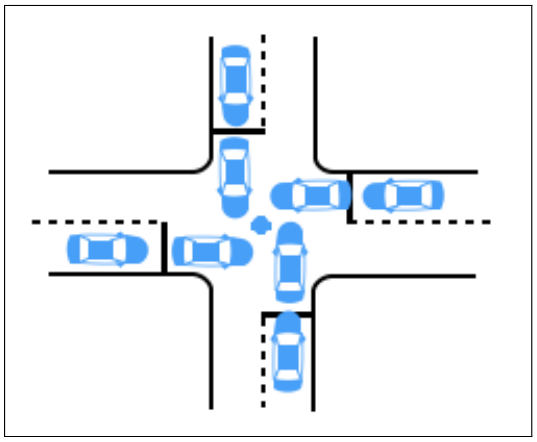


Figure 4.3: Traffic Light Deadlock

chronization is that no two vehicles may be in each other's path (following behind is allowed, reversing is not). A protocol like 'go only on green' is safe as long as the signals are properly coordinated. There may be a deadlock, however, if a slow vehicle that enters the intersection on green is not able to get through before the light turns green for the cross-traffic. See Figure 4.3 for a deadlocked configuration. One may modify the protocol to prevent these deadlocks. For example, vehicles could go on green only if there is no cross-vehicle in the intersection. Now, there would be no deadlock, but starvation is possible. Too many slow vehicles could ensure that a waiting vehicle continually sees a vehicle in the intersection while its light is green (and it turns red again before any progress is made). Furthermore, the vehicle throughput may reduce. This situation arises in many synchronization protocols, and simple solutions to prevent deadlocks often risk starvation.

4.4 *Mutual Exclusion*

We now introduce a few standard synchronization primitives. *Mutual exclusion* is one such widely used primitive. As the name suggests, it prevents mutual execution of two code fragments. We use the term *critical section* to refer to such a code fragment. Although any part of a thread's execution can be designated as critical, such usage is generally limited to parts of the execution that modify a shared resource or state, *e.g.*, a shared-memory location. If a thread is executing any part of its critical section, no other thread may be executing a competing critical section. Rather, critical sections are strictly ordered in their impact on the shared resource. This also means these impacts appear to be atomic to those competing threads.

We next discuss some synchronization methods that support mutual exclusion.

Lock

A simple synchronization tool is *lock*. Each lock has a name known to all participating threads. The simplest locks are exclusive. A thread is allowed two main operations on a lock: *acquire* (also called *lock*) and *release* (also called *unlock*).

Acquire(x): Attempt to acquire a lock named x and hold it. Acquisition succeeds if x is not already held by some other thread, *i.e.*, some thread has not already acquired it. Otherwise, the requesting thread waits until the current holder releases x. Acquire operation blocks until the acquisition is successful (although non-blocking variants exist also). If two concurrent threads attempt to acquire an available lock, only one succeeds (if the lock is exclusive).

Release(x): Allow subsequent acquisitions.

Non-exclusive locks are *counting locks*, which allow up to n holders at a time for some fixed value of n. Locks may also be *re-entrant*, meaning additional acquisition attempt by the holder is allowed. In such a situation, the definition of holder (*e.g.*, a thread or a process) must be explicit. In non re-entrant locks, the holder trying to re-acquire an exclusive lock would lead to a deadlock. The following example shows how to safely perform the counter increment operation described in section 1.4.

Listing 4.6: Lock based mutual exclusion

```
Acquire(counter_lock$)
counter$ = counter$ + 1
Release(counter_lock$)
```

If every thread updating counter$ follows this protocol, and updates to counter$ are seen consistently in all threads, counter$ is incremented safely and atomically. Consistent ordering is ensured if `Acquire` and `Release` are synchronization operations for the variable counter$. In fact, it is sufficient that these two are processor-consistent write operations.

Notice that deadlock is not possible in fragment 4.6. If thread i waits, that is only because some other thread j holds the lock. Thread j that holds the lock eventually executes the release, since there is no other synchronization it waits for. Of course thread i may yet starve as there is no guarantee that if the lock is released, it won't be repeatedly offered to another requester. In other situations, the holder may fail to release the lock. For example, thread j may crash. Algorithms to recover from such crashes are quite complex[7,8,9] and will not be discussed in this book.

[7] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1): 1–20, 1991

[8] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 111—-122, 2002

[9] E. N. (Mootaz) Elnozahy, L. Alvisi

It is sometimes possible to effect synchronization merely with the help of shared-memory locations. We assume that the shared-memory operations are sequentially consistent. A write operation completes at some instant: all threads see the old value before this instant and the new value after that. Two writes may not occur at the same instant.

We describe two algorithms next that ensure mutual exclusion using only shared-memory: Peterson's algorithm and Bakery's algorithm.

Peterson's Algorithm

Peterson's algorithm guarantees mutual exclusion between two threads. Both execute code 4.7, which may be executed any number of times by each thread. This method employs shared variables `ready$` (an array of size 2) and `defer$` to achieve exclusion. Assume the two threads are identified by IDs 0 and 1, respectively. The value of the ID is always found in an automatic private variable `threadID`. Private variables, even if they have the same name in each thread, are local to each thread and thus not shared. Initially, `ready$[0] = ready$[1] = false`.

Listing 4.7: Peterson's algorithm for two thread mutual exclusion

```
other_id = 1 - threadID;
ready$[threadID] = true;              // This thread wants in
defer$ = threadID;                    // This thread defers
while(ready$[other_id] && defer$ == threadID); // Busy-wait
// Critical Section goes here
ready$[threadID] = false;             // Not critical.  Not ready.
```

A thread indicates its intent to execute the critical section by setting its `ready$` flag. It then indicates its willingness to defer to the other thread. The order of these two operations is important. Of course, `defer$` is shared, and the other thread could overwrite `defer$`. However, each thread is willing to busy-wait in the while loop if `defer$` remains set to its ID, and the other thread has indicated it also wants to enter the critical section. A thread remains ready in the critical section and only turns not-ready after it completed its execution of the critical section.

Safety means that if a thread exits its loop and enters the critical section, the other thread is guaranteed to not enter until the first is out of the critical section.

Figure 4.4 demonstrates the possible order of operations on the shared memory. In the top row are operations by thread 0 and the bottom row has those by thread 1. By our assumption that each

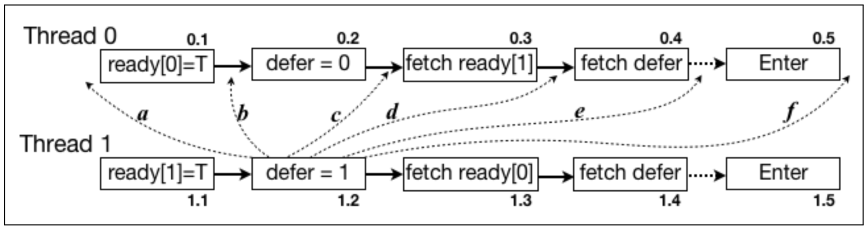


Figure 4.4: Operation order for Peterson's algorithm

operation is atomic, no two operations overlap. In particular, $i.j$ occurs before $i.j' \ \forall j' > j$. Solid arrows from $i.j$ to $i.j'$ help visualize this. (Recall that $i.j$ is the j^{th} operations of thread i.) Note that we do not, in general, have any pre-determined order between $i.j$ and $k.l$ if $i \neq k$. Accordingly in our example, an operation like 1.2 could occur between any $0.j$ and $0.(j+1)$. These possibilities are shown in dashed arrows and marked $a-f$. Regardless, 1.1 must occur before 1.2, which must occur before 1.3, and so on. No two operations on the same shared memory may occur simultaneously; otherwise, the value read or written would be undefined.

In a given execution, if possibilities a or b materialize, thread 0 must find `defer$ == 0` at 0.4. Similarly, if possibilities c or d occur, the value would be 1. Finally, if e or f materialize, thread 0 would still find value 0. Let's analyze each case.

Case a, b: Thread 0 is guaranteed to find `ready$[1]` to be true at 0.3 as 1.1 is guaranteed to occur before 1.2. Hence, thread 0 does not proceed to the critical section.

Case c, d In cases c and d, thread 0 finds `defer$ == 1`. Hence the loop ends, and 0.5 follows. Since 0.1 and 0.2 occur before 1.2, and hence before 1.3, thread 1 cannot get past its loop as it finds thread 0 ready and `defer$ == 1`.

Case e The behavior depends on 1.1. If 1.1 occurs before 0.3 (call this case e1), neither thread 0 nor thread 1 may exit their respective loops. Both find the other is ready and both find their own IDs in `defer$`. However, this cannot last. As both execute the next iteration of their busy-wait loops, thread 0 now finds 1 in `defer$` and enters the critical section. Thread 1 does not.

If, on the other hand, 1.1 occurs before 0.3 (case e2), thread 0 enters the critical section finding that thread 1 is not ready. This time thread 1 would not enter the critical section as it would find `defer$ == 1` at 1.4 and `ready$[0] == 1` at 1.3 as long as thread 0 remains in the critical section.

Case f In case f, 1.1 cannot occur before 0.3. If it did, thread 0 would not reach 0.5 as explained for case e. That's a contradiction.

Thus 1.1 may only occur before 0.3. This is similar to case *e*2.

In other words, thread 0 exits its busy-wait loop either because thread 1 was not ready, or thread 0's write to `defer$` had been overwritten by 1. If `defer$` was indeed found to be 1 by thread 0, thread 1's write to it would have happened *after* thread 0's. This means thread 1 is guaranteed to find 1 in `defer$` and wait at its loop as long as thread 0 remains ready.

On the other hand, if thread 0 exits the loop because `ready$[1]` is false when tested, if later thread 1 becomes ready and then sets `defer$` to 1, it is guaranteed to find `defer$` to be 1 in its loop condition. Thus, thread 1 cannot enter the critical section until thread 0 stops being ready, post its execution of the critical section.

The same argument holds for thread 1. Peterson's algorithm is also deadlock-free and starvation-free.

A deadlock could occur only if both threads are indefinitely stuck in their busy-wait loops. This would imply that thread 0 continually finds `defer$ == 0` and thread 1 continually finds `defer$ == 1`. Both cannot be true because neither thread has a chance to change `defer$` while busy-waiting.

No thread can starve either. Suppose without loss of generality that thread 0 does. This would imply that thread 1 is able to repeatedly complete its critical section, return for the next round and overtake the busy-waiting loop of thread 1. This means that each time thread 0 checks, `defer$ == 0` and `ready$[1]` is true. But only thread 0 may ever set `defer$` to 0, never thread 1. Rather, if thread 1 is able to repeatedly execute the protocol, it is obligated to set `defer$` to 1 each time. How then does then `defer$` become 0 without thread 0 re-setting it? And it couldn't if it is busy-waiting. That's a contradiction.

Peterson's algorithm is not wait-free. If a thread is indefinitely delayed in the critical section, the other thread would not be able to proceed. It is not even lock-free, since a delay in a thread while it is in a critical section, blocks the progress of all competing threads. Indeed, synchronization protocols like mutual exclusion or locks are not lock-free (even as that appears to be a tautology).

In large parallel systems, synchronization is seldom between only two threads. Lamport's Bakery algorithm addresses this problem.

Bakery algorithm

Bakery algorithm is based on a ticket system. When a thread becomes ready, it takes a 'number,' and await its turn. The pseudo-code is as follows:

Listing 4.8: Bakery algorithm for n thread mutual exclusion

```
1 ready$[threadID] = true;                    // This thread wants in
2 number$[threadID] = maximum(number$) + 1;   // Take a number
3 while(∃ id != threadID s.t. ready[id] && \  // busy wait
4   number$[id] < number$[threadID] || number$[id] == number$[threadID] && id < threadID);
5 // Critical Section goes here
6 ready$[threadID] = false;                   // Critical no more
```

Its structure is an extension to Peterson's. When ready, a thread takes a number – one more than the maximum number taken by any thread. It then busy-waits until no ready thread has a smaller number. Note that finding the new number itself is not protected by a critical section or synchronization. This means two (or more) threads may obtain the same number on line 2 of code 4.8, say n. When this occurs, ID is used to break the tie: the thread with lower ID exits loop first. If that winning thread later wants to enter the critical section again, the next time its number is guaranteed to be greater than n after line 2. Thus `number$[`i`]` strictly increases every time thread i completes line 2.

To prove safety, show that two threads may not be in a critical section at the same time. Suppose these are threads i and j with, say, $i < j$. When j exited its busy-wait loop on line 3, either `number$[`j`] < number$[`i`]` or `ready$[`i`] == false`.

If `ready$[`i`]` was `false`, thread i set `ready$[`i`]` to `true` later than thread j's condition test. This means thread i also computed its number after j's test and hence `number$[`i`] > number$[`j`]`. Hence, it could not have exited its loop, as `ready$[`j`]` remains `true` until after j completes the critical section.

If, instead, `ready$[`i`]` was `true`, `number$[`j`] < number$[`i`]` at j.3 (*i. e.*, at line 3 for thread j), meaning i.2 occurred after j.2. Since j is in the critical section, `ready$[j]` would be `true` at i.3 and thread i could not exit its busy-wait loop.

Bakery algorithm is also deadlock-free and starvation-free. Deadlocks are avoided because there exists a total order on updates to `number$` and some thread with the smallest number is always able to get past the busy-wait loop. At the same time, an increasing number ensures no thread is able to overtake one that got a number earlier. Such number based design is common in many wait-free protocols.

The drawback of Bakery algorithm is the need for large shared arrays (`ready$` and `number$`). It turns out that there exists no algorithm that can guarantee mutual exclusion with a smaller size using only shared-memory reads and write operations.

Compare and Swap

Synchronization as described above using only shared-memory reads and writes have limited utility. In particular, they have a low consensus number[10]. There exist more powerful primitives that have lower cost and greater generality. The most common one is called compare and swap. It atomically performs two shared-memory operations. It compares a given value to a shared-memory location and then operate on the shared-memory location depending on the result of the comparison. Here is an example function for an integer shared variable with address ref$:

[10] ***Defined :*** Consensus number indicates the number of threads that can achieve wait-free consensus. See the notion of consensus below.

Listing 4.9: Compare And Swap

```
boolean compareAndSwap(void *ref$, int expected, int newvalue) {
  Do Atomically —
  int oldvalue;
  fetch *ref$, store the value in oldvalue;
  if(oldvalue == expected) {
    store newvalue into *ref$;
    return true;
  }
  return false;
}
```

The updates to *ref$ are seen in a consistent order by all threads using compareAndSwap. Many hardware-supported implementations of this function exist and are lock-free. Each call or execution returns true if the old value was as expected after writing the new value to the shared location. If the value was not as expected, the function returns false. Other variants exist. For example, ones that return the old value instead. (The caller may compare the old value to the expected value to decipher what happened inside the function.) This peculiar primitive can help implement a rich set of synchronization functions, including n thread mutual exclusion as shown below (assume `turn$` is initially -1):

Listing 4.10: n thread mutual exclusion using Compare And Swap

```
while(compareAndSwap(&turn$, -1, threadID));
  // Critical section
turn$ = -1;
```

If a thread is able to find -1 in `turn$`, it writes its ID in that variable. Since this is done atomically, two threads may not both find it to be -1. Exactly one succeeds in writing its ID. The others retry. Actually, compare and swap primitive's application is much broader than mutual exclusion; it can be used to implement many wait-free data structures and algorithms. One important handicap of the compare and swap primitive is its fixed granularity, the size of data it operates upon. Sometimes, we need a whole set of variables to remain unchanged, while a thread applies its updates (to one or more locations). Doing this in a lock-free or wait-free manner is challenging.

Transactional Memory

Transactional memory is an emerging paradigm that seeks to address the challenges of Compare and Swap. The main idea is to define a set of operations on the shared state as a transaction and ensure serialization of these transactions. One way to ensure such serialization is, of course, mutual exclusion. Another is to optimistically perform unsynchronized operations on shared resources. These are performed in a tentative sense, but the risk of races is detected by identifying other instances of transactions. In case no race is detected, the transaction is committed and considered complete. In case a race is detected, the entire transaction is discarded and effectively rolled back. After the discard, an alternate course is followed: simply retry the transaction or apply mutual exclusion this time. Note that multiple conflicting transactions would all be discarded and retried.

Transactions are a higher-order primitive than locks and compare and swap. Hence, they may be easier for programmers: it could be as simple as encapsulating a sequence of operations into a transaction that appears to execute atomically. Further, transactions can also be nested – a transaction can consist of sub-transactions, and only the sub-transaction is discarded if it encounters a conflict. Transactions can also be composed. However, they are not do-all. For example, intricate interaction between threads, *e.g.*, in a producer-consumer problem, is not easily expressed as transactions.

It is also worth noting that the catching of all races is an expensive proposition. Data races[11] are a little easier to detect. An implementation of transaction memory may not be able to detect all races. The reality is that the more guarantees the transaction memory provides, the slower it can get. Furthermore, interactions between transaction style and traditional synchronization can also become complex to manage. In particular, roll-back of transactions may not be truly possible in all cases, *e.g.*, when interaction with a file-system, network or a user may be involved.

[11] ***Defined Data race:*** Data race occurs when two or more threads access a shared location in an unsynchronized fashion, and at least one of them is a write operation.

Barrier and Consensus

Barrier, like transactions, is a higher-order and a collective primitive. It is a contract among a group of threads, which we can call the barrier group. It's a collective because each member of the group must reach the barrier event. Every member blocks at the event until they have surety that all the other members have reached their respective barrier events. This is quite clearly not a non-blocking nor a lock-free operation. A stand-alone barrier for an n member barrier group may be implemented as follows. Initially, `numt$` is 0.

Listing 4.11: Barrier

```
void Barrier {
  int num = numt$;
  while(! compareAndSwap(&numt$, num, num+1))
       num = numt$; // Re-read count and retry incrementing
  while(numt$ < n); // Busy-wait
}
```

Each thread reads the then-current value of `numt$`. If no other thread has modified it in the interim, the thread writes the incremented value into `numt$`. Otherwise, it re-reads the new value of `numt$` and retries incrementing it. It needs to retry no more than n times before it must succeed because a successful thread does not retry. Once the thread succeeds in registering its presence, it moves to check if all threads have registered. It busy-waits until then. This barrier may be used only once. It is possible to modify it so multiple barriers can re-use the same variables. `numt$` would need to be reset to 0. But also note that threads may exit their busy-wait loops as soon as `numt$` equals n, but some could be delayed. Either a thread's next entry into the barrier must be prevented until the last one is out, or the entries would need to be otherwise separated. Implementation is left as an exercise (see Exercise 4.14).

More complex barrier variants perform additional operations once the barrier is reached. For example, the following vote function is a barrier, which each member of the barrier group calls with an argument `true` or `false`. The function returns in each thread only after all members have made the call. Further, the returned value is `true` if at least half the member call `vote(true)` and `false` otherwise.

Listing 4.12: Voting Barrier

```
// define:
bool vote (bool value);
```

Although other fancier versions exist, *e.g.*, one returns the sum of integer values supplied by the members, this simple version is instructive. It is related to *consensus*: having all threads reach the same value. Consensus is often used to argue about the power of synchronization primitives[12]. In the basic consensus problem, the returned value of vote must be the same for all members of a group,

[12] Block-chains are based on the consensus problem.

S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL https://bitcoin.org/bitcoin.pdf but the semantics of barrier is not required. Threads may provide their values and proceed. Later they may fetch the consensus value. In fact, the only requirements are:

- the consensus value must be the same for every thread
- at least one thread must provide the value determined as consensus

Compare and Swap is able to provide wait-free consensus among an arbitrary number of threads, meaning its consensus number is ∞. A sample implementation of vote follows. Assume `one_value$` is initially -1.

Listing 4.13: Consensus

```
bool consensus (bool value) {
    compareAndSwap(one_value$, -1, value);
    return one_value$;
}
```

Exactly one thread succeeds in storing its value into `one_value$`. Others find it to be the value written by the successful thread. It turns out that simple shared-memory reads and writes (as assumed by Bakery or Peterson's algorithm) cannot be used to achieve consensus of even two threads in a wait-free manner. A solution to the consensus problem can be used to implement solutions of a large variety of concurrent problems. This is called the universality of consensus[13]. Another important cog in our understanding of concurrency is that consensus is impossible to guarantee if any one of $n > 1$ threads may fail[14,15]. This is true in the shared-memory model (using only read/write) as well the message-passing distributed-memory model. In particular, consensus is not guaranteed if threads (and messages) can be arbitrarily slow. For controlled environments, which a parallel computer system may be, the knowledge of the bound on delays is employed to achieve consensus in a practical manner, even in the presence of failure. In this book, we will not focus on fault-tolerant algorithms, which continue to provide synchronization and safety in the presence of failure.

[13] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13: 124–149, 1993

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411

[15] Herlihy, 1993

4.5 *Communication*

Much of the preceding discussion on synchronization is described in terms of shared memory. That does not restrict it to virtual address space controlled by a single operating system. It also applies to distributed shared memory systems, which are built on top of message-passing primitives and give the appearance of a unified shared address space across computing systems. Alternatively, a program may explicitly employ message-passing. For example, a message-passing program may unify separate address spaces by annotating an address with the name of the computing system that owns it. A write to a unified shared address now translates to two steps: sending a write message to the corresponding owner, followed by that owner writing the value within its local address space. If such writes are non-blocking on the originator system, the previous discussion on synchronization and consistency directly applies.

Note that scalable and consistent distributed shared memory implementation is rather complex, and good performance can be hard to achieve. For many applications, direct use of message-passing primitives is easier to design, synchronize, and reason about. Note that there is natural coordination required for two or more threads to communicate among themselves. Inter-thread interactions are more direct and more explicit compared to the shared-memory model, where a passive memory location has no ability to detect anomalous interactions. Hence, it is important to understand the nature of message-passing based programming. Broadly speaking, in the message-passing model, shared states and critical sections are eschewed in favor of the synchronization implicit in communication, which has certain features of the barrier. We will see detailed examples in Chapter 6.

As a practical matter, it is useful to realize that synchronization across message-passing threads is likely to be slower as network delays are generally much higher than local memory latency. We will briefly review the communication system next. With this understanding, we can devise more efficient inter-thread interaction. Common network topology is discussed in chapter 1. In this section, we move the discussion to a slightly higher level. We assume the availability of an efficient routing protocol that is able to reliably deliver messages in order from one addressable node to another. We will also abstract address details by simply addressing a thread by its ID. Let us first understand point-to-point communication in some detail.

Point-to-Point Communication

There are two essential components of communication. A thread must *send* data, and another must *receive* it.

Listing 4.14: Point-to-point communication primitives

```
Send(IdType destinationID, DataType *buffer);
Receive(IdType sourceID, DataType *buffer);
```

Sender and recipient each must have a local buffer where that data must be stored. The sender sends from its `*buffer`, and the recipient receives into its `*buffer`. This means that the recipient must have sufficient space in its buffer to hold the entire data that the sender sends. Possibly, this size is shared in advance. Or, the communication could use fixed-size buffers, but that bounds the size of each message. A sender would have to subdivide larger messages, and that unnecessarily complicates program logic. Moreover, some setup is required to send each message (for example, route setup or buffer reservation on intermediate switches). Subdividing messages may incur the overhead of repeated setup. The other big concern is synchronization between the sender and the recipient. How does `Receive` behave if the sender has not reached its corresponding `Send` and *vice-versa*?

To answer such questions, let us delve deeper into how communications happen on the sender and recipient nodes. There is at least one network interface card (NIC) in a computing system. A special NIC processor is responsible for the actual sending of data onto an attached link or receiving data on the link. Links are passive; hence, there must be two active execution units on both ends of a link. These execution units are built into the NIC and usually have their own buffers for temporary storage of data. This means that an application program does not need to concern itself with the transmission details, nor be forced to synchronize simultaneously executing fragments on both ends of the link. For security and generality, the access to NIC operations is through the operating system, and usually through several layers of software, which may have their own limits on message or packet size. This, in turn, implies that the user buffer may be subdivided into multiple packets and copied several times (from user buffer to operating system buffer to NIC buffer). Some of this copying is done by the operating system code on behalf of the application, and some is managed by the DMA engine (see Section 1.2) associated with the NIC. See Figure 4.5.

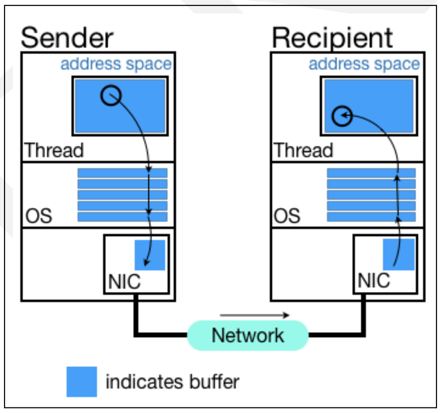


Figure 4.5: Buffer copying for communication

Some NICs also support remote DMA, or *RDMA*. The RDMA mechanism is designed to bypass most copies. The DMA engines on the sender NIC and the recipient NIC collaborate to copy data directly from the sender's buffer to the recipient's buffer without an explicit involvement of CPU code. Naturally, this method requires that the send and receive buffers be both ready (also called *registered*) before DMA can begin and remain available for the duration of the transfer. This imposes strict synchronization constraint between the sender and recipient. This constraint can be relaxed by replacing one or both buffers with operating system owned buffers, but that leads to its own complications.

One problem with DMA-based operations is their interference with the virtual memory paging system. Page management is the operating system's domain and requires CPU instructions, but the DMA engine's job is to off-load the copying from the CPU. Hence the operating system needs to lock or *pin* to real memory the pages that are in use by DMA. Thus memory registration is a heavy-weight operation, not to be repeated incessantly. Re-using registered memory for multiple data transfers is important. However, the size of the actual message is known only at the Send event, and pre-registered buffers could be too small to accommodate a transfer of the required size. Algorithms exist for dynamic re-registration and pipelined re-use of small parcels of memory[16], but we will not discuss those in this textbook.

[16] Tim Woodall, Galen Shipman, George Bosilca, Richard Graham, and Arthur Maccabe. High performance RDMA protocols in HPC. pages 76–85, 09 2006

RDMA or not, the separation of concerns between the application program and the network subsystem allows the program to 'fire and forget,' assuming that the entire message will be delivered 'as is' without any loss, corruption, or the need for further intervention or acknowledgments. The network subsystem is also able to guarantee that between one source-destination pair, all messages are delivered in the order they were sent. The use of the network subsystem and NIC buffers also means that a Send primitive may proceed asynchronously with the Receive primitive. A Send without its matching Receive means the message resides in an intermediate buffer, and the sender's execution may proceed beyond the send event. Later, when a matching Receive is finally executed, the data is copied to the recipient buffer from the intermediate buffer.

Similarly, the recipient also need not be blocked at the Receive event, even if there is no matching Send, as long as the recipient does not expect to access the received data immediately past the Receive event. This can be accomplished by subdividing the Receive event into a *StartReceive* event and a *CompleteReceipt* event. Indeed, analogously to the shared-memory case, *CompleteReceipt* could be implemented as a blocking event, or it might be executed in the busy-wait style. StartReceive allows the recipient to provide the receive buffer. CompleteReceipt ensures that the data is filled in the buffer.

Symmetrically, the Send primitive may also be similarly subdivided into *StartSend* and *CompleteSend* events. StartSend initiates sending. The sender contracts to keep the data unchanged in its buffer. Getting past the CompleteSend event releases the sender from this contract. When both Send and Receive events occur together, the communication is called *synchronous*. When they can occur at their own pace without necessarily overlapping, the communication is called *asynchronous*.

RPC

RPC, or remote procedure call, is a type of point to point communication but not described explicitly as a Send-Receive pair. Rather, a thread makes a function call that looks similar to a local function call, except the call is executed on a remote system. This means that the arguments of the function are packed into a message and sent to the designated recipient. The ID of the recipient may be a part of the call or pre-registered with the function's name. On receiving the message, the recipient unpacks the arguments and calls a local function, which in turn may make another RPC. Once the function execution is complete, the function provider packs the value returned by the function into another message and sends it back to the initiator of the RPC. Synchronous RPC requires that the initiator only proceeds beyond the call after receiving the results back. Asynchronous RPC, not unlike asynchronous Send and Receive, allows the initiator to continue execution beyond the RPC call without receiving the results. The initiator may later invoke a CompleteRPC function to receive the result back. The basic operation is demonstrated in Figure 4.6.

RPC can be thought of as a higher-level communication primitive built on top of the Send-Receive primitive.

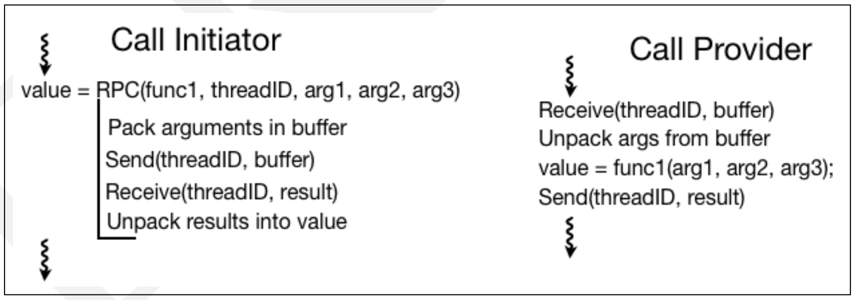

Call Initiator
value = RPC(func1, threadID, arg1, arg2, arg3)
Pack arguments in buffer
Send(threadID, buffer)
Receive(threadID, result)
Unpack results into value
Call Provider
Receive(threadID, buffer)
Unpack args from buffer
value = func1(arg1, arg2, arg3);
Send(threadID, result)

Figure 4.6: Remote Procedure Call

Collective Communication

Sometimes a more complex pattern of communication can exist among a set of cooperating threads. Describing complex communication among a set in terms of several individual point-to-point pairs is wasteful. Such higher-level primitives can again be built on top of the Send-Receive primitive. These group level, or *collective communication primitives*, are similar to the barrier: all threads in a group encounter this event. However, unlike the barrier, they need not do so simultaneously. Rather, communication may be asynchronous and strict synchronization mandated by the barrier is not necessary. Some common collective communication primitives are listed below. Chapter 6 describes specific implementations and contains some more detail.

Broadcast:
Message from one sender is received by many recipients

Scatter:
n messages from one sender are distributed to n recipients, one each

Gather:
One message each from n senders are received by a single recipients

AlltoAll:
Each member of a group consisting of $n + 1$ threads scatters n messages (one each to the other members) and consequently gathers n messages (one from each recipient).

Reduce:
This is a special type of gather, in which one gatherer collects data from n others, but instead of storing these n items in n locations, it reduces the vector of n items to one using a reduce operation (*e.g.*, by taking their sum).

4.6 *Summary*

This chapter broadly discusses inter-thread interaction. These may be through shared memory or direct message passing. A key lessons in this chapter is that memory operations are not instantaneous. They take finite time, the length of which can vary significantly. Moreover, multiple memory operations of each thread in a parallel environment may overlap in execution. This can cause unexpected program behavior because operations started earlier could end later. Therefore, when evaluating program logic, it is important to understand the guarantees provided by the programming platform.

In particular, one must not assume that if one thread observes the effect of memory operation o_1 before o_2, all other threads would observe the same order. If the platform does not guarantee so, the program must include explicit synchronization to ensure consistency where needed. Missing synchronization often leads to errors that may be hard to reproduce. Memory consistency errors are among the most obscure. Correct execution in a large number of test cases should not be taken as a proof of correctness. To reiterate the main points:

- Addresses hold values and may be accessed by multiple code fragments. The instructions of two or more code fragments that share addresses may execute in parallel or interleave. Their accesses are hence concurrent, and their order of execution is non-deterministic.

- Two fragments that access a shared address experience a data race if at least one of them updates the value of the address, and the order of their accesses is non-deterministic. Not all data races impact the results. If the relative timing of execution of instructions by fragments that share some addresses impacts the correctness of results, we call it a race condition.

- In parallel execution of threads that share memory, the sequential nature of memory operations does not hold. The operations by concurrent threads have no defined order between them, and there need not be a common serializing storage, which allows one operation at a time. Hence, the update of shared memory data by one thread can become visible to other threads at different times. More importantly, two updates may become visible to different threads in different orders. This means, *e.g.*, that in the view of one thread, variable x may change from 3 to 5, while in another thread's view, the value 3 may appear later. Thus, their decisions based on the value of x may become inconsistent.

- To avoid such inconsistencies, the order in which the threads view the updates must be consistent. Two views are consistent if they both lead to the same result. However, under what circumstances should the results be the same? We have seen several kinds of circumstances. The brute-force definition leads to sequential consistency: all operations seem to happen in a sequence – one strictly after the previous with no overlap. This means that if in any thread's view, operation o_1 seems to occur before o_2, no other thread may see the effect of o_2 before that of o_1, even if the reordering has no impact on the results of the execution. Note that it is not necessary for any specific thread to see o_2 before o_1.

Inconsistency ensues even if, *e.g.*, one thread view o_2 before o_3 and another views o_3 before o_1. Ordering respects transitivity.

- Other notions of consistency are useful. Examples include Causal consistency, Processor consistency, FIFO consistency, etc., which enforce ordering requirement only on certain operations. Many programs can be proven to have the expected behavior even under the relaxed definition. An understanding of memory consistency is important to prove the correctness of shared-memory programs. Indeed, when inspecting shared-memory code, we often unconsciously assume certain consistency. It's important to know when we may be over-assuming. A guarantee of consistency by the platform has performance implications. Hence, in practice, popular programming platforms only guarantee consistency on demand – on certain variables at certain times. This allows the program to increase performance when strict global ordering can be dispensed with. An understanding of memory fences helps this endeavor.

- Memory fences are special operations within threads whose order is globally consistent across threads. This allows threads to ensure global visibility of regular memory accesses at each fence. Thus, accesses are not all individually consistent, but can be grouped into sets such that the order between the sets is globally consistent.

- Like memory fences, computation fences are also useful. Computation fences are more flexible and programmable and are called synchronization in general. Other than serializing execution steps of different threads, synchronization afford the ability to pause and resume the execution of a thread based on a global condition, one that depends on the execution of other threads.

- Some synchronization is possible using consistent shared variables (*e.g.*, Peterson's algorithm). These protocols assume a consistent order of updates visible to all threads, and are often non-blocking (meaning that the execution continues within the program), but not necessarily lock-free (because their progress could be stalled by other threads). Other synchronization protocols require additional scheduling support from the programming platform, where the program is blocked from execution and subsequently resumed only if the synchronization condition holds.

- Tools of synchronization include mutual exclusion (exclusive access by a thread to one or more shared addresses during the execution of a specified code fragment), signal-wait (suspending execution until some variables attain certain values), barrier (suspending until all threads in a group have completed their execution of a related code fragment), and atomic instructions (a small fixed code fragment providing mutual exclusion). Protocols using these tools require care to avoid deadlocks and starvation.

- When designing synchronization protocols, it is useful to be clear about the activity that is being synchronized. In particular, one may formulate global conditions that must remain true after the activity is complete. The design of the synchronization events then depends on the type of activity, performance requirements, and ease of programming.

- Communication between threads through shared memory is somewhat indirect and asynchronous, with the possibility of separate synchronization involving two or more threads. In contrast, some synchronization is built into message passing – all participating threads must take explicit action for each communication. These actions may be synchronous (akin to a barrier) or asynchronous. Nonetheless, there is a one-to-one matching of actions, meaning that each action of a thread can be associated with a corresponding action of partner threads. Communication through shared memory is often fine-grained, whereas message passing is usually coarse-grained. This is because message passing requires significant setup and often requires successive copies to a pipeline of buffers.

- Communication can be direct and point to point between a pair of threads. High order communication involves multiple threads exchanging their data in some pattern. In either kind, the participating threads may choose to enforce synchronization along with communication. Or, they could communicate by leaving messages in a mailbox to be fetched asynchronously. Even then, every receive event must match some send event. Even for collective message passing, *i.e.*, data exchange among a group of threads, the collective event appears in each thread and matches each other.

The textbook by Herlihy et al. [17] contains an excellent treatise on shared-memory programming and general issues of concurrency. The unifying idea of using Consensus numbers to argue about the power of synchronization primitives was introduced by Herlihy [18]. The survey by Adve et al. [19] covers the gamut of memory consistency, whereas Mosberger [20] analyzes the trade-offs of weaker consistency models. Among the most successful high level message-passing interfaces is MPI[21,22], which we will discuss in some detail in Chapter 6. Common communication interface[23] offers a deeper look at the breadth of message-passing issues.

[17] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375

[18] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13: 124–149, 1993

[19] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. DOI: 10.1109/2.546611

[20] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1): 18–26, January 1993. ISSN 0163-5980. DOI: 10.1145/160551.160553. URL https://doi.org/10.1145/160551.160553

[21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22 (6):789–828, 1996. ISSN 0167-8191. DOI: https://doi.org/10.1016/0167-

Exercise

4.1. Explain with an example how a cache-coherent memory system could be sequentially inconsistent.

4.2. Could a cache-coherent memory system be FIFO inconsistent? Explain.

4.3. In most ways, a file shared by multiple threads acts like shared memory. Consider a file system in which, instead of general write operations, a thread may only *append* to the 'end' of a shared file. Note that threads may share multiple files. The platform guarantees that the data of two concurrent appends to one file are serialized, meaning their data are not interleaved. Reading threads may read from any address. What additional support from the platform is necessary to ensure that the files are sequentially consistent?

4.4. Provide two examples of situations when a data race may be harmless and does not require synchronization? (Consider memory consistency issues.)

4.5. Rewrite the following dining philosopher's pseudo-code to eliminate the possibility of deadlock, assuming each thread in the group executes this code.

```
philosopher(place, numplaces):
  left = (place-1)%numplaces
  right = (place+1)%numplaces
  Repeat:
    lock(lock$[left])
    lock(lock$[right])
    Eat()
    unlock(lock$[left])
    unlock(lock$[right])
    Ponder()
```

Hint: Change the protocol depending on whether place is odd or even.

4.6. Change the code in Exercise 4.5 to make it non-blocking. Is your code lock-free also?

4.7. Identify the race-condition in the following code if multiple threads may execute it concurrently. The lock and unlock act as memory fences.

```
if (Ref$ == null)
    lock(lock1$)
    tmp = allocateMemory()
    initilalize(tmp)
    Ref$ = tmp
    unlock(lock1$)
use(Ref$)
```

4.8. We modify the code in Exercise 4.7 as follows.

```
if (Ref$ == null)
    lock(lock1$)
    if (Ref$ == null)
      tmp = allocateMemory()
      initilalize(tmp)
      Ref$ = tmp
    unlock(lock1$)
use(Ref$)
```

Does it resolve the race-condition? On execution by two threads A and B, thread B fails with the error “Uninitialized Ref$". Explain.

4.9. Consider the following code with shared variables A$ and B$.

```
x = 2*A$;
B$ = A$ + B$
```

Suppose the compiler optimizes away the second read of A$ on line 2, and reuses instead the value read earlier at line 1 (that it had saved in a register). Could that ever violate FIFO consistency if the memory subsystem guarantees FIFO consistency?

4.10. Some languages include syntax to designate a variable as 'volatile,' which means it is not cached and that the compiler does not reorder or eliminate its read/write. Suppose threads share only volatile variables. Suppose also that a single memory block processes all reads and writes in the order it receives them in. Does that guarantee sequential consistency? If so, prove it. If not, what additional conditions are necessary to meet before sequential consistency can be guaranteed?

4.11. Reconsider the following listing (Listing 4.3)

```
A$[threadID] = 1
print A$[1-threadID]
```

If two threads execute this code concurrently, and both print 0, is the platform FIFO consistent? Is it Causally consistent? Explain.

4.12. Rewrite Peterson's algorithm for mutual exclusion using memory fences assuming the platform only guarantees FIFO consistency.

4.13. Compare and Swap depends only on the current value of the associated variable, and not on its history. For example, a thread performs its swap if it sees the value v that it expects. However, the value could have been changed from v to v' by some thread and back to v by some thread. This is often harmless. But if the value is an address, even if the address itself changed back to v, the contents at address v could have changed. Propose a modification to the compare and swap protocol which allows a thread a guarantee that there has been no change made to v.

4.14. Implement the function barrier described in Section 4.4, which can be called by all members of a thread group any number of times.

4.15. Barriers can exist in shared-memory based interaction as well as message-passing based interaction between threads. Message-passing based barriers may be implemented using point-to-point messages. What is the fewest number of point-to-point messages required to accomplish the barrier functionality in that case? What is the minimum number of shared addresses and accesses required to implement barrier for shared-memory threads?

4.16. Memory fences are also known as memory barriers. How are memory barriers different from (computation) barriers?

,

4.17. Consider the collective communication primitive: *gather*, in which n_i data items are to be received from thread number $i, i \in 1..N$ by thread 0. Thread 0 must gather these data items contiguously in its local address space in the order of thread numbers from which the data came. Describe the steps required to implement this gather using point-to-point communications. You may use StartSend/CompleteSend and StartReceive/CompleteReceipt primitives.

4.18. Suppose a message transport system is available which subdivides large messages into fixed-size packets and sends them with the guarantee that packets sent by thread i to thread j arrive in the order they are sent. Threads are allowed to use *Send* and *Receive* primitives in the order of their choosing. Under what conditions may *Send* or *Receive* deadlock?

4.19. What is the difference between the terms *lock-free* and *non-blocking*? Could a *barrier* event be lock-free? Could it be non-blocking?

4.20. Provide a protocol to implement *Consensus* among message-passing threads.

4.21. Implement lock-free ATMs. All ATMs share the addresses `Balance$[accountNumber]`. ATMs must provide lock-free methods for:

i Withdraw(accountNumber, amount)

ii Deposit(accountNumber, amount)

iii Transfer(accountNumberFrom, accountNumberTo, amount)

You may use Compare and Swap. Assume that the provided account number is valid, and the same account number may be used at multiple ATMs at one time. Neither the bank nor any account holder should lose money.