Atomic Transactions

Hello, and welcome to this lesson on atomic transactions.

Transactions are alternative to lock-based synchronization.

We will first explore a number of problems that are inherent to locks. These range from reduced concurrency to difficulty of building large systems using locks.

We will then introduce transactions as an alternative to locks. Transactions are well-known from database systems, but we will treat them as just a way of grouping together sequences of operations.

We will learn about serializability, which is very much related to atomicity. We will learn that, as long as we can ensure serializable execution of operations of transactions, it is then easy to provide atomicity.

The problem is that serializability must be ensured, and we will discuss a lock-free method that allows us to do this. This uses time stamps and is called either "**time-stamp-based**" or "**optimistic concurrency control**".

We will also have to deal with how to handle transactions that abort, for whatever reason.

By the end of this lesson you will understand the benefits of using transactions, and you will understand (in general terms) how transactions are implemented using time stamps.

Let's begin.

Locking has many Problems

So, what's the problem with locks?

As we know from designing concurrent data structures, locks can severely reduce concurrency in the system, and therefore reduce its performance and scalability.

Locks can also lead to deadlocks.

Let's look at an example.

Locks lead to Deadlocks

Let's assume that we have 2 threads, Thread 1 and Thread 2.

And we have two locks, A and B.

Thread 1 locks A and acquires it.

After a while Thread 2 attempts to lock B

It succeeds as well. (5)

After some more time, Thread 2 attempts to acquire lock A, which is locked. It therefore blocks.

Thread 1 now wants to acquire lock B, and blocks, because B is owned by Thread 2.

Now the two threads are deadlocked, each waiting for the other to release the lock.

Locking has many Problems

Locks also can lead to priority inversion, which is more subtle than deadlocks, but can cause all kinds of difficulties as well.

Locks lead to Priority Inversion

Maybe the most famous example of a failure due to priority inversion happened on Mars as part of the mid-nineties "**Mars-Pathfinder Mission**".

Soon after the pathfinder landed on Mars on July 4, 1997,

It started having, what was euphemistically called "**software glitches**".

In reality, the pathfinder's computer system started resetting each time it started gathering meteorological data.

The reason for this was that a periodic watchdog process, whose job it was to check whether all other processes had completed in time, detected uncompleted process executions. In order to recover, the watchdog triggered a reset of the system.

After quite some studies of what happened, the engineers were able to identify the reason for the overruns, which was priority inversion due to locks.

Priority Inversion on Mars Pathfinder

Let's look in detail at what happened by drawing a timing diagram of the events that led to the timing overruns.

There is a low-priority data collection thread, a few other threads, which were running at intermediate priority, and a high priority data distribution thread, which had to make sure that all the data was distributed over a very slow bus.

Let’s follow these threads as they execute.

The data collection thread starts, and after some time locks a mutex. This mutex was not directly visible to the programmer, but was inside the operating system. The programmer had called a “**select**” call, which internally created a mutex to protect the “**wait list**” of file descriptors for devices that support “**select**”. In this case, this was the “**pipe**” device for inter-process-communication. (5)

Without the programmer’s knowledge, the thread now entered a critical section that was protected by the hidden mutex.

After a while, the high-priority data-distribution thread wakes up and starts distributing the collected data over the bus.

Because it has a high priority it preempts the low-priority data acquisition thread. This thread is currently inside the critical section. The high-priority thread starts executing.

At some point, the high-priority thread needs to communicate over a pipe, for which the operating system needs to acquire the lock that is currently used by the data acquisition thread.

This is not a problem, as the high-priority thread simply gets blocked, and the low-priority thread continues.

Whenever the latter releases the lock, the high-priority thread enters the critical section and continues.

Here we highlight the period of time when the high-priority thread was blocked by a low-priority thread.

Instead of the high-priority thread, a low-priority thread is running.

This is called “**priority inversion**”.

Until now it looks rather innocuous, but let’s look at what happened when other threads were involved.

-

In the failure scenario, the low-priority thread started, acquired the lock, but was now interrupted by a medium-priority thread.

Sometime later the high-priority data distribution thread starts and preempts the currently running thread.

It then, blocks on the mutex, which is held by the low-priority data collection thread.

Rather than returning control to the low-priority thread, it goes to the preempted medium-priority thread, which of course has a higher priority than the former.

The running thread is preempted by some other medium-priority thread with slightly higher priority. (5)

When this one completes, the first medium-priority thread resumes until the second thread becomes ready again and interrupts the first.

This goes on for a while, until a timer triggers the watchdog.

This one detects that the data distribution thread is blocked and assumes that something went wrong. After all, the data distribution thread as assigned the high-priority because the designers wanted to reduce its response time. As a result, the watchdog resets the system.

We notice that the priority-inversion period is now much longer. This is because the high-priority thread is blocked by the low-priority one, which in turn is starved by the medium priority threads.

If you are interested, check out **Mike Jones's** description of how the **Jet Propulsion Laboratory at Caltech**, in collaboration with the operating system suppliers, was able to fix this problem on Mars from Earth. The solution was to modify the behavior of the mutex. In general, mutexes can be configured to inherit the priority of higher priority threads that are blocked. In this way, priority inversion is greatly reduced. However, one has to be aware that such a mutex exists. In this case, the mutex was hidden deep inside the innards of the operating system.

We will re-visit this problem when we talk about composability.

Locking has many Problems

The next in the long list of problems with locks is that they lead to so-called "**convoying**".

Locks lead to Convoying

This happens when a thread, here Thread 1, locks a mutex for a significant amount of time, during which many other threads get blocked.

After the first thread releases the mutex, it takes a while for the queue to clear.

Also, there is a good chance that the same threads queue up again on the same or a different lock.

Locking has many Problems

An additional problem that is inherent to locks is their lack of support for what may be called “**soft**” composability. Here we distinguish “**soft**” from what we call “**hard**” composability. By soft we mean that composability is difficult, whereas lack of “**hard**” composability means that composability is not possible or very difficult.

By “**composability**” we mean in general that one should be able to compose a system by combining together existing components. For example, one may want to compose a multilevel scheduler by combining together multiple components, each of which implements a round-robin queue, plus an aging component maybe. In a system that supports what we call “**soft**” composability this should be possible without knowing anything about the internals of the components.

As soon as we need to know about the internals of a component, this means that we need to know how to use it beyond the level that is specified by the component interface, and we find ourselves in the realm of so-called “**convention-driven programming**”.

Locks lack “soft” Composability

Let’s look at how well convention-driven programming does inside the Linux code.

An early study of Linux found that over a third of all bugs in Linux were synchronization bugs.

A bit later, a study found 12 deadlock bugs, of which 4 were confirmed, and for the remaining 8 it was not possible to confirm whether they would indeed lead to deadlocks.

The main problem was that there is no systematic way to manage locks, other than for trivial critical section scenarios. Rather, each use of locks has to be documented in detail.

For example, one particular function inside the memory manager has a 50-line comment describing the detailed order of how locks were used. The comment described how locks were used 4 function calls deep into the file.

A programmer making use of this map has to have intimate knowledge how locks are used inside its functions, which leads to all kinds of problems.

- Cont.

Another example of detailed documentation of the lock behavior inside a function is given here. In this case, the code is part of the buffering system for the file system. The programmer is given clear instructions on how to use this code so as to not generate deadlocks. Writing code using components like this is not fun.

Locking has many Problems

What we call "**hard**" composability is even more difficult to deal with.

We talk about "**hard**" composability when the components cannot be combined into a larger system even if the programmer knows all the details about the components and is willing to adhere to all conventions.

Locks lack "hard" Composability

Let's look at an example.

Let's assume that we have a component that implements a queue.

The queue is protected by a lock to enforce mutual exclusive access to the internal data structures of the queue.

Now we want to build a larger system of two queues, which we want to concatenate for some reason.

The challenge is to concatenate these queues so that we atomically transfer the item at the front of queue 1 to queue 2.

By atomically we mean that no other threads can see that **x** is missing from both queues or that it is present in both queues.

This will be much more complicated than expected.

- Cont.

One solution is to remove **x** from Q1, and then add it to Q2.

+The problem with this solution is that once we have dequeened **x**, other threads can observe that it is missing from both queues.

A different approach would be to first get a copy of **x** from Q1 without dequeuing it, then adding it to Q2.

And finally removing the copy in Q1 by dequeuing it.

The problem with this solution, of course, is that while we are transferring **x**, other threads can observe that at some point **x** is stored in both queues.

One solution could be to wrap the two queues into a pairwise structure of two queues, which is protected by a new lock.

The problem with this is that other threads can still access the individual queues. If the queues are hidden, existing threads that rely on the queues break.

Locking has many Problems

With this list of problems, it comes to no surprise that system designers have been looking for alternatives to locks.

Alternative: Transactions

One such alternative is to group operations that one would like to execute together into so-called transactions.

A transaction would be marked with a beginning and an end, in this case with some simplified notation, and the operations in-between would be executed together.

Maybe one would like some sort of a way to abort a transaction, for which an exception mechanism would come in handy.

If one could execute these transactions atomically, then this would greatly simplify the implementation of the previous two-queue problem.

One would define a transaction, the bracket, the dequeuer, and enqueue operation of the Object to transfer from one queue to the other, by a pair of transaction begin and end instructions. This would require the dequeue and enqueue operations to be implemented using transactions as well. If they were, it would be no problem to nest these transactions together. In this example, the two-queue transaction would contain one transaction that dequeues the object and another transaction to enqueue the object again.

In order to keep the following discussion simple, we assume that there are two operations that we can do on objects. We can read the current value of the object, and we can write a new value to the object. We will see later that both operations can trigger a **TransactionAbortException** when they would lead to a conflict with another transaction.

Atomicity Transaction-Style: Serializability

For this all to work, we would like for the transactions to execute atomically.

By atomically, we mean that the operations in a transaction either execute all at once, or none of the operations execute at all.

The way this is achieved in transaction is to execute the transaction such that they maintain serializability of the operations.

We observe that, since transactions are groups of operations, any form of atomicity is about how the operations in the group are executed.

One way to organize the execution of operations across multiple instructions are so-called "**serial**" executions, where the operations are scheduled one transaction at a time. As a result, the transactions are executed, each to completion, one after another.

One serial execution would be to execute all operations of transaction T1 before the operations of T2, or vice versa, with T2 executing before T1.

If we only have to deal with serial execution, then providing atomicity is simple. Whenever a transaction aborts, undo its effects by rolling it back, and start the next transaction.

- Cont.

We have learned about the detrimental effects of serial execution on system performance earlier, and so we are interested in some way to interleave the executions of instructions that still allows for some form of atomic execution while increasing concurrency and system performance.

One family of such executions is called "**serializable**" execution. We say that execution of a set of transactions is "**serializable**" if it has the same effect as at least one serial execution, where transactions would have executed one at a time. In other words; an execution is serializable if it is equivalent to at least one serial execution.

By this we mean that, say, two transactions can well executed in an interleaved fashion, but at the end, the effect should be that of one transaction executing after the other, either T1 followed by T2 or the other way.

Serializability

Let's look at an example, we have three transactions, T1 to T3, where each transaction sets **x** to zero and then increments **x** by either 1, 2, or 4.

Let's look at a few executions of these three transactions.

In the first example, T1 executes to completion, followed by T2, and then by T3.

Since this execution is serial, it is clearly serializable.

In the first example, T1 executes to completion, followed by T2, and then by T3.

Since this execution is serial, it is clearly serializable.

Let's look at a different execution, where first T1 and T2 each set **x** to 0; then T1 increments **x** to 1, then T3 starts and sets **x** back to zero, T2 completes by incrementing **x** to 2, and T3 finally incrementing **x** by 4, which sets **x** to 6.

This is clearly not serializable, because there is no serial execution where **x** would have value 6 at the end.

How to ensure Serializability

We recall that the beauty of serializable executions is that they have the same effect as serial execution. In a serial execution, no transaction ever observes the "**innards**" so-to-say, of another transaction. So, as long as transactions don't abort, they execute as if they were atomic.

The question now is, how to ensure that the execution stays serializable.

As a side-note, it is easy to enforce serializability with locks, using so-called two-phase locking, but locks are exactly what we try to abort.

So instead, we let every transaction go without attention to serializability.

While we do so, we keep track of all objects that we access.

When a transaction wants to commit, typically at the end of the transaction, we check for potential conflicts with other transactions.

If there are conflicts, we abort.

The approach is to assign a unique timestamp to each transaction. This can be easily done with a fetch-and-add or some other atomic CPU instruction.

As the transaction executes, we have to make sure that it has the same effects as that serial execution that executes the transactions in order of increasing time stamps.

The question of course is, how can this be done?

Ensure Serializability: Scenario 1

Let’s make a few observations as we look at some scenarios.

We have two transactions, one with **timestamp 5**, and the other with **timestamp 6**. Think of it as “**5pm**” and “**6pm**”.

We have a single object, which is initialized to zero.

The transactions keep executing, accessing other objects.

At some point, transaction T1 writes a “**1**” to Object A.

At some point later, Transaction T2 reads the value, which is “**1**” from Object A.

The question is: Is this consistent with an execution where T1 executes before T2?

The answer is “**Yes**”, as T1 writes the “**1**” at 5pm and T2 reads that new value at 6pm, which is after T1 has completed.

The transactions continue their execution.

Ensure Serializability: Scenario 2

Let’s look at another scenario, with the same transactions and Object A.

The transactions start executing, until T2 reads from Object A, which has value 0.

A bit later, T1 writes a “**1**” into Object A.

The question now is whether this is consistent with a T1-before-T2 serial execution.

It is not consistent, because T1 overwrites a value, which has been previously read by a transaction with a later timestamp. T2 should have read a “**1**”, but the value of object A was 0 at that time.

Ensure Serializability: Scenario 3

In another scenario, the two transactions execute until T2 writes a “**1**” to A.

Later, T1 reads the new value from A.

Is this consistent with an execution where T1 executes before T2?

No, it is not, because T1 reads a value at “**5pm**” that in a serial execution would not have been written until “**6pm**”, well after T1 completed.

Ensure Serializability: Scenario 4

Let’s look at a last example.

The two transactions execute, until T2 writes a new value into A.

Then we continue until a new transaction, T3 with timestamp 8, reads from A.

This read operation is consistent with T2 executing before T3.

We then continue and T1 attempts to write a new value into A.

Is this write consistent with the serial execution T1, T2, T3?

No, it is not because T1 is writing a value that in the serial execution would have been overwritten by T2.

We can simply ignore this write operation, given that it would be overwritten by T2 anyway and the transactions can continue.

Ensure Serializability: Scenario 1

These observations lead to a mechanism that allows us to ensure serializability.

For this, we associate each object, here Object A, with two timestamps which we call a READ time stamp, and a WRITE time stamp.

Whenever a transaction reads from the object, we set the read time stamp of the object to the TS of the transaction.

Whenever a transaction writes, we do the same with the WRITE TS.

The transactions start, and when T1 writes to A, we check whether the operation makes sense. It does because A has not been written or read yet. We infer this from the read and write TSs.

We update the WRITE TS of object A and continue. T2 now reads from A.

We check whether this read makes sense. It does, because A would have been written in the past in a serial execution. We infer this from the WRITE TS of A.

We update the READ TS of A and continue.

Ensure Serializability: Scenario 2

Let’s apply this approach to the second scenario.

The transactions execute until T2 reads from A.

This operation is ok, because nobody has read or written A yet. T2 reads the initialization value for A.

We update the READ TS of A, and continue. T1 now attempts to write a new value to A.

We check the READ TS of A and detect that the READ TS of A is larger than T1's TS. In the serial execution, T1 followed by T2, the current value of A would have been read by T2 after T1 would have written the new value. This write would therefore be inconsistent with the serial execution.

We must abort this transaction, and undo all of its effects. We note that if other transactions depend on this T1, they must be aborted as well. This can lead to cascaded aborts.

We assume in this example that T2 does not depend on T1, and T2 therefore continues.

Ensure Serializability: Scenario 3

Let's look how this approach detects the problem in the third scenario.

The transactions execute until T1 writes a "**1**" to A.

Object A's time stamps indicate that A has not been written or read yet. So this operation is ok.

We update the write time stamp and continue. Now T1 reads from A.

By looking at A's time stamps we see that the WRITE TS of A is larger than T1's TS. This means that in the serial execution T1 followed by T2, T2 would have written this value to A after T1 terminated. This read operation is therefore inconsistent with the serial execution T1 followed by T2 and we abort T1.

T2 can continue.

Ensure Serialization: Scenario 4

Let's look at the last scenario.

The transactions execute until T2 writes a new value to A.

This is ok, because we see from A's time stamps that A has not been accessed yet.

We set A's WRITE TS to T2's TS and continue. The new transaction T3 reads from A.

T3's timestamp, which is 8, is larger than the READ or the WRITE TS of A, which makes the read operation consistent with the serial execution T1 followed by T2, followed by T3.

We update the READ TS of A and continue. T1 now attempts to write to A.

The read TS of A is larger than T1's TS, which means that T1 would have to be aborted. Fortunately, the WRITE TS of A is larger than T1's TS as well, which means that in a serial execution T2 would have overwritten T1's value anyway.

This means that if we simply IGNORE this write, we don't need to abort T1 and all transactions continue.

Timestamp-based Optimistic Conc. Control

These examples showed how we can make use of these time stamps to formulate an algorithm to ensure that the execution of transactions remains serializable. The general term for this approach is called "**optimistic**" concurrency control, as opposed to "**lock-based**" concurrency control.

Each object is tagged with a read-time and a write time.

Whenever we read or write from or to an object, we follow these rules.

A transaction cannot read a value if in the serial execution the value would be written after the transaction terminates.

We check this by comparing the TS of the transaction with the write time of the object. If the write-time is larger, we abort the transaction and have it restart with a later TS.

Similarly, a transaction cannot WRITE a value to an object if the current value would later, in the serial execution, be read by another transaction.

We check this by comparing the TS of the writing transaction with the read time of the object. If the read time is later, we abort the write, and with it the transaction. We restart the transaction with a later time stamp.

There are two comparatively innocuous conflicts.

- **The first is when the transaction reads a value that, in the serial execution would be read later by another transaction. This is ok.**
- **The second is when the transaction tries to write to an object that would be overwritten later in the serial execution, that is, the write time is larger than the writing transactions TS. In this case, we can simply ignore the write operation.**

This is the complete algorithm to ensure serializability of the execution of transactions.

Timestamp-Based Conc. Control

We can formulate this in something that resembles more of a pseudo-code.

We call the TS of the transaction **T**, and the read and write times of the objects **TR** and **TW**.

We perform a READ if T is larger or equal to TW and then we update TR.

Similarly we perform the WRITE if T is larger than both TR and TW, and then we update TW.

We ignore the WRITE if T is between TR and TW and we abort the transaction otherwise.

We note that this is slightly more conservative than our discussion of Scenario 4, where we reasoned that we can ignore the write operation despite the READ by T3.

Dealing with Aborting Transactions

One problem that we have not discussed is how to deal with aborted transactions.

This is handled by having transactions be prepared for aborts.

For this, transactions manipulate objects inside a private workspace only.

Typically, transactions maintain a "**read set**" of objects, which are objects that have been read.

Similarly, they maintain a "**write set**", which are objects that have been written to by the transaction.

When the transaction is "**done**", we say that the transaction "**commits**", and then writes the objects into global memory. In some implementations of transactions, objects are briefly locked while they are being committed to memory.

If now a transaction aborts, it releases the read and write sets. If other transactions depend on this transaction's operation, they may have to abort as well.

Atomic Transactions Conclusion

We have come to the end of this lesson on atomic transactions.

We started by enumerating a whole series of problems with locks, starting from reduction of concurrency to deadlocks and to their shortcomings in building larger systems.

We then learned about transactions. Atomic transactions eliminate many of the problems typically associated with locks. In particular, they greatly support composability if implemented correctly.

In order for transactions to provide atomicity, their operations have to be executed in what is called serializable order, meaning that the order of execution of operations in multiple instructions must mimic an execution where the transactions execute one at a time.

We learned how this can be done by associating time stamps with transactions and data objects. This is an example of so-called optimistic concurrency control.

We then briefly mentioned how to deal with aborting transactions.

It is important to keep in mind that many of the benefits of transactions, in particular deadlock-freedom and composability, derive from the time-stamp based implementation of the serializability module. Many products that provide transactions, in particular many data base systems, use lock-based methods to enforce serializability. This can lead to all kinds of deadlocks if the programmer is not aware of this.

We hope that you enjoyed this lesson on atomic transactions. Transactions are often associated with data base systems, but you should be aware now that the concept of transactions is much broader and has applications even in the design of operating systems.

Thank you for watching.