Asynchronous programming

Donnerstag, 30. November 2023 13:46

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

What is your understanding of Asynchronous programming?

Goal
This is a broad topic that is beyond any language.
There is no right - or wrong answer.
The aim is to get the feedback from candidate - what is candidate experience in dealing with this topic.
We can talk on this topic - through two aspects:
Asynchronous (parallel) programming
- Concurrency and memory model
- Language (C++) specific mechanisms
  - std::thread (std::jthread), std::async task / std::packaged_task
  - Synchronization primitives

Asynchronous (in time) programming
- Architecture: Event-driven
- Language (C++):
  - Coroutines
  - Sender/Receiver
- Design patterns: AOT - *Active Object Thread*, Publisher-Subscriber
- Libraries: Reactive library (RxJava/Android), Actors
- Android: Intents, Binder: AIDL Service-Client IPC

The both aspects are rather complementary - than stand side by side.

Asynchronous (parallel) programming

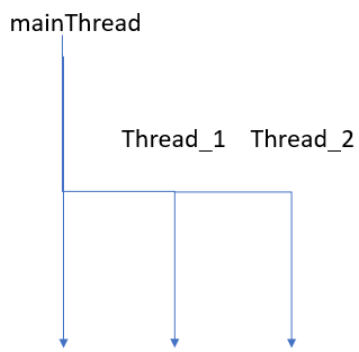
Asynchronous programming is about **having parallel execution flows**.
There is a *main* execution flow - that can branch on parallel execution flows, with certain limit.

What is the number of physical threads in the system?

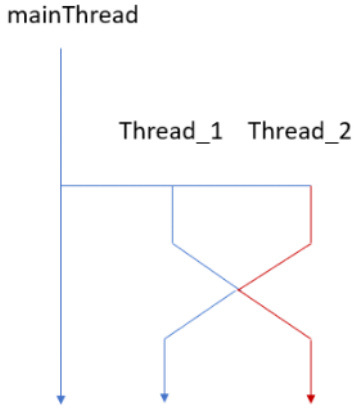
This is limited by the number of CPUs/Cores which determines the number of physical threads that we can have in system (*std::thread::hardware_concurency*)

What is difference between process and threads?

When we talk about parallel execution flows - we talk about the *threads*: not the processes (IPC).
The process is just the resource manager, a frame that provides the resources common to all threads that belong to the same process -
like the same address space: the memory.
The threads can run **without any intersection points** - we talk then about completely independent paths: a parallel programming in true meaning of the word
(@note GPU programming is out of scope).
These threads can do independent tasks, or they can do some cooperative work, but on the separate data range.
(With std library, for certain algorithms - starting with C++17, you can specify for the *execution policy* to be the parallel one, or there is even some
parallel version of certain algorithms: *std::accumulate* has *std::reduce* as parallel version)



If there is an intersection point - we talk about **concurrency**, since we have multi-thread access to the
same - shared piece of memory, for which we need to ensure some kind of synchronization: to
prevent the *data race*: having two or more threads that simultaneously access the same data, where
at least one thread try to modify it. Two (or more) threads may see the same data in a different - inconsistent state,
which is a path to the UB (Undefined Behavior).
We can say that data race is subset of the *race condition*: where the correctness of the code relies on an indeterministic
behavior of a program - like timing, or the specific sequence of operations, what may be challenged in multi-thread environment



To prevent this, there is a different kind of **synchronization primitives**:
Which synchronization primitives do you know (POSIX complain) and how they are typically used?

- mutex (std::mutex)
  Mutex (MUtual Thread EXclusion) serves to establish a *critical session* between lock/unlock calls that need to occur within the same thread.
  All other threads that acquire the very same mutex are blocked (suspended) - waiting to be waked up, at the point when thread which holds the mutex - releases it.
  Which thread will be waked-up and gain the mutex, depends on the *scheduling policy* and thread priority.
  More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/android/Thread__Attributes.pdf

  To prevent the *priority inversion*: having two threads synchronized on the same mutex, where the mutex is hold by the thread of the low priority that can be preempted by any thread of higher priority - leaving the high-priority thread locked: starving, waiting on the mutex to be unlocked.
  To prevent this, POSIX semaphores can be instantiated in a way to change the priorities of the threads synchronized on the same mutex: so that all threads become the priority of the highest priority thread among them.
  There is also *recursive mutex* (std::recursive_mutex) that can be locked multiple times by the same thread: it needs to be unlocked the matching number of times, so that another thread is eligible to acquire the mutex.

- condition variables (std::condition_variable)
  It's mechanism used along with mutex to model event notification - as a signaling mechanism.
  We have one or more threads, waiting on the same condition to be signaled by some other - single thread.
  The waiting thread that acquires the mutex checks whether the condition is satisfied - if not, it will atomically and automatically release the mutex and remained blocked on condition.
  The thread which sets the condition - after acquiring the mutex, signals that condition is met either only one thread (std::condition::*notify_one*), or all threads waiting on the same notification (std::condition::*notify_all*).
  To prevent **spurious wakeup**: the predicate (boolean) is used, so that we are sure that the thread waiting on condition is not woken by some arbitrary system interrupt - by inspecting this predicate.
  It's the same for premature (missed) notification: where thread T1 signals the condition before thread T2 is entered the waiting state: the predicate comes to rescue - will be evaluated afterwards as true.

  There is *future/promise* inter-thread communication channel that can be used for synchronizing the thread on the result of asynchronous task (std::async) - as alternative to std::condition_variable.
  Cons: single shot event

- semaphores (*std::counting_semaphore*<std::ptrdiff_t LeastMaxValue>)
  Part of the std library starting from C++20.
  Semaphores are normally used as tokens - to specify the number of available slots of some resource (memory/buffer), as internal counter (maximal) value.
  @note Usually, it's determined with pair (initValue, maxValue) - initValue: is the initial value of the semaphore, and maxValue is upper limit.
  As long as there are free tokens - as long as the semaphore counter is greater than 0 - the semaphore is unlocked.
  Each call that acquires semaphore (token) - decrement the internal counter.
  In the same way, each call that release the semaphore (token) - increment the internal counter (up to maximal value - depends on the implementation).
  Unlike mutex, semaphores are not tied to the thread where are acquired - the release can happen on another thread as well.
  <Example: **produce-consumer queue**>: https://godbolt.org/z/zzs1Ks4Pq

- latches (*std::latch*)
  Part of the std library starting from C++20.
  Unlike semaphores, latches are **downward counting** locking mechanism - typically, they serve to synchronize the waiting thread that depends on the jobs that have to be done by other threads, before the waiting thread can proceed.
  This means - as long as the internal counter is greater than 0 - the thread waiting on the latch (std::latch::wait) will be blocked.
  When all jobs are completed - and internal counter is decremented to 0 - the waiting thread will resume.
  <Example>: https://godbolt.org/z/rdKM3bzYG

- thread barriers (std::barrier)
  Part of the std library starting from C++20.
  It's the same counting-down synchronization mechanism as with latches - the only difference is that barrier is **reusable** after being unlocked: can be reinitialized and used again for inter-threads synchronization
  <Example>: https://godbolt.org/z/WKrGeddh6

What about lock-free programming?
atomic variables and new memory model introduced with C++11

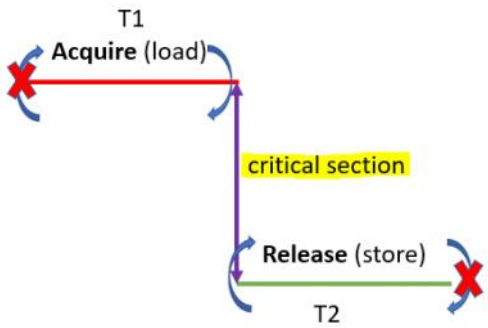
Lock-free programming

Atomic variables are the corner stone of the C++ memory model.
They guarantee that the **operation on atomic variables are atomic** - that they can't be interrupted by the threads in a way that different threads synchronized on the same atomic variable can observe the intermediate result: they can't.
They can see the value before, or after atomic operation - but not in between.

The **new memory model** introduced with C++11, reflects the modern - contemporary HWs with multicores architectures, equipped with layered memory.
Each CPU has its own cache - actually, they are different levels of cache.

- Each core of CPU has its own L1 cache (smallest and fastest)
- L2 cache is shared between the cores of the same CPUs, and finally the
- L3 cache is shared between the all CPUs (biggest and slowest)

Atomic variables are the way to have the synchronization between threads hosted on a different CPUs (or event cores: L2, L3), in a lock-free manner, by disabling the compiler optimization: reordering of the operations, so that non-atomic write operations: the side-effects of a one thread, are observable in another thread - synchronized on the same atomic variable.
This is accomplished by specifying the **memory ordering** (std::memory_order - cppreference.com) in atomic variable instantiation.
The **sequentially consistent** ordering is default - and the most rigid one: as if all atomic operations - on all threads, across the all CPUs, are seen in a single - global ordering.
On the other hand, **relaxed ordering** is the - as the name implies, the most relaxed one: it doesn't impose any ordering at all, except guarantee that the operations on atomic types (certain one, provided by the atomic interface itself: *store/load*, for integral types arithmetic one: incrementing: *fetch_add*, decrementing: *fetch_sub*) will be atomic.
Acquire-Release memory ordering is the one which establishes the *critical section* (similar to mutex).



This means, we have two (or more) threads synchronized on the same atomic variable.
The thread T1 which acquires the atomic value (loaded from the main memory) will observe all write operations in T2 thread that *happen-before* the releasing the atomic value: that are inside the critical section.
Acquire memory ordering means that no operations (write/read) can't be reordered by the compiler (optimization) by moving them BEFORE the acquire memory barrier - no operations can't leave the critical section.
This would violent the transient happens-before relationship.
Similar, for *Release* memory ordering - no operations that happen before release barrier can't be moved by the compiler AFTER the barrier.

Probably the most important operation with atomics - which is base for the many lock-free algorithms is compare_exchange_* - also known as *CAS* (*Compare And Swap*) idiom
It comes in two flavors(*):
- **compare_exchange_weak** that can spuriously fail (but it's more efficient)
- **compare_exchange_strong**
More on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Lock-free%20programming%2C%20part%201.pdf

```
template<std::integral T>
auto fetch_mul(std::atomic<T>& a, T val) {
  T expected = a.load(std::memory_order::relaxed);
  while(!a.compare_exchange_strong(expected,// passed by the reference
                        expected * val,// new value
                        std::memory_order::acq_rel,// memory ordering for load/store in case of success
                        std::memory_order::relaxed)// memory ordering for load in case of the failure
                        );
  return expected;
}
```

How it works?

It will compare the current value stored into atomic variable with the expected one - passed by the reference.
If the values are different - it means that another thread has updated the same variable: the expected will be updated with this new value.
If there is no new update on a repeated check - this result the updating the atomic variable with the multiplication (second argument) - as a new value, and check returns true.

With C++20, we got more:
- atomics that behaves as signalization mechanism (like std::condition_variable)
- atomic smart pointers: std::atomic<std::shared_ptr> (std::weak_ptr)
- atomic references

More on that: https://www.youtube.com/watch?v=c0I9nlpUH4o