— WRITTEN BY TRIANGLES ON MAY 10, 2019 • UPDATED ON MAY 12, 2019 • ID 72 —

Introduction to thread synchronization

A look at one of the most popular ways of concurrency control in a multithreaded application.

OTHER ARTICLES FROM THIS SERIES

- A gentle introduction to multithreading — *Approaching the world of concurrency, one step at a time.*
- Lock-free multithreading with atomic operations — *Synchronizing threads at a lower level.*

As emerged from my previous introduction to multithreading, writing concurrent code can be tricky. Two big problems might emerge: data races, when a writer thread modifies the memory while a reader thread is reading it and race conditions, when two or more threads do their job in an unpredictable order. Luckily for us there are several ways to prevent these errors: in this article I will take a look at the most common one known as **synchronization**.

What is synchronization

instructions is called **critical section**. You want to make sure that critical sections are executed *atomically*: as defined in the previous episode, an atomic operation can't be broken into smaller ones, so that while a thread is executing it no other thread can slip through;

2. **ordering** — sometimes you want two or more threads to perform their job in a predictable order, or put a restriction on how many threads can access a specific resource. Normally you don't have control over these things, which might be the root cause of race conditions. With synchronization you can orchestrate your threads to perform according to a plan.

Synchronization is implemented through special objects called **synchronization primitives** provided by the operating system or any programming language that supports threading. You then make use of such synchronization primitives in your code to make sure your threads don't trigger data races, race conditions or both.

Synchronization takes place both in hardware and software, as well as between threads and operating system processes. This article is about synchronization of software threads: the physical counterpart and process synchronization are fascinating topics that will surely get some love in a future post.

Common synchronization primitives

The most important synchronization primitives are **mutexes**, **semaphores** and **condition variables**. There are no official definitions for these terms, so different texts and implementations associate slightly different characteristics with each primitive.

Operating systems provide these tools natively. For example Linux and macOS support **POSIX threads**, also known as **pthreads**, a set of functions that allows you to write safe multithreaded applications. Windows has its own synchronization tools in the C Run-Time Libraries (CRT): conceptually similar to POSIX threads functions but with different names.

Unless you are writing very low-level code, you usually want to employ the synchronization primitives shipped with the programming language of your choice. Every programming language that deals with multithreading has its own toolbox of synchronization primitives, along with other functions to fiddle around with threads. For example Java provides the `java.util.concurrent` package, modern C++ has its own `thread` library, C# ships the `System.Threading` namespace and so on. All these functions and objects are based upon the underlying operating system primitives, of course.

There are many other synchronization tools around. In this article I'll stick to the three mentioned above, as they act as a foundation often used to build more complex entities. Let's take a closer look.

Mutexes

A **mutex** (**mut**ual **ex**clusion) is a synchronization primitive that puts a restriction around a critical section, in order to prevent data races. A mutex guarantees *atomicity*, by making sure that only one thread accesses the critical section at a time.

Technically, a mutex is a global object in your application, shared across multiple threads, that provides two

If another thread jumps in and tries to lock a locked mutex, the operating system puts it to sleep until the first thread has finished its task and has unlocked the mutex. This way only one thread can access the critical section; any other thread is excluded from it and must wait for the unlock. For this reason a mutex is also known as a **locking mechanism**.

You can use a mutex to protect simple actions like a concurrent read and write of a shared variable, as well as bigger and more complex operations that need to be executed by one thread at a time, such as writing to a log file or modifying a database. Anyway, the mutex lock/unlock operations always match the boundaries of the critical section.

Recursive mutexes

In any regular mutex implementation, a thread that locks a mutex twice causes an error. A **recursive mutex** allows this, instead: a thread can lock a recursive mutex multiple times without unlocking it first. However no other thread can lock the recursive mutex until all the locks held by the first thread have been released. This synchronization primitive is also known as **reentrant mutex**, where **reentrancy** is the ability to call a function multiple times (i.e. to enter it again) before the previous invocations are over.

Recursive mutexes are difficult to work with and are error-prone. You have to keep track of which thread has locked the mutex how many times and make sure the same thread unlocks it completely. Failing to do so would leave locked mutexes around with nasty consequences. Most of the time a regular mutex is enough.

Reader/Writer Mutexes

As we know from the previous episode, multiple threads can concurrently read from a shared resource without harm as long as they don't modify it. So why bother locking a mutex if some of your threads are operating in "read-only" mode? For example consider a concurrent database that is frequently read by many threads, while another thread seldomly writes updates. You certainly need a mutex to protect the read/write access, but most of the time you would end up locking it just for read operations, preventing other reading threads to do their job.

A **reader/writer mutex** allows *concurrent* reads from multiple threads and *exclusive* writes from a single thread to a shared resource. It can be locked in *read* or *write* mode. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released.

Semaphores

A **semaphore** is a synchronization primitive used to orchestrate threads: which one starts first, how many threads can access a resource and so on. Like a street semaphore regulates the traffic, a programming semaphore regulates the multithreading flow: for this reason a semaphore is also known as a **signaling mechanism**. It can be seen as an evolution of a mutex, because it guarantees both *ordering* and *atomicity*. However in a few paragraphs I will show you why using semaphores for atomicity only is not a great idea.

Technically, a semaphore is a global object in your application, shared across multiple threads, that contains a *numeric counter* managed by two functions: one that increases the counter, another one that decreases it. Historically called `P` and `V`, modern implementations use more friendly names for those functions such as

choose that number according to your needs. Then, a thread that wants to access a shared resource calls `acquire`:

- if the counter is *greater than zero* the thread can proceed. The counter gets reduced by one right away, then the current thread starts doing its job. When done, it calls `release` which in turn increases the counter by one.

- if the counter is *equal to zero* the thread cannot proceed: other threads have already filled up the available space. The current thread is put to sleep by the operating system and will wake up when the semaphore counter becomes greater than zero again (that is when any other thread calls `release` once its job is done).

Unlike a mutex, *any thread can release a semaphore*, not only the one that has acquired it in the first place.

A single semaphore is used to limit the number of threads accessing a shared resource: for example to cap the number of multithreaded database connections, where each thread is triggered by someone connecting to your server.

By combining multiple semaphores together you can solve thread ordering problems: for example the thread that renders a web page in your browser must start after the thread that downloads the HTML files from the Internet. Thread A would notify thread B when it's done, so that B can wake up and proceed with its job: this is also known as the famous Producer-Consumer problem.

Binary semaphores

A semaphore whose counter is restricted to the values 0 and 1 is called **binary semaphore**: only one thread at a time can access the shared resource. Wait: this is basically a mutex protecting a critical section! You can actually replicate the mutex behavior with a binary semaphore. However there are two important points to keep in mind:

1. a mutex can be unlocked only by thread that had locked it first, while a semaphore can be released from any other thread. This could lead to confusion and subtle bugs if what you want is just a locking mechanism;

2. semaphores are signaling mechanisms that orchestrate threads, while mutexes are locking mechanisms that protects shared resources. You should not use semaphores to protect shared resources, nor mutexes for signaling: your intent will be more clear to you and to anyone who will read your code.

Condition variables

Condition variables are another synchronization primitive designed for *ordering*. They are used for sending a wake up signal across different threads. A condition variable always goes hand in hand with a mutex; using it alone doesn't make sense.

Technically, a condition variable is a global object in your application, shared across multiple threads, that provides three functions usually called `wait`, `notify_one` and `notify_all`, plus a mechanism to pass it an existing mutex to work with (the exact way depends on the implementation).

A thread that calls `wait` on the condition variable is put to sleep by the operating system. Then another thread that wants to wake it up invokes `notify_one` or `notify_all`. The difference is that `notify_one` unfreezes only

A emits a signal when it's done so that thread B can start its job.

Common problems in synchronization

All the synchronization primitives described in this article have something in common: they put threads to sleep. For this reason they are also called **blocking mechanisms**. A blocking mechanism is a good way to prevent concurrent access to a shared resource if you want to avoid data races or race conditions: a sleeping thread does no harm. But it can trigger unfortunate side effects. Let's take a quick look at them.

Deadlock

A **deadlock** occurs when a thread is waiting for a shared variable that another thread holds, and this second thread is waiting for a shared variable that the first thread holds. These things usually happen when working with multiple mutexes: the two threads remain waiting forever in an infinite circular loop: thread A waits for thread B which waits for thread A which waits for thread B which...

Starvation

A thread goes in **starvation** mode when it doesn't get enough love: it remains stuck indefinitely in its sleep state while waiting for access to a shared resource that is continuously given to other threads. For example a poorly designed semaphore-based algorithm might forget to wake up one of the many threads behind the waiting line, by giving precedence only to a subset of them. The starving thread would wait forever without doing any useful work.

Spurious wake-ups

This is a subtle problem that comes from how condition variables are actually implemented in some operating systems. In a **spurious wake-up** a thread wakes up even if not signaled through the condition variable. That's why most synchronization primitives also include a way to check if the wakeup signal really comes from the condition variable the thread is waiting on.

(high priority) is blocked by the thread that displays the interface (low priority), resulting in a bad glitch through your speakers.

What's next

All these synchronization problems have been studied for years and many solutions, both technical and architectural are available. A careful design and a bit of experience help a lot in prevention. Also, given the non-deterministic, (i.e. extremely hard) nature of multithreaded applications, people have developed interesting tools to detect errors and potential pitfalls in concurrent code. Projects like Google's TSan or Helgrind are just a few of them.

However, sometimes you want to take a different route and get rid of any blocking mechanism in your multithreaded application. This would mean to enter the **non-blocking** realm: a very low-level territory, where threads are never put to sleep by the operating system and concurrency is regulated through **atomic primitives** and **lock-free data structures**. It's a challenging field, not always necessary, which can boost the speed of your software or wreak havoc on it. But this is a story for the next episode...

Sources

Wikipedia - Synchronization (computer science)
Wikipedia - Reentrant mutex
Wikipedia - Reentrancy (computing)
Wikipedia - Semaphore (programming)
Wikipedia - Spurious Wakeup
Wikipedia - Priority inversion
Wikipedia - Deadlock
Wikipedia - Starvation (computer science)
Columbia Engineering - Synchronization primitives
StackOverflow - Definition of "synchronization primitive"
StackOverflow - Lock, mutex, semaphore... what's the difference?
StackOverflow - Why is locking a std::mutex twice 'Undefined Behaviour'?
Operating Systems: Three Easy Pieces - Concurrency
Jaka's Corner - Data race and mutex
Java 10 API specs - Class Semaphore
Oracle's Multithreaded Programming Guide - Read-Write Lock Attributes
Oracle's Multithreaded Programming Guide - Avoiding Deadlock
Just Software Solutions - Locks, Mutexes, and Semaphores: Types of Synchronization Objects
Just Software Solutions - Definitions of Non-blocking, Lock-free and Wait-free
Cppreference - std::shared˙mutex
Cppreference - std::condition˙variable
Quora - What is the difference between a mutex and a semaphore?
gerald-fahrnholz.eu - Using condition variables - the safe way
Politecnico di Milano - Thread Posix: Condition Variables
SoftwareEngineering - Spurious wakeups explanation sounds like a bug that just isn't worth fixing, is that right?
Android Open Source Project - Avoiding Priority Inversion