# Multithreading Part I

## # Review on Threads
A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. In a multiprocessor CPU, each thread runs on a different physical or logical core.

## # Why Threads?
In modern CPUs, there are 8 to 32 cores in a single chip. All of these cores are able to run simultaneously. Threads are the way to run different instructions on different cores. If you are not using all the cores available in the CPU, you miss out on the benefit of having multiple cores. The money end users dump on high end processors makes no significant difference if the program they are running is limited to a single thread, which is a waste of resource and performance the modern programs are expected to have.

## # Threads API
OS manages all of the hardware resources. In order to create a thread, OS specific API is to be used, which is different for Linux, Windows and Mac OS.

System call present in different OS for creating and managing threads are as follows:
Windows:
https://docs.microsoft.com/en-us/windows/win32/procthread/creating-threads
Linux:
https://man7.org/linux/man-pages/man7/pthreads.7.html
Mac:
https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html

Thankfully C11 wraps these and creates a platform independent thread library:
https://en.cppreference.com/w/c/thread

Unfortunately, since MS is bad at supporting C, there is no support for C11 threads in the MSVC compiler.
For this, we have whipped up a single header file that wraps the Windows threads with C11 API calls as follows:
https://github.com/IT-Club-Pulchowk/Advance-C-Workshop/Lecture-5/Samples/threading.h

## Important!!!
Before we get into the nitty gritty details of threads, following notes assume you to look up the code in github repo
https://github.com/IT-Club-Pulchowk/Advance-C-Workshop/Lecture-5/Samples/, run them and follow the instructions. In short, instead of being a passive reader, you are expected to follow along with the codes provided. Skimming the provided reference is a plus (and necessary in most cases).

# Creating Threads
Reference: https://en.cppreference.com/w/c/thread/thrd_create
Code: **threads_0_creating_threads.c**
(Compile this code in optimized build with **-O2** flag. In linux, you need to link against the pthread library i.e. add **-lpthread** flags when compiling.)

Q: Why is the **printf** in **thread_main** not being printed in an optimized build?
A: Because the main thread exits before the child thread could start executing the printf function. When the main thread finishes executing, the Operating System will pause the execution of all child threads (threads created by the main thread) and remove the entire process from the memory. Thus, the **printf** function in the thread we created didn't execute.

# Waiting Threads
Explanation: https://en.cppreference.com/w/c/thread/thrd_join
Code: **threads_1_waiting.c**

Try moving **thrd_join**  function before the **printf** in the main function and notice the change in output.

# Passing Arguments
Explanation: https://en.cppreference.com/w/c/thread/thrd_create
Code: **threads_2_args.c**

Here we pass a pointer to the simple struct **Thread_Arg** into **thrd_create** function. We receive that in the **thread_main** function which is the function we call for the child thread. The last argument in that function receives the pointer passed to the thrd_create function. We use the information in **Thread_Arg** struct to print a number of lines. In this way we can pass various arguments from the main thread to the child threads.

Why **void \***? You guys might have already learnt in the previous C workshop, but essentially any kind of pointer can be cast to **void \*** and you can cast **void \*** to any kind of pointer. We'll be learning about more usages of **void \*** to implement more cool and neat techniques to solve various kinds of problems in the last lecture. For now, we use **void \*** to pass arguments into the function whose parameters we don't have full knowledge about and we might need to pass different and many arguments. Most API developed in C which takes in a variable number of unknown arguments which you may have come across previously, if you have not then, you will come across this technique very often. So you are encouraged to be very familiar with the uses of **void \***.

The thing that you need to be careful about when passing arguments to the child thread is that arguments passed to the child thread must be valid while it is being used in the child thread. If the arguments passed to the child thread are not valid, then the program might crash (which is a good thing) but sometimes the program might not crash (which is a bad thing). The type of bugs

in which the program does not crash but is using a invalid memory address is very hard to debug.
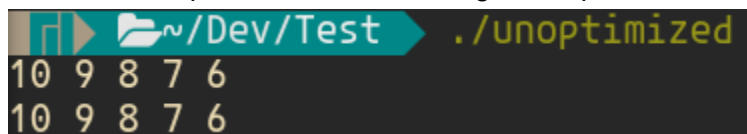

# Passing Invalid Arguments
Code: **threads_2_invalid_arg.c**
In this code, **numbers** is valid only when the scope is valid. But since it takes time to create a thread, before the main thread is created, the **numbers** become invalid and the child thread will use invalid memory (memory that is already freed) and cause memory corruption.

Try removing the quotes and run the program again.

**Note**: While running this program on windows(msvc compiler) with or without optimizations may not give any error, that's not always the case for Linux.
The code compiled on Linux in debug mode produces the following output.



The reason why this does not happen in windows is that when optimizations are not turned on, visual studio (msvc compiler) adds extra padding in the stack (to store various debug information) and when optimizations are turned on, everything gets ripped out and it directly prints the numbers!

# Adding upto 100000 multi-threaded
Code: **threads_3_mt_add_attempt0.c**
Run this code multiple times without optimizations turned on. Later we will go through the code with optimizations turned on.

Q: Why did the program fail?
A: Read write conflict / Data race / Race condition
A race condition or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when one or more of the possible behaviors is undesirable.
Assume that two threads each increment the value of a global integer variable by 1. Ideally, the following sequence of operations would take place:

| Thread 1 | Thread 2 | | Integer value |
| --- | --- | --- | --- |
| | | | 0 |
| read value | | ← | 0 |
| increase value | | | 0 |
| write back | | → | 1 |
| | read value | ← | 1 |
| | increase value | | 1 |
| | write back | → | 2 |

In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

| Thread 1 | Thread 2 | | Integer value |
| --- | --- | --- | --- |
| | | | 0 |
| read value | | ← | 0 |
| | read value | ← | 0 |
| increase value | | | 0 |
| | increase value | | 0 |
| write back | | → | 1 |
| | write back | → | 1 |

In this case, the final value is 1 instead of the expected result of 2. This occurs because here the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location.


Run this code once again but with optimizations turned on.
Q: Why did the program once pass now?
A: Creating threads takes time, when the first thread was created and **thread_one** started executing, **thread_two** was still not created. The **thread_one** finishes execution before **thread_two** starts executing it's code.
This is the reason why debugging multi-threaded code is not easy, it might run in either debug or optimized builds and fail in other builds. Many times replicating / reproducing the bugs to debug them is also very difficult.

Explanation:

A race condition can be difficult to reproduce and debug because the end result is nondeterministic and depends on the relative timing between interfering threads. Problems of this nature can therefore disappear when running in debug mode, adding extra logging, or attaching a debugger. A bug that disappears like this during debugging attempts is often referred to as a "Heisenbug". It is therefore better to avoid race conditions by careful software design.

So how does this happen in the CPU?
Each core in the CPU has a separate cache from which the core reads and writes the variable.
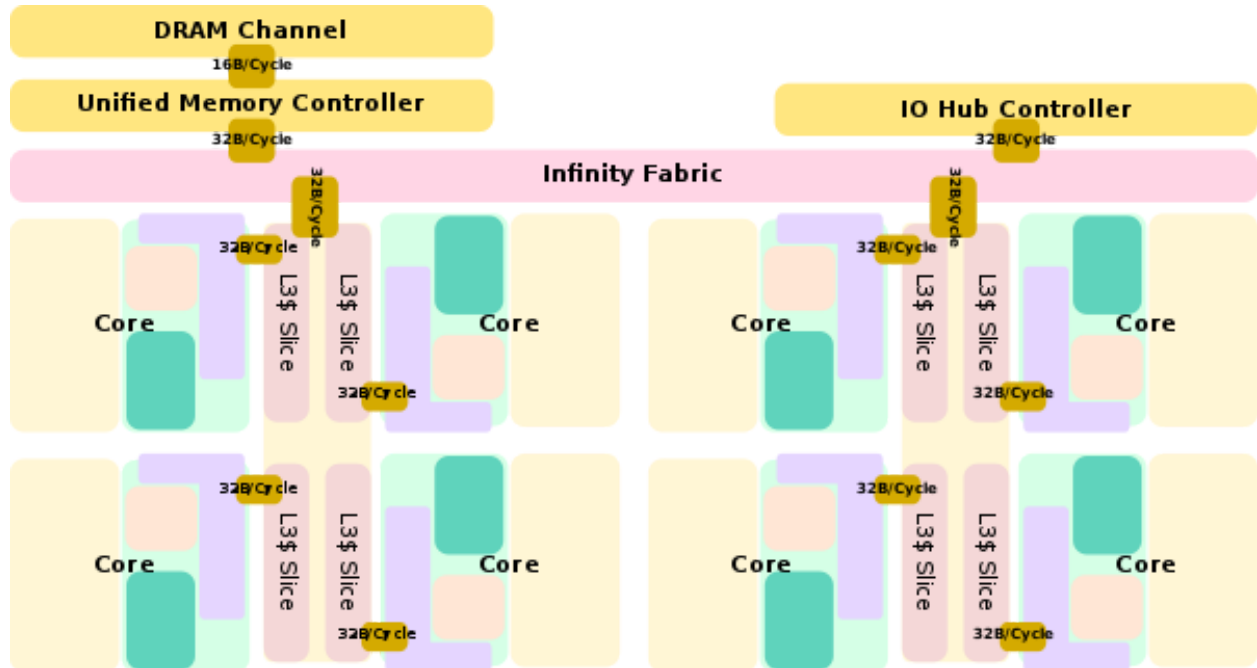


Here in the above image, we can see that each core has its own Cache (L2 - Green, L1 - Skin). Each core loads the memory required for its operations in the cache. After performing operations it does not write back to the memory immediately. This is because writing back to the main memory takes time and we might immediately update the same memory again. But at the same time another core could be using the same memory in it's cache, and it may update that value as well. In this case there's a data race, the core that writes back to the memory last replaces the update done by the previous cache. That might be what the programmer intended or it might not be.

The code section in which the processor will be accessing the shared memory (where there's a potential for bugs), is called the **critical section**.

Let us try another attempt
Code: **threads_3_mt_add_attempt1.c**
The number of errors to happen in the program has gone down but u need to fix all the threading bugs in your program. Gotta make the program run 100% accurately all the time. The problem is not visible not because we have fixed the problem entirely, but because we have reduced the probability of the bug. Currently there's no error because one thread starts late,

which means that by the time it gets to the shared variable, another thread will have already updated the shared variable, so there will be no error. Again the reason why it is hard to debug threaded programmes.

So what do we do to solve our problem?
Mutex: a lock or mutex (from mutual exclusion) is a synchronization primitive: a mechanism that enforces limits on access to a resource when there are many threads of execution.

Mutual exclusion in C11: https://en.cppreference.com/w/c/thread (See section: Mutual Exclusion)

Initializing mutex: https://en.cppreference.com/w/c/thread/mtx_init
Locking mutex: https://en.cppreference.com/w/c/thread/mtx_lock
Unlocking mutex: https://en.cppreference.com/w/c/thread/mtx_unlock

When working with mutex, you initialize the mutex. When you want to enter the critical section, you lock the mutex and after finishing the work in the critical section, when you want to leave the critical section, you unlock the critical section. When mutex is locked, and another thread tries to enter the critical section, it won't be able to further lock the mutex, since it is being locked by another thread. Then the thread has to wait until the thread that is in the critical section to unlock the mutex.

Code: **threads_3_mt_add_attempt2.c**
Here we have used the mutex to lock our shared variable, so the access of the variable between the threads is synchronized.
Q: While the program is correct, why is the program slower than before?
A: Locking and unlocking mutex is expensive. The write back buffers in the cpu need to be flushed. We'll come back to this later.

Q: What to do to solve this problem?
A: Same as before but lock the mutex only after the loop, until then use a local variable.

Q: But exactly how does mutex work?
Let's create our own mutex to understand more about the workings of the mutex.

Let us start with the basic mutex implementation. Implementing a basic mutex without using atomic operations.

Demo: **threads_4_impl_mutex.c**
(Again compile without optimizations)

Q: We created our own mutex, but why is the mutex not working??
Note that this might work in release builds, because before one thread is being created another thread will have finished it's work.

But let us check some assembly (No, you aren't expected to understand all asm):

```
1    thread_zero(void*):
2            add      DWORD PTR g_mt_result[rip], 1249975000
3            xor      eax, eax
4            ret
5    thread_one(void*):
6            sub      DWORD PTR g_mt_result[rip], 544892296
7            xor      eax, eax
8            ret
9    mutex_init(Mutex*):
10           mov      DWORD PTR [rdi], 0
11           ret
12   mutex_lock(Mutex*):
13           mov      DWORD PTR [rdi], 1
14           ret
15   mutex_unlock(Mutex*):
16           mov      DWORD PTR [rdi], 0
17           ret
```

As we can see here, the spin lock for the mutex is non-existent. The compiler just puts 0 into the g_mutex->value. It doesn't even check if it was 1 before!!



Q: What happened???
A: The C compiler does not understand that it is using that variable in multiple threads for synchronization. If the program was to be a single threaded program, the value of the mutex would not have side effects on the execution of the program.

Q: So what do we do now?
A: We just gotta tell the compiler that the value of the mutex can change anytime in any thread and the compiler should not touch and modify it!! We use the keyword volatile in C to tell the compiler that the value of the mutex can change anytime and will have side effects in the program.

Code: **threads_4_impl_mutex_volatile.c**
Here's what the assembly looks like…

```
1    thread_zero(void*):
2            mov     ecx, DWORD PTR g_mt_result[rip]
3            mov     edx, 50000
4    .L2:
5            mov     eax, DWORD PTR g_mutex[rip]
6            cmp     eax, 1
7            je      .L2
8            mov     DWORD PTR g_mutex[rip], 1
9            mov     DWORD PTR g_mutex[rip], 0
10           sub     edx, 1
11           jne     .L2
12           mov     DWORD PTR g_mutex[rip], 0
13           lea     eax, [rcx+1249975000]
14           mov     DWORD PTR g_mt_result[rip], eax
15           xor     eax, eax
16           ret
```

Now we see our mutex being used correctly, yay!!

Note: We are using godbolt to view the assembly code, here's the full link:
https://godbolt.org/z/TqP1hrhnn

Let's run the code again.
Q: The results are still not correct, why?
A: The reason is the same as before, the shared value in the mutex is not thread-safe. One thread can access and update in it's local cache and the other thread can do the same.

Q: So how do you make the mutex thread safe?
A: Atomic operations

Q: What are atomic operations?
A: Atomic operations are the operations that the processor performs atomically, which means that the processor makes sures that operation happens in only a single CPU at a time. Some of the atomic operations available in modern processors are as follows:
   - Bit operations: BTS, BTR, BTC
   - Exchange operations: XADD, CMPXCHG
   - XCHG

- INC, DEC, NOT, NEG
- ADD, ADC, SUB, SBB, AND, OR and XOR

Note: All these assembly instructions need to have LOCK prefix in them for atomicity except XCHG. XCHG is always atomic.

But we won't be using assembly, LOL. What we'll be using is called intrinsics. "Intrinsics" are those features of a language that a compiler recognizes and implements without any need for the program to declare them.

"Intrinsics" are different in different compilers, and figuring that out is homework. We have created a wrapper function for these intrinsics for simplicity, which you can use as well.
Link: https://github.com/IT-Club-Pulchowk/Advance-C-Workshop/Lecture-5/Samples/threading.h
Let us first understand how interlocked increment, interlocked add and interlocked exchange works:

Interlocked increment is pretty simple, it takes in a pointer to a value and increments it atomically. Same goes for interlocked add, it adds some value to the destination atomically. Both these intrinsics return the result of the operation.

As for interlocked exchange

```C++
LONG InterlockedCompareExchange(
  [in, out] LONG volatile *Destination,
  [in]      LONG          ExChange,
  [in]      LONG          Comperand
);
```

It takes in the *Destination* which will be compared against the *Comperand*. If value in *Destination* is equal to *Comperand* the value in *Destination* gets replaced by *ExChange*. It returns the original value that *Destination* had. All this happens atomically. This is the fundamental operation for a very large number of multi threaded codes and data structures.

Now let us use this atomic operation to implement our mutex, finally!
Code: **threads_5_custom_mutex.c**

Finally our mutex works correctly!!
And whether you realized it or not, the code that uses our mutex runs much faster than the one with the OS provided mutex.

Q: Why did our mutex run much faster than the OS provided mutex and is there some trade off.

A: The reason for our mutex code running faster is that we don't sleep for some time if we fail to lock the mutex but the OS mutex does. In windows, the default minimum sleep period is ~10 millisecs, in linux it's ~1 millisec. To change the default minimum timer resolution in windows, you need to call the following function:
https://docs.microsoft.com/en-us/windows/win32/api/timeapi/nf-timeapi-timebeginperiod
This is the reason why locking mutex is expensive. If you don't want to sleep if the lock fails, then you need to create a separate mutex and spin lock the CPU. The downside to this approach is that the program will use much more CPU without doing actual work. It spins the lock checking if the lock has been unlocked or not and nothing more.

BUT wait, different CPUs have different cache, and the values are read from different cache and written to different cache. Although atomic operations ensure that the operation takes place in only one CPU at a time, couldn't the CPU be using outdated values from the cache?

Here comes Memory Barriers.
But first a little background.

Out-Of-Order Execution:
There are 2 things you need to know. The order in which your code gets executed. If two or more statements produce the same result despite the order of their execution, the compiler may determine the order in which they get executed.

Let us take a simple example

```
int foo() { return 42; }
int bar() { return 69; }

int foo_bar(int a, int b) { return a * b; }

int main() {
    int x, y;
    int result;

    x = foo();
    y = bar();

    result = foo_bar(x, y);

    return 0;
}
```

In this code, the compiler does not guarantee that if foo() gets executed first or bar().
But it guarantees that foo_bar() gets executed after the execution of foo() and bar() because x and y are dependent on foo() and bar().

So in order to solve this problem, we put memory barriers which are again compiler dependent, sigh C.

```c
int foo() { return 42; }
int bar() { return 69; }

int foo_bar(int a, int b) { return a * b; }

#define SOFTWARE_BARRIER /* Compiler specific */

int main() {
    int x, y;
    int result;

    x = foo();
    SOFTWARE_BARRIER;
    y = bar();

    result = foo_bar(x, y);

    return 0;
}
```

But as we have learned in the previous lecture, x64 is an out of order processor, which means that the execution of independent instructions may change the order of execution that is in the file. We need another memory barrier, telling the processor not to rearrange my instructions.

```c
int foo() { return 42; }
int bar() { return 69; }

int foo_bar(int a, int b) { return a * b; }

#define SOFTWARE_BARRIER /* Compiler specific */
#define HARDWARE_BARRIER /* Compiler specific or Some asm code */

int main() {
    int x, y;
    int result;

    x = foo();
    SOFTWARE_BARRIER;
    HARDWARE_BARRIER;
    y = bar();

    result = foo_bar(x, y);

    return 0;
}
```

Memory barriers ensure the order of execution of the instructions, but they are expensive as the compiler and the processor change the order of execution to optimize and run the code as fast as possible.

Q: So, what do Memory barriers actually do?
We won't go into much detail about working of Memory barriers, but basically what it does is flushes the write back cache, which basically results in all the CPU caches with shared values getting synced. To go into a little more detail, the results from instructions such as addition, subtraction, or any other instructions done in the ALU is not immediately written back to the registers / caches, since they might get operated on again, so what the processor does is stores them in the local cache call the write back cache, and uses them directly from the write-back cache. This is done to increase the speed of operation on the same memory. But what memory barriers do is flush this write back buffer, i.e. send all the values in the write back buffer to the cache and registers. These values may need to be updated in other CPU cache as well. Memory barriers also ensure that the latest values are loaded in the cache by communicating with neighbouring CPU caches. The communication protocol is called MESI or extended MESI protocol, which we'll not go into detail here. This is a pointer to people who are interested to look that up themselves.

Okeh, wait what? So you guys remember the problem we had with atomic operations not being synced with other CPU caches? So what we can do is use memory barriers after executing each atomic operation.

But we have been lying a little . You don't have to do all that (at least right now).

The processor inserts memory barriers after every atomic operation. So you don't have to worry about that.

Q: Keen observers should have seen something about our code till now, that we don't need mutex for this example code we have been running till now.

A: We could have made the global variable volatile and did atomic increment in the variable itself!

Let's see that: **threads_6_no_mutex.c**

Q: What if you want to let 2 or a certain number of threads access the critical section?
A: Semaphores to the rescue

Semaphores are basically the same as mutex, but instead of being binary, that is either it's locked or unlocked, semaphores keep the count for the number of times it is locked.

The implementation of Semaphore will be provided as sample code in the Workshop repository. Just like mutex, there will be semaphore implementation from OS API as well, which similar to mutex sleeps if the semaphore count is at the max value. In our case we simply loop indefinitely.

A question to everyone before we move to the last section: There's one console in a process, and we print values from different threads, but they didn't collide in our first examples, why?
A: The reason is printf uses mutex, and only lets one thread actually print to the console.

**# Deadlock**
Let's start with a simple scenario :
Suppose you brought a toy (that comes in two parts like a drum and a stick) to two childrens. They ran toward you to grab that toy (drum-stick) and one child got a drum and another got a stick.Neither children could play with that toy now. Say both children are stubborn (like most children are), one demands the other to give the other part of the toy and they are basically stuck. Since both demand parts which are in possession of the other, neither can move forward to play with that toy.

The similar thing happens when two (or multiple)  threads of execution, each hold one lock and tries to get ownership of the lock held by another thread which in turn is waiting for this thread to release its lock. Mutex is just an example, it can happen over ownership of any shared resources. In this case neither thread can progress since one thread is waiting for the other to release their lock (those stubborn threads :D). This is basically a **deadlock**.

Q. How to resolve deadlock?
A: Deadlock is often difficult to detect and there may not be a simpler way  to resolve it even if it is  detected. Some guidelines that can be followed to avoid deadlocks are :

a. If two mutexes are locked in the same order (no restriction on unlocking), deadlock never occurs. If thread A holds lock1 first, then lock2 and thread B tries to get ownership of lock1 it must block until thread A releases its lock, which avoids deadlock.
   But if thread B tries to lock2 first, there's a potential deadlock.

b. Lock two or more locks with so called deadlock avoidance algorithm :
   Let M1, M2, M3 be three locks. If we try to lock M1,M2 and M3 in any sequential order then there's a risk of deadlock (if another thread tries to lock them in different order).
   So, what we can do is, use a procedure **deadlock_avoid_lock** with mutexes as parameters,
           deadlock_avoid_lock(M1,M2,M3);
   Say if the current thread acquires locks M1 and M2 and can't get ownership of M3 (for a certain time),then there may be a potential deadlock lurking around. In this case we release the lock M1 and M2 and try to get ownership of lock M3 first. It avoids any deadlock that could have been caused if another thread holding M3 was waiting for ownership of M1 and M2. After getting ownership of lock M3, then it tries to regain ownership of M1 and M2 and so on. This continues until it locks all the mutexes.

c. Don't use mutual exclusive ownership of resources (which may not be feasible everytime) i.e if there's no lock, then no deadlock

d. Use lock-free programming -> Guaranteed to never deadlock (that is if you get it right :D)

e. Ignore deadlock

## Deadlock handling   [ edit ]

Most current operating systems cannot prevent deadlocks.[9] When a deadlock occurs, different operating systems respond to them in different non-standard manners. Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one.[10] Major approaches are as follows.

### Ignoring deadlock   [ edit ]

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm.[10][11] This approach was initially used by MINIX and UNIX.[7] This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

Ignoring deadlocks can be safely done if deadlocks are formally proven to never occur. An example is the RTIC framework.[12]

Reference links:
- https://www.youtube.com/watch?v=p4sDgQ-jao4
- https://www.youtube.com/watch?v=emYmQQZXRT8
- https://en.wikipedia.org/wiki/Race_condition#Example
- https://www.youtube.com/watch?v=nh9Af9z7cgE
- https://www.youtube.com/watch?v=ZQFzMfHIxng&t=631s
- https://preshing.com/20120515/memory-reordering-caught-in-the-act/
- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

- https://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Atomic-Builtins.html#Atomic-Builtins
- https://docs.microsoft.com/en-us/cpp/intrinsics/x64-amd64-intrinsics-list?view=msvc-170
- https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- https://developer.amd.com/resources/developer-guides-manuals/