Chapter 1: Introduction to Performance and Concurrency

- Performance should not be an afterthought, it needs to be part of the initial design.

Chapter 2: Performance Measurements

- Perf Linux
- GPerg Tools
- Google Benchmark

Performance Benchmarking

We have a couple of tools in our toolkit:

1. C++ chrono timers
    a. Chrono measures real time, but often we want to measure CPU time, which is the time that is passing only when the CPU is working (not when its idle).
        i. In a single threaded program, CPU time cant be greater than real time.
2. High-resolution timers
    a. To measure the CPU time, we have to use OS-specific system calls, on Linux and other systems, we can use the clock_gettime call to access the hardware high-resolution timers.
        i. Returns seconds since some point in the past.
            1. You can call it with CLOCK_REAL_TIME, CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID.
                a. REAL_TIME is the amount of real world time that passes
                b. CPU time is the total amount of time used by all threads combined (so in a multithreaded environment, it can be greater than real time)
                c. THREAD TIME refers to how long the calling thread is running. For example, if we launch another thread via async and wait on the future, we are idle, while idle no time is accruing.

Better Way

1. Most modern profilers use one of three approaches:
    a. **Hardware event counters** that are present in all modern CPUs to track certain hardware events, such as an instruction.
    b. Other profilers require that the code is **instrumented** with special instructions during ompilation or linking. This provides additional information to the profiler. This provides additional information to the profiler (e.g. when a loop begins or ends).
    c. Other profilers rely on **time-based sampling**: they interrupt the program at a certain interval (once per 10 milliseconds), record the values of the performance counters, as well as the current location in the program. For instance, if 90% of calls happen during a call to compare() function, we can assume that the program spends 90% of its time doing sting comparisons.

2. Linux perf uses hardware performance counters and time-based sampling.
    a. You can use *perf list*command to show what hardware events exist. A list of events include:
        i. Cycles
        ii. Instructions
        iii. Branches
        iv. Branch misses
        v. Cache references (how many times we had to go to memory to fetch information)
        vi. Cache misses
    b. You can use the *perf record* command, specifying samples taken per second. It writes to a perf.data file, which can be visualized via perf report as well.

// SKIPPED GOOGLE BENCHMARK

3. Usually in profiling there is an initial burn-in period as the program starts fetching information from cache on load (populating its instrument and data cache). Benchmark library writers aren't ignorant to this, and often do a burn-in on every test (ignore the initial measurements of a test until the results 'settle'

Chapter 3: CPU Architecture

- Skip in favor of other documentation.

Chapter 5: Threads, Memory and Concurrency

- A **thread** is a sequence of instructions that can be executed independently of others threads. All threads share the same memory, so by definition, threads of the same process run on the same machine.

Thread vs Process

Process

Each *process* provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

Thread

A *thread* is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's

process. Threads can also have their own security context, which can be used for impersonating clients.

- **Hyper threading** or **symmetric multithreading** is when a single processor appears to the operating system and the program as two or more separate processors, each capable of running independently. In reality, all threads running on one CPU compuete for the shares internal resources, such as registers. SMT can provide performance gains if each thread does not make full use of these shared resources, or is in a blocked / waiting state.
    - Sometimes the amount of work that can be completed by threads depends on hardware outside their control, like memory bus throughput. If all threads are writing, the rate of data transfer may hit a ceiling.
- C++ gives us the ability to declare atomic variables. These variables can perform non-interuptable transactions (any other thread observing this variable will see its state either before or after the atomic operation, but never during).
    - Beyond changing an atomic variable, there are side effects that involve the *visibility* of changes made before and after the change in that atomic variable as observed by two threads that share this variable.
        - The semantics used describe how side effects (writes and reads made prior to altering the atomic variable) of altering the atomic variable become visible to both threads
            - In general we want to use the memory order that is restrictive enough for the correctness of the program but no more restrictive than that.
    - One can think of relaxed ordering as only guaranteeing atomicity and modification order consistency. It does not impose order among concurrent memory accesses.
    - One can think of **release acquire** semantics creating a barrier with respect to the operations that happen before altering the atomic variable by the writer thread. This barrier is semi-permeable, meaning that operations executed before the barrier cannot cross it; but, operations that happen after it can be visible to the reading thread.
        - On the other hand, the thread that reads also needs to specify a barrier. Why? Because the guarantee that everything the producer has done up to the atomic write is not enough. We need to maintain the guarantee that the operations executed by the consumer to process this new object cannot be moved backward in time to a moment before the barrier when they could have seen the object in an unfinished state.
    - One can think of **sequential consistency** as the enforcement of a single global order amongst processors. If you think of a program as a deck of cards, where each processor has part of the deck, when that deck is combined and shuffled, a single processor's cards appear in the same order throughout the deck as they appeared in the process, despite being interleaved with other cards from other processors.
- The best performance you can have for accessing shared data for write purposes is singe-threaded performance.

- **False sharing** occurs when a cpu believes that it has exclusive access to a cache line, but that cache line is shared with another cpu, hence, the cpu acts serially, as though data is shared with the other line-locking thread. It needs to wait until the cache line is relinquished, invalidating its own cache line, and re-retrieve that line from LX cache.
    - For example, an atomic increment reads a single word at the specified address, increments it, and writes it back. Where does it read from? The CPU has direct access only to the L1 cache, so it gets the data from there. If its not there, the data needs to get from main memory into the cache. Its copied over the memory bus, which is much wider, and ships data in larger chunks known as cache lines
        - Cache lines are always 64 bytes on x86 CPUs.
    - When a CPU needs to lock a memory location for an atomic transaction, even though it may be a single word (8-bytes) it has to lock the entire cache line.
        - If two CPUs are allowed to write to the same cahce line into memory at the same time, they'll overwrite each other, which is messy.
        - This is why, if even one bit of a cache line is changed, the entire line is invalidated in the cache of all other CPUs that hold the same line.
    - Even though adjacent array elements are not shared between threads, they occupy the same cache line and act as though they are shared (each thread can only access their atomic variable one at a time).

Chapter 6: Concurrency and Performance

- **Compare-exchange** takes two parameters: the first one is the expected current value of the atomic varible, and the second one is the desired new value. If the actual current value does not match the expected one, nothing happens. However, if the current value does match the expected one, the desired value is written into the atomic variable.
    - The return value is true to indicate if the write happened. If the result is false, the actual value is returned in the first parameter.
    - There are two more parameters. They are memory order parameters. The first one applies when the compare succeeds and the write happens.
- There are three types of concurrent programs:
    - Lock-based – One thread is holding the lock that gives it access to shared data. Just because its holding the lock does not mean that its doing anything with the data, or that its even making any progress.
    - Lock-free – There are no locks, and multiple threads may be trying to update the same shared value. Only oner of them will succeed. The rest have to discard the work done based on the original value, read the updated value, and do the computation again. But, what we do know is that at least one thread is always guaranteed to commit its work and not redo it. Thus the entire program is always making progress.
    - Wait-free – Often associated with atomic variables where, from the programmers point of view, there is no waiting for anything. An atomic operation requires exclusive access to a cache line, but beyond that there is no blocking, or retrying. Each thread is executing the operations it needs and is always making progress.

Spinlock

1. The idea behind a spinlock is simple: the lock itself is just a flag that can have one of two values: 0 and 1.
    a. 0 means that the lock is not locked
    b. 1 means that the lock is currently held. Therefore any thread that sees 1 needs to wait until the value changes back to 0.

A side-effect seen here is that, since the CPU will think that the thread hammering the flag is doing useful work, it will get more processing time while, the thread that wants to release the lock, may not get scheduled for some time. Hence the solution is for the waiting thread to give up the CPU after a few attempts, so some other thread can run and, hopefully finish its worth and release the lock.

Problems with Locks

1. Deadlock -
2. Livelock – Two threads pass each others locks to each other like hot potato, and no thread can ever make any progress when it tries to acquire the lock it just passed to the other thread. Kind of like starting a phone call with your grandpa and you're both going back and forth yelling 'hello, can you hear me', 'yes I can hear you can you hear me?', 'yes I can hear you can you hear me?' to each other.
3. Convoying – Since threads don't have the notion of priority with respect to locks, its possible that, after a thread that has acquired a lock releases it, while the other threads are waking up, the thread that just released it acquires it again for another operation, perpetually starving the other threads of being able to do any work.