Introduction to Synchronization

Synchronization: Review

Hello, and welcome to this introductory lesson on synchronization.

We will explore, based on several real-world cases, how uncontrolled execution of concurrent threads can lead to concurrency problems, which present themselves as errors.

We will focus on two types of such problems, which are called, by some authors, atomicity violations and ordering violations.

We will see that atomicity violations occur when the execution of some code is interrupted by execution of other code in a fashion that renders the execution incorrect.

Ordering violations happen when the code executes in a order that has not been considered by the programmer and that may lead to errors.

Synchronization is needed to enforce a certain ordering of the execution of the threads that is in line with the expectations of the programmer.

One such ordering is represented by critical sections. We will learn that critical sections are portions of code that must be executed in a mutually exclusive fashion, that is, by at most one thread at a time. We will see that critical sections are ideally suited to address the atomicity violation problem. We will wrap up by giving a very brief outlook on how critical sections can be realized using a lock semantics.

In this lesson we set the stage for the next lessons, where we will study how synchronization mechanism are implemented in operating systems and in user-level thread libraries.

Let's begin.

Atomicity Violation: MySQL Bug Example

We start by looking at a few concurrency errors that occur in the wild. We borrow these from a popular study that examined concurrency errors in a large number of open source projects. The concurrency errors were categorized into (1) so-called atomicity violation errors, (2) order violation errors, and (3) deadlock errors.

Let's look at an example of atomicity violation first. This error was found in the well-known MySQL data base software.

In this example, we have the following piece of code, which checks whether the process information of a thread control block is null. If it is not, it does something with the process information.

The implicit assumption that the programmer made when writing this code, was that the process information would be non-null during the execution of the body of this code. In sequential, that is, non-concurrent code, it is easy to ensure that this assumption stays valid: simply do not modify the pointer to the process information once it has been checked. This becomes significantly more complicated when the program has multiple threads, for example.

If the process information is set to NULL in another part of the code, which is executed by another thread, then the assumption made by the programmer may become invalid, and errors occur.

Let's look at an example of how things could go wrong. We have two threads, Thread 1 and Thread 2.

Thread 1 has the CPU and is running. Thread 2 is either blocked or ready.

Now Thread 1 checks whether the process information is NULL, and continues executing.

After some time Thread 1 gets either blocked or preempted, and gives up the CPU. Thread 2 gets to execute.

Thread 2, maybe after some execution of something else, sets the process information pointer to NULL.

At some point later, Thread 2 gives up the CPU, and – maybe after some other threads – Thread 1 acquires the CPU and continues its execution.

Thread 1 now calls the function **fputs** with the process information pointer as argument. The pointer is now NULL, which causes a run-time exception, or some other form of error.

This error is caused by what is called an *atomicity violation*, where a piece of code was intended to be executed atomically, but was not.

We will look a bit more in detail at the definition of atomicity, but for now we say that the execution of a sequence of instructions happened *atomically* if either all instructions happen at once, or none of the instructions happened at all. One side effect of this is that an atomic sequence of instructions cannot be interrupted. Back to our example, if this sequence of operations were atomic, its execution could not be interrupted by another thread, and the assumption about the process information being non-NULL would be true all the time.

Another Atomicity Violation

Atomicity violations can happen whenever we manipulate data structures.

Let's take the insertion of an element into a doubly-linked list as an example.

We have a section of the doubly-linked list, with a pointer to the current element.

Let's assume that we want to insert into the doubly-linked list an element *new* after the current element. This is done in four steps, which we remember from our elementary data structures course.

First, we point the **new.next** pointer to the next element in the list.

Then we we point the **new.previous** pointer to the current element.

We point the **current.next** pointer to the new element, and finally we point the previous pointer of the next element to the new element. Until now no surprises.

Let's see what can go wrong in a multithreaded execution of this code. We have one doubly linked list, and two threads, which

- both want to insert one element each at the current position.
- Thread 1 is running.
- And it performs the first two
- of the 4 steps to insert element **new1** into the list.
- For some reason Thread 1 now gets blocked or preempted, and Thread 2 gets to execute. (5)
- and it starts inserting its element
- **new2** into the list.

It completes the insertion, after the four steps. The element **new2** is now part of the doubly-linked list.

After a while Thread 1 continues with its execution. (10)

Now Thread 1 connects the next pointer of the current element to element **new1**, and does the same with the previous pointer of the next element. Which in effect cuts out again element **new2** from the doubly linked list. This error occurred because the 4 insertion steps were supposed to be executed atomically, and Thread 2 interrupted the insertion steps of Thread 1.

Order Violations Example

Let's look at another form of concurrency errors. These are called *order violations* because the execution violates an assumption that the programmer made about the order in which code was supposed to be executed.

In this, rather subtle, example, the code creates some form of new thread and passes the function **mMain** as the thread function pointer. The function **PR_CreateThread** returns a pointer to the thread control block of the newly-created thread.

In some other part of the code, the function **mMain** is defined. Inside this function, we set the local variable to the state in the thread control block.

In order to appreciate what can go wrong when we execute this code, we need to know that the variable **mThread** is not set to the new thread control block immediately when the new thread is created, but rather, AFTER, the new thread has been created. In this example, the pointer to the thread control block is returned in a register, and after the create function returns, we assign the register value to the variable. This gives raise to all kind of concurrency errors, as we will see next.

Let's assume that we have Thread 1 that is running. It calls function **PR_CreateThread** to create a new thread. The function creates the thread, and returns a pointer to the new thread control block in some register.

The execution of Thread 1 is interrupted, and – possibly after a few other threads have executed – the newly created thread gets to execute.

The new thread starts executing the thread function, which contains the instruction sequence to set the local variable **mState** to the state stored in the thread control block **mThread**.

This of course causes a NULL-pointer exception, because the value for the pointer **mThread** has not been defined yet.

This is an – admittedly subtle – example of an *order violation* error, where the desired order of execution of instructions is flipped. Some sequence A of instructions should be executed before some other sequence B. If this is not enforced during execution, a so-called *order violation* can occur.

We will discuss methods to enforce desired orders of execution to avoid order violations in a later lesson. In this lesson we set the stage to later describe methods to avoid atomicity violations.

Avoiding Atomicity Violations: Critical Sections

Let's assume that we have some code to be executed in a multithread environment. Let's also assume that, during some portion of the code, marked here, one or more invariants are temporarily violated. For example, we may be inserting or deleting an element from a data structure, and during this portion of the code the data structure is inconsistent.

Whatever the reason, only one thread is allowed to execute this section of the code at any given time. We call such a portion of code a *critical section*.

The execution of code in a critical section has to be mutually exclusive, meaning that at most one thread can at any given time execute code of a critical section.

As we have seen in the previous examples, we need something that enforces the mutually-exclusive execution. In particular, the code has to be written such that, in order to start executing code of a critical section – we say *enter* a critical section – some special code needs to be executed. This code makes sure that only one thread is executing inside the critical section at any time.

Typically, a thread that leaves a critical section executes some other code, which may inform the system that now there nobody in the critical section, and the next thread can enter the critical section. Together, these two pieces of code are said to "protect" the critical section.

After a thread leaves the critical section it may execute some other code, for which there are no restrictions about how many threads can executed it concurrently.

This may all be part of a larger portion of code, which we represent here as an infinite loop, just to make the code a bit more interesting.

Let's look at an example.

Let's look at critical sections in practice, using two threads.

At the beginning of our example, thread 1 is running, and it just executes the code needed to enter a critical section. Thread 2 is either ready or blocked.

Thread 1 then executes the code in the critical section.

And then exits the critical section by executing the code to do just that.

Some time later, Thread 1 gives up the CPU because it is either blocked or is preempted. Thread 2 starts executing.

Thread 2 executes the code to enter the critical section. (5)

It is now inside the critical section and executes the critical section code.

It then exits the critical section.

Some time later it gives up the CPU, and at some point Thread 1 acquires the CPU and gets to execute the non-critical section of the code.

The first example was not particularly interesting, so let's look at a slightly different execution scenario of the same code.

We have the same two threads, and again Thread 1 is running, while Thread 2 is either ready or blocked. Thread 1 executes the code to enter the critical section.

It succeeds because nobody else is in the critical section.

While Thread 1 is in the critical section it is either blocked or preempted and therefore gives up the CPU. Thread 2 is now running.

At some point in its execution now Thread 2 tries to enter the critical section. The code to enter the critical section detects that there is somebody else in the critical section, and Thread 2 is blocked, until whoever is in the critical section leaves.

Because Thread 2 is blocked, it gives up the CPU, which at some point later is acquired by Thread 1 again. (5)

Thread 1 leaves the critical section, and maybe starts executing its remainder section.

If at some time later now it gives up the CPU and thread 2 takes over, Thread 2 can now try to enter the critical section again. Unless a third thread has entered the critical section, Thread 2 now succeeds, and it can execute the critical section.

Let's look at a third example, again with the same two threads.

Thread 2 is running.

It enters the critical section, and leaves it again.

Some time later it is either blocked or preempted, and thread 1 eventually takes over.

Thread 1 successfully enters the critical section.

After that, it is either blocked or preempted, and thread 2 takes over again. (5)

Thread 2 executes code of its remainder section. This poses no problem because it does not matter – for the execution of the remainder section – whether or not Thread 1 is in the critical section.

When Thread 1 takes over again at some point later, it exits from the critical section and continues.

Non-Consecutive Critical Sections

Until now we have had a single segment of consecutive code be a critical section. Critical sections don't need to be consecutive, as the following example shows.

Let's assume that there is some invariant that is violated during this portion of code.

It must be therefore be treated as a critical section. We denote it by CS1 to indicate that this is the first segment of critical section code.

A little bit later in our code there may be another piece of code that temporarily violates the same – or a related – invariant.

We treat it as being part of the same critical section as the one above. We denote it as CS2 to indicate that it is the second segment of the critical section code.

We could protect the two critical section segments with one bracket of code to enter and exit both critical sections. In this way we would be treating the two critical section segments as one large consecutive critical section. (5)

But it may be better for concurrency and overall system performance to treat the two critical section segments separately, and basically "break" the critical section in two.

This would allow us to treat whatever code is between the critical section as a non-critical section.

Let's look at an execution scenario with the two critical section segments.

Thread 1 is running, and enters the first critical section segment.

After a while, Thread 2 gets to execute and attempts to enter the critical section. It gets blocked.

After a while (remember that we have other threads in the system) Thread 1 gets the CPU again and exits the critical section. It may also execute the non-critical section.

It then gives up the CPU, and Thread 2 eventually gets to execute.

Thread 2 enters the critical section and executes the first critical section segment.

When it gives up the CPU and at some time later Thread 1 acquires the CPU, Thread 1 may attempts to enter the second segment of the critical section. Because this segment is part of the same critical section that Thread 2 is in, Thread 1 gets blocked.

Thread 2 at some point later takes over the CPU and at some time leaves the critical section.

When Thread 1 acquires the CPU again, it can now enter the second segment of the critical section and continue.

Multiple Critical Sections

Code can also have multiple, distinct, critical sections. We show this with the following example.

Let's assume that some portion of the code violates an invariant.

We treat this code as Critical Section A.

Some other portion of code may violate some other, totally unrelated invariant.

We treat this code as a critical section as well, but a different critical section than A. We call it Critical Section B.

Critical Section A is protected by code that ensure mutual exclusion in A, while critical section B is protected by code the ensure mutual exclusion in B only.

As usual we may have non-critical sections as well.

Let's illustrate how this works with an example. Task 1 is running.

It successfully enters and executes critical section A.

It gets interrupted and yields the CPU, which at some point is acquired by Thread 2.

Thread 2 now attempts to enter critical section B.

It succeeds because it critical section B is free. The fact that critical section A is busy is irrelevant because A and B are segments of unrelated critical sections.

Some time later thread 1 gets to run again, (5) it leaves critical section A, and after some time attempts to enter critical section B. Here it gets blocked because thread 2 is in critical section B.

Thread 1 is blocked, and Thread 2 to gets to execute.

It leaves critical section B and after some time yields the CPU. Thread 1 is now unblocked and when it acquires the CPU again, it can enter and execute critical section B. (10)

We have done quite a lot of gymnastics with critical sections and we are ready to conclude this lesson.

Before we do so, we would like lift what for some may be a bit of a mystery regarding the entry and exit code for critical sections. The easiest way to think of this code is of course in terms of locks. In order to enter critical sections, you must acquire the lock that protects the critical section. If somebody else owns the lock, you wait. When you exit the critical section, you make the lock available to others.

In this particular case of two critical sections A and B, we have two locks, called *lock_A* and *lock_B*.

These locks support two operations, called "lock" and "unlock".

A thread acquires a lock by calling the function lock. This function returns immediately if nobody else owns the lock, and blocks otherwise. A thread releases a lock by calling unlock, after which the lock is available to other threads. If another thread is blocked waiting for the lock, it gets unblocked, is assigned a lock, and returns from the lock function.

We can now very easily replace the enter and exit code for the critical sections with those lock functions, as this example shows. If you pause the video and check the code carefully, you will see that the lock and unlock functions satisfy the requirements for the enter and exit code that protect the critical sections.

This example gives a good indication how locks can be used to implement critical sections. In the next lesson we will explore how locks are implemented in the operating system and in user-level thread libraries.

Synchronization: Review - Summary

We have come to the conclusion of this introductory lesson on synchronization.

We pointed out in an earlier lesson that one of the disadvantages of thread-based concurrency is that inter-task invariants must be enforced at all times. Since threads, during their normal execution, occasionally occasionally violate invariants temporarily, the only way to enforce these invariants across threads is through careful prevention of certain interleavings of execution of the tasks. This is called synchronization.

In this lesson we looked at a number of real-world examples of what can go wrong in multithreaded systems.

We identified two typical types of problems: when atomicity is violated and when when a particular required or assumed ordering of execution is violated.

We explored how the type of atomicity requirements that we encountered in these examples can be represented through critical sections, which require mutual exclusive access, and we saw how mutual exclusion, and therefore atomicity, is enforced through enter and exit code that protect critical sections.

Finally, we learned how enter and exit code for critical sections can easily be represented using a lock semantics.

In the next lessons we will learn how locks are implemented in operating systems and in user-level thread libraries.

We hope that you enjoyed this introductory lesson into synchronization.

In this lesson we focused on the need and the looks of synchronization. This will put you in a good position to appreciate as we explore the implementation of synchronization primitives such as locks in the lessons that follow.

Thank you for watching.