Algorithm	Mutually Exclusive	Starvation-Free	Deadlock-Free	Fair (FIFO)
Peterson Lock	Yes	Yes	Yes	Yes
Dekker Lock	Yes	Yes	Yes	No
Filter Lock	Yes	Yes	Yes	No
Bakery Lock	Yes	Yes	Yes	Yes
Atomic-RMW-Locks	Yes	No	Yes	No

Table 1: Comparing Different Lock Implementations

1 Comparing all Discussed Lock Implementations

We compare the different lock implementations we have encountered in the lecture here regarding their key properties (mutual exclusion, deadlock-freedom, starvation-freedom, fairness) to get an overview. Some of those entries are far from obvious though, so we also discuss how we can prove some of those properties (especially the ones not discussed in detail in the lecture) and what the proof (sketches) of some of them are.

2 Before we Start: Assumptions

In order for these definitions and proofs to make sense and be meaningful, we need to make a few assumptions. Make sure you understood those, else, it will be very difficult to make sense of the rest.

2.1 The Anatomy of a Lock Algorithm

When analyzing a lock algorithm, we divide the code into a few parts:

- **Non-critical section**: This is the code part before the thread starts acquiring the lock. It does not have anything to do with the lock algorithm itself, but it is important to have it there: A thread is not simply acquiring and releasing a lock (with a critical section in between), but it also executes some non-critical code (that is, code that does not require mutual exclusion) before trying to acquire the lock. This is modelled with this non-critical section.
- **lock() method**: This might be called differently in the lecture and literature, but we are going to refer to the algorithm that is executed to actually acquire the lock as `lock()` method. We split the `lock()` method into two parts, the doorway and the waiting section as listed below.
- **Doorway**: A *finite* part of the `lock()` method. Usually, flags are set and variables written that are used to coordinate and communicate with competing threads. Note that the doorway must contain a finite number of instructions, so it cannot contain for example spin-loops.
- **Waiting section**: A part of the `lock()` method that may contain an unbounded number of instructions. Usually, this section comes after the doorway and is simply a spin-wait to wait for some condition to be fulfilled (such that the thread can actually enter the critical section, i.e. acquire the lock).
- **Critical section**: This is the code that is executed while a thread actually holds the lock. Like with the non-critical section, the actual code here is irrelevant.
- **unlock() method**: The code that is executed after the critical section to make the lock available again for other threads.

Let us show on the example of Peterson's algorithm how we would divide it up into these different sections:

```
/* non-critical section:
* Code the thread executes before trying to acquire the lock
*/
flag[id] = true; // lock() method: doorway
victim = id; // lock() method: doorway
```

```
while (flag[1-id] == true && victim == id); // lock() method: waiting section
/*
*  critical section
*/
flag[id] = false; // unlock() method
```

2.2 The Progress Assumption

For many proofs, we make an assumption that is vital. Without it, many proofs do not make sense (because they would be wrong) and thus we need to clarify it. We call it the *progress assumption*: We assume that threads in the `lock()` method, the critical section and the `unlock()` method have finite time in between instruction executions. That is, we assume it is impossible that a thread can simply not get scheduled forever. It must execute its next instruction in *some* finite time. Intuitively, this means that a thread cannot "die" in these parts of the code and we can assume it will eventually carry on with its algorithm, albeit after an arbitrarily long time. However, we do not make this assumption for the non-critical section, which precedes the call to `lock()`. There, it is possible that a thread "dies" in the sense that it might enter an infinite loop for example.

3 Mutual Exclusion and Deadlock-Freedom

Notice that all the discussed algorithms provide mutual exclusion (that is their primary point, so that is expected). A usable lock must at minimum provide deadlock-freedom in addition to mutual exclusion, so they also all provide that. Remember that deadlock-freedom means that when at least one thread is contesting for the lock (they called lock(), i.e. they are trying to acquire it), (at least) one of them is guaranteed acquire it in finite time. If that would not be the case, the algorithm would be quite unusable.

4 Fairness

Fairness proofs are not central to the course. However, if you want to understand why some locks are fair and some are not, you will find very precise definitions of fairness and proofs that are both rigorous and also give intuition in this section. By simply reading the argumentation, this might help you remember which locks are fair and which are not and why this is intuitively the case.

4.1 Fairness in a Lock

First of all, there is no universal definition of fairness in the literature. Some sources consider fairness as something we refer to as starvation-freedom. In this course, we consider a lock algorithm to be fair, if it fulfills FIFO order. Note that FIFO (first-in-first-out) and first-come-first-served mean exactly the same thing.

The nice thing about this notion of fairness is the simple definition of a FIFO lock: A lock algorithm fulfills FIFO (or first-come-first-served) ordering, if and only if a thread that is "first-in" (compared to some other thread) is also guaranteed to acquire the lock first.
The hard part is then to define what "first-in" means. Let us first motivate why this is a very difficult definition: Assume we say "first-in" means that the thread first called `lock()`. However, imagine a lock that actually works in a FIFO queue manner (enqueues arriving threads). We would definitely consider this fair. However, in the `lock()` method, a thread presumably would need to first enqueue itself. Here we could say that it might not do execute this enqueue for a long time and thus a thread that actually called `lock()` at a later point could enqueue itself earlier and thus overtake our thread. We see that this definition of "first-in" is not meaningful.

Another idea might be to define "first-in" such that some thread is "first-in" if it executed the first instruction of the `lock()` method before some other thread. However, what an instruction is, is quite arbitrary. Consider the Peterson lock. We can divide the first instruction into a load, modify and store and thus basically nullifying the advantage of the "first-in" thread. We can do this since in such pseudo-code, there are no real rules about what an instruction has to be. However, even when taking the actual Peterson algorithm, saying a thread is "first-in" if it raised its flag first, this would not guarantee that the thread acquires the lock first. So, with this "first-in" definition, Petersons algorithm would not be considered FIFO and thus not fair (according to our notion of fairness). However, Peterson's algorithm is quite universally considered FIFO. We need to find something else:

Remember our definition of the `lock()` method and its doorway and waiting section. We consider a thread P to be "first-in" compared to a competing thread Q, when P finishes its doorway section before Q starts its doorway section. And now, a lock algorithm fulfills FIFO ordering, if and only if one can divide its `lock()` method into a finite doorway and (possibly unbounded) waiting section, such that a thread being "first-in" compared to some other thread, is guaranteed to acquire the lock first.

This is not a terribly simple definition, but it is very precise when defining "first-in" the way we did.

4.2 Fairness of Peterson

Petersons lock is easily shown to be fair (remember Peterson only synchronizes two threads) when we consider the doorway and waiting section as described in 2.1: Assume thread P is first-in, which means according to above definition that it finished its doorway (i.e. set its flag and set the victim to its ID) before a competing thread Q started its doorway. Now either of the follwing happens:

- The competing thread Q has still not started its doorway when P evaluates the condition of the waiting section for the first time. In that case, Q has not set its flag. Hence, P can advance to the critical section and will also do so before Q. This is because Q cannot advance unless either P sets the victim to its own id again or sets its flag to false. Both of which can only happen after P exited the critical section. So, Q is blocked and due to the progress assumption, P will enter the critical section in finite time, in particular before Q.
- The competing thread Q has already entered the doorway section when P evaluates the condition of the waiting section for the first time. The progress assumption guarantees us that Q will finish the doorway in finite time and in particular will write victim=Q. Now, P can advance to the critical section and will do so before Q because of the same reasoning as in the first case (Q is blocked due to victim==Q and progress assumption guarantees P will continue eventually).

In all cases, P enters the critical section before Q, and thus the algorithm is fair.

4.3 Why is Dekker's Algorithm not Fair?

Consider Dekkers Algorithm:

boolean wantp ← false, wantq ← false integer turn ← 1	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp ← false	q5: wantq ← false
p6: await turn = 1	q6: await turn = 2
p7: wantp ← true	q7: wantq ← true
p8: critical section	q8: critical section
p9: turn ← 2	q9: turn ← 1
p10: wantp ← false	q10: wantq ← false

Say now thread P was "first-in" before thread Q. Here, we can only define the doorway to be instruction 2, the raising of the flag. This is because the doorway must contain a finite amount of instructions, and thus we cannot include even parts of the while loop (lines 3-7). Thus, P being "first-in" simply means it raised its flag first. Now, first-in does not guarantee us that P evaluates the while-condition (p3) before Q enters its doorway and raises its flag (q2). If that happens, both P and Q have to execute their while-loops to decide who can enter first and there, the turn variable decides. Since turn is initialized to 1, Q will enter first here and thus, "first-in-first-out" is violated.

4.4 Why is the Filter Lock not Fair?

Consider a thread P that is "first-in" compared to some other thread Q at a level i in the filter lock. It is sufficient to show it is possible that Q can move on to level i+1 first to disprove fairness.

Then, being "first-in" means that P set its level to i and victim[i]=P (those two instructions can be defined as the doorway) before Q could do any of that. Now, consider an interleaving where Q sets its level to i and victim[i]=Q. Now, our thread P could advance, but assume it does not do so yet (remember, we only need to show it is *possible* that Q advances first, not that it *must* happen). Now, a third thread R enters level i and sets its level to i and victim[i]=R. Now, also Q can advance and nothing prevents Q from advancing before P does so. Hence, P can get overtaken and thus, fairness is violated.

Critical readers might now point out that our fairness definition only specified that *some* valid doorway definition must exist to show fairness. Hence, we would have to exhaustively show for all valid doorway/waiting partitions of the `lock()` method that FIFO can be violated. However, only showing this one provides enough intuition as to why the lock is not fair.

5 Starvation

5.1 Proving Starvation Freedom in General

Starvation formally means that any thread that calls `lock()` will acquire the lock in finite time (that is, the `lock()` method will terminate/return). So, to prove starvation freedom, we assume an arbitrary thread (that called `lock()`) and show that it will finish its `lock()` method in finite time.

5.2 Starvation Freedom of Filter Lock

Assume an arbitrary thread. It is sufficient to show that the thread can advance from some arbitrary level i to level i+1 in the filter lock in finite time. In a fully formal proof (which is not expected in this course), the following reasoning would roughly correspond to the induction step (a formal proof would perform induction over the number of levels in the lock).
We proceed by case distinction:

- Either, a new thread reaches level i and sets the victim to itself. Thus, our thread can advance (consider the while-condition of the filter lock) and will do so eventually due to the progress assumption.
- Or, no other thread ever reaches level i after our thread. Say now that k threads are ahead (or on the same level) of our thread in the lock. Deadlock-freedom of the filter lock guarantees us that in finite time, one of them will acquire the lock and due to our progress assumption will finish the critical section and unlock in finite time as well. Now, k-1 threads remain ahead of our thread or on the same level. Remember that no additional threads can come, because we are in the case where no other thread ever reaches level i. We can simply use deadlock freedom (plus our progress assumption) again to reach a scenario with k-2 threads ahead of, or at the same level as our thread. We use deadlock freedom k times in total until no thread remains ahead of or on the same level as us.
  Now, the while-loop evaluates to false and our thread can advance (and will do so eventually due to the progress assumption).

Since these two cases cover all possible scenarios, the statement follows (the thread can indeed reach level i+1 in finite time). Since the thread was arbitrary and we can extend this (imagine induction with this as the induction step) to all n levels in the lock, starvation freedom holds.
This case distinction should provide enough intuition as to why a thread must make progress (and thus cannot starve) in the filter lock: Again, either a new thread reaches the level and thus our thread can advance (because it is no longer the victim). Or, no thread comes, but then all the ones ahead must drain at some point and then the thread can advance also. If you simply remember this simple two-sentence reasoning, you will be able to reconstruct why the filter lock is starvation-free. A full formal proof is not required in the course.

5.3 Why are Locks using Atomic-RMW Operations not Starvation-Free (and not Fair)?

Let us consider a spinlock using the atomic TAS (test-and-set) operation:

```
void lock(boolean lk) {
  while (!testAndSet(lk));
```

```
}

void unlock(boolean lk) {
  lk = false;
}
```

This is pseudo-code. Remember the semantics of test-and-set: The operation *atomically* checks whether `lk` is false and writes it to true if it is. In that case, the operation succeeded, and test-and-set returns true. If `lk` is already set to true, the operation fails and test-and-set returns false.

Now, a thread tries to acquire the lock by calling `lock()` and spins in the while loop until test-and-set returns true. It could return false because another thread currently holds the lock. Our progress assumption guarantees us however that this thread will finish its critical section and release the lock in a finite amount of time. But even then, test-and-set can fail again when a competing thread succeeds and our thread fails. There is nothing about these atomic read-modify-write operations that guarantees us something about the ordering that competing threads succeed. Our thread could simply never succeed, because every time the lock is released again, another thread that also executes the TAS operation could succeed and thus our thread fails.

With the same reasoning, one can also show trivially that these locks are not fair. The *lock()* method consists *only* of a waiting section, there is no possibility of defining a doorway (because a doorway must contain a bounded number of instructions, so no spin-loops allowed). Hence, being "first-in" means essentially nothing for such locks, since it does not even guarantee that the "first-in" thread starts executing the atomic operation first.

5.4 How can it be that Atomic-RMW based Locks are Used then?

This is not super relevant knowledge for the exam, but it may answer some questions and concerns interested readers may have and provide some food for thought.
It may seem strange, as we defined a correct lock needs to fulfill starvation freedom. However, locks based on atomic-RMW operations are the only lock implementation actually used in modern systems and they are **not** starvation-free. How can this be? Starvation-freedom is just not a super important property. Think about it, a lock that can deadlock is unusable. Even when the probability of it deadlocking is low, such an algorithm deployed in, say, a banking system could prove fatal. This is because there is by definition no escape from a deadlock. The system is just in a state where it cannot get out of. But what the lack of starvation-freedom in atomic spinlocks means, is that *technically*, the atomic operation could always fail (due to contention). In the sense that there is nothing that guarantees us that it will eventually succeed. But, the probability of this happening is 0. Because to actually starve, a thread would need to try the atomic operations an infinite amount of times and fail every single time.

In an actual implementation, using backoff, the probability of failing multiple times in a row is already extremely low. But, it *could* technically happen that the atomic operation just keeps failing. That is why we are not guaranteed starvation freedom. But, this barely has any implications on a real-world implementation.

To further emphasize how starvation-freedom and deadlock-freedom relate, consider a deadlock-free lock algorithm. Let us assume we have a finite number n of threads that try to acquire the lock. Each of those n threads can also acquire the lock at most k times in total (cannot reenter `lock()` more than k times). These are strong assumptions, but in a real-world scenario, threads are probably not going to access a lock an infinite amount of times anyways and we also do not have an infinite number of threads. So, they are strong assumptions, but not entirely unrealistic. The key point is, we have at most $n * k$ lock acquisitions in total, which is a finite number.

Now, deadlock-freedom guarantees us that at least one of the competing threads will succeed in acquiring the lock. Due to our progress assumption, we also have that this successful thread completes the critical section and releases the lock again in finite time. Thus, we land in a scenario where there are at most $n+k-1$ competing lock acquisitions in a finite time. We can again apply deadlock-freedom here to get to a scenario with at most $n+k-2$ competing acquisitions and so on. At the end, all of the $n+k$ competing acquisition attempts must succeed (by applying deadlock-freedom $n+k-1$ times). Thus, we have starvation-freedom. We see that in such a scenario (where the total number of acquisition attempts is bounded), starvation-freedom and deadlock-freedom are equivalent. Starvation-freedom can only be a stronger criterion than deadlock-freedom when we have an infinite number of lock acquisitions.