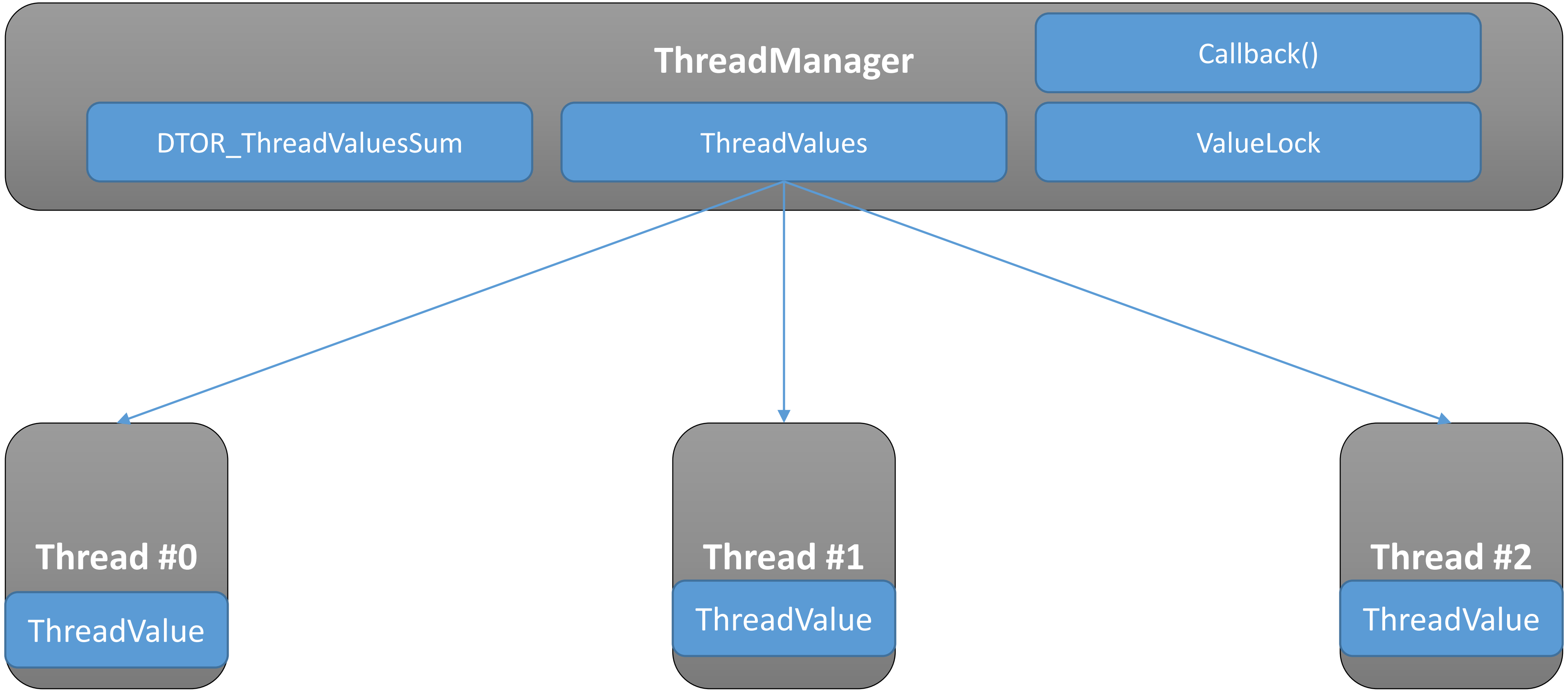
C++11 `ThreadManager` and `ThreadValue`

Managing thread-local storage from a single class

ThreadManager architecture



`ThreadValue` CTOR and DTOR

- `ThreadValue` is allocated as thread-local storage using C++11 `thread_local` qualifier.
- The constructor of `ThreadValue` appends the address of the latter object to the `ThreadManager::ThreadValues` vector and updates the manager, in a thread-safe way.
- The destructor of `ThreadValue` deletes the address of the latter object from the `ThreadManager::ThreadValues` vector after adding its `Value` to `ThreadManager::DTOR_ThreadValuesSum`, in a thread-safe way.

`ThreadValue` fields and methods

- The `ThreadValue` class contains 3 fields: `Value`, `Threshold` and a reference to a `ThreadManager`.
- `GetThreadValue()` atomically retrieves the `Value` field.
- `AdjustThreadThreshold(_X)` atomically exchanges the `Threshold` field with `Value` + `_X`.
- `Decrement()` decrements `Value` and then waits for the `ThreadManager::ValueLock` to be released.
- `Increment()` increments `Value`. If `Value` exceeds `Threshold`, it calls `ThreadManager::UpdateManager()` and then waits for `ThreadManager::ValueLock` to be released.

X86/AMD64 Atomicity and Memory Ordering

- The atomic Read/Write aforementioned (`ThreadValue::GetThreadValue()` and `ThreadValue::AdjustThreadThreshold()`) can of course be done using `std::atomic<>`. ➔ Reliable but not fast…

- Using Microsoft™ Visual Studio® `volatile` specific behavior, we get the desired memory ordering: (refer to https://msdn.microsoft.com/en-us/library/12a04hfd.aspx)

> - A write to a volatile object (also known as volatile write) has Release semantics; that is, a reference to a global or static object that occurs before a write to a volatile object in the instruction sequence will occur before that volatile write in the compiled binary.
>
> - A read of a volatile object (also known as volatile read) has Acquire semantics; that is, a reference to a global or static object that occurs after a read of volatile memory in the instruction sequence will occur after that volatile read in the compiled binary.

X86/AMD64 Atomicity and Memory Ordering

- However, the Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A, « 8.1.1 Guaranteed Atomic Operations » states that:

> **8.1.1 Guaranteed Atomic Operations**
>
> The Intel486 processor (and newer processors since) guarantees that the following basic memory operations will always be carried out atomically:
>
> - Reading or writing a byte
> - Reading or writing a word aligned on a 16-bit boundary
> - Reading or writing a doubleword aligned on a 32-bit boundary
>
> The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:
>
> - Reading or writing a quadword aligned on a 64-bit boundary

➔Both atomicity and ordering constraints can be obtained by using `volatile` and by aligning our data, using `__declspec(align(#))`

`ThreadManager::GetGlobalValue()` algorithm

In a thread-safe way:

- Acquire `ThreadManager::ValueLock`, so no further calls to `ThreadValue::Increment()/Decrement()` can occur.
- Traverse the `ThreadManager::ThreadValues` vector and compute the sum of `ThreadValue::Value`.
- Release `ThreadManager::ValueLock`.
- Return the aforementioned sum + `ThreadManager::DTOR_ThreadValuesSum`.

This retrieves the sum of thread-local values (including deleted ones) at the time the function is called, due to the lock preventing subsequent changes.

`ThreadManager::UpdateManager()` motivation

- The `ThreadManager` may need to be periodically updated if the `ThreadValue`s (or their sum) have to be tracked.

- In this perspective, the `ThreadManager` class defines a user-defined `Callback()` function, used to set the « goal » global value for the next update, based on current global value.

  ➔ Threshold policy: Return a new, greater threshold if the global value exceeded the old one. Otherwise, return the latter.

  ➔Constant policy: Return the global value + or * a given factor.

- The `ThreadValue::Threshold` is computed so:

  ➔ The number of calls to `ThreadManager::UpdateManager()` is optimal for balanced workloads.

  ➔ The observed global value can be greater than the « goal » global value by a ratio of `MAX_ERROR` at most.

`ThreadManager::UpdateManager()` algorithm

- In a `std::call_once` way (ie. only one thread can call the function at a given time, other threads wait and return):
  - Traverse the `ThreadManager::ThreadValues` vector and compute the sum of `ThreadValue::GetThreadValue()`.
  - `GlobalValue` = `ThreadManager::DTOR_ThreadValuesSum` + the latter sum.
  - `Threshold = Callback(GlobalValue)`
  - `NewMargin` = `std::max(Threshold – GlobalValue, Threshold * MAX_ERROR) / #Threads`

    This is done to impose a minimal change, as the first expression converges to 0.
  - Traverse the vector a second time, calling `ThreadValue::AdjustThreadThreshold(NewMargin)` on each element.

ThreadManager performance

- 4 threads either having their own ThreadValue, or sharing a single variable (either `std::atomic<>` or a `long (long)`)
- 300,000,000 Increments per thread
- Benchmarks repeated 50 times
- Configuration: Microsoft® Windows™ 10 64 bits, Visual Studio™ 2015, Release Mode x86/x64, CPU Intel® Core™ i3 3110M (2.4 GHz, Ivy Bridge, HT enabled)

Time in seconds (s)	ThreadValue (std::atomic<>)	ThreadValue (X86/AMD64)	std::atomic<>	X86/AMD64
X86 (long operands, 32b)	**5.97**	**1.55**	34.8 (5.8x)	8.69 (5.6x)
AMD64 (long long operands, 64b)	**5.91**	**0.98**	29.6 (5.0x)	8.81 (9.0x)

4-6x greater performance

Due to very high contention

ThreadManager conclusion

- Good performance due to thread locality, which can be greatly improved on X86/AMD64 platforms.
- However, it is slower and less straightforward to **retrieve the sum** of the values stored in every thread, compared to a single `std::atomic<>` shared by all threads.
  ➔ `ThreadManager` is useful only if we do not retrieve this value often.