GThread: A Safe C++ Multithreading Runtime System

Garrett Wang

I. Project Overview

Due to the physical constraints on a single processor such as heat dissipation and energy consumption, hardware manufacturers have been struggling to improve their processors' clock speeds for the past decade. At the same time, those constraints have also led to a replacement of single-core processor by multi-core processor. While hardwares have improved their performances by installing more cores into machines, unmodified serial programs will not benefit from any of those upgrades since they will still just be run on a single core. That explains why for the past decade, multi-threaded programs have taken over the place of serial programs to better exploit the performance benefits provided by the underlying hardware.

However, writing multi-threaded programs in C/C++ correctly and having them run efficiently is by no means an easy task. Due to the nature of sharing address space among threads, an entirely new class of errors and issues such as deadlock, race conditions, atomicity violations and order violations are introduced the moment developers try to create a second thread in their programs. My final project for this course implements a system named G-Thread: a safe C++ multithreading runtime system, that eliminates concurrency errors for multithreaded C++ programs that are based on fork-join parallelism.

The key insight of G-Thread is turning threads into processes and imposes a sequential semantics to create the appearance of deterministic sequential execution for programmers to safely run and reason about their programs while still being able to take advantage of the number of cores to run code concurrently. By running threads in different processes deterministically and atomically eliminates the issues of atomicity violations and race conditions. Due to the sequential semantics kept by G-Thread, threads are run in program order so the order is guaranteed as well. Since the users of G-Thread never need to use locks, there is no such thing as deadlock in this system.

The overhead of G-Thread in worst case which leads to rollback in every single thread is around 20 times to 40 times comparing to the corresponding pthread programs. The overhead of G-Thread running without touching the heap entirely is around 8%.

Most of the implementation details and system ideas of G-Thread originated from a system named Grace published in 2009 by Professor Tongping Liu and Professor Emery Berger. As a matter of fact, G-Thread actually implements a subset of Grace excluding the support for global variables and transactional I/O.

II. Design & Implementation

A. High-Level Characteristics

Sequential Semantics & Fork-Join Parallelism

G-Thread enforces deterministic execution of programs that rely on fork-join parallelism. This model is commonly used and simple: each thread spawns off some child threads and waits for them to complete. However, the downside of supporting only fork-join parallelism means that concurrency control through synchronization primitives like condition variables, semaphores, etc. is not supported. The reason that G-Thread imposes this limitation is that every fork-join based paralleled program has a serial counterpart which is running those threads in program order, and we refer to it as the sequential semantics kept by G-Thread. Instead of pthread's creating threads without knowing when those threads will actually get run and if they will ever be switched around during the middle of the execution , G-Thread guarantees that, as far as programmers concerned, the threads are run in the order they are created and thread always run atomically. It may seem like by doing so, the entire program has been serialized and loses the most important reason to write multi-threaded programs in the first place. However, G-Thread actually makes sure that during the transaction, the code will run concurrently to take advantage of the number of cores available. The only serial part in the whole system is when the threads try to commit their local updates to the global state since G-Thread needs to make sure that only one thread can commit at a time.

Optimistic Concurrency Control

To achieve concurrent speedup of the program, G-Thread adopts the idea of executing threads speculatively, and then committing them deterministically, similar to how software transactional memory (STM) works. G-Thread tries to commit the transaction and starts a new one whenever the current thread spawns off a new thread or joins a thread. If there is a conflict between the local state and the global state when it tries to commit the transaction, G-Thread aborts all local updates to the local heap that thread has been using for the transaction and rollbacks to the beginning of the current transaction.

B. Major Components

Processes as Threads

One of the key insights of having efficient optimistic concurrency control is to use processes instead of threads. Rather than spawning off threads, G-Thread actually forks off process to run each “thread”. One of the most obvious and the largest advantages of using process instead of thread is different processes have different address spaces. This property ensures that no memory conflicts will ever happen during the execution of the thread. But since processes don’t share address spaces, they won’t be able to see the memory updates done by other processes. G-Thread solves this issue by reserving two memory mapping for each thread: one shared global mapping that reflects the latest committed state for the entire program, one copy-on-write private mapping that each thread uses directly. When a thread requests memory from the OS, a pointer on the private mapping will be returned to the user so that any changes done through this pointer will not be reflected in the global state immediately. When a thread finishes execution and is able to commit its local change to the global state, updates done on the private mapping will be copied over the global mapping. In addition, before the beginning of the each transaction, the local mapping is guaranteed the sync up with the global state.

Heap

G-Thread uses its custom allocator to take full control of the memory used by the users. The access to the memory is tracked at a page granularity, sacrificing the precision for the performance. It keeps a read set and a write set keep track of the read and write access to the page respectively. Similar to one local and one global view of memory kept by each thread, each thread also has a local and global map of page version numbers which reflects the version of the page locally and globally.

Global & Transactional I/O

The original system Grace has also implemented similar tracking scheme for global variables to ensure global variables are also free of concurrency issues as mentioned above. It also implemented transactional I/O to make sure that revocable I/O will be buffered and irrevocable I/O will wait until only the appropriate time to perform. However, due to the time constraints of this project, G-Thread doesn’t implement those two features. As a result, global variables and I/O related operations will suffer from concurrency issues.

C. Thread Execution

Initialization

G-Thread initializes a transaction at the beginning of the entire program and at the start of every thread. It first saves the context of the execution, which includes program counter, registers and stack contents. Then it sets the protection of every page to `PROT_NONE` so that any access, both reads and writes, will trigger a segfault notifying G-Thread to record the access.

Execution

During the execution, G-Thread tracks accesses to pages by handling the segfaults mentioned above. The first access to each page is always considered as a read. Then G-Thread will change the permission of that page to `PROT_READ`. If that access is really a write, once the control returns from the signal handler, it will immediately trigger the segfault again. For the second segfault to the page at any time, it will be treated as a write and the permission of that page will be restored to `PROT_READ | PROT_WRITE`. Any subsequent access to the page will proceed at full speed without any kind of tracking or logging by G-Thread.

Completion

Once a thread finishes its thread function, it will try to commit its local updates to the global state. G-Thread first checks to see whether the read set is empty. If that is the case, G-Thread is able to commit that thread immediately since there is no updates made to the heap. While this can be rare, it is an important optimization since otherwise, the thread needs to wait its logical predecessor to finish before it can commit.

Committing

If a thread won't be able to commit through the case mentioned above, it needs to wait until its logical predecessor to finish. Then it does a consistency check to make sure that the page version of each page in the read set matches the page version of those pages in the global page version map. If any of them does not match, that means there is a memory conflict happened during thread's execution and we need to rollback to the beginning of the transaction where we save the execution context. If they all match, then G-Thread can safely copy over the contents of each page in the write set to the global state and finishes the transaction.

III. Evaluation

All evaluations are done on a 4-core Linux system with 8GB of RAM. Each core is running at a fixed speed of 3.3GHz. For correctness purpose, test programs are always run at least 50 times to make sure the concurrency issues are indeed avoided and the runtime of the test programs are measured through `time` command.

I try to evaluate G-Thread by asking the following three questions as outlined in the proposal:

1. **Is G-Thread really able to be free of common concurrency bugs (deadlock, race condition, order violation, atomicity violation)?**

   Deadlock

   Since the user of G-Thread will never need to acquire a lock, there is no such thing as deadlock. As programmers transfer their code from using pthread to G-Thread, they do need to remove all lock related code to avoid another layer of overhead imposed by locks.

   Race Condition

   Race conditions often happen when multiple threads are trying to update at the same memory region. One of the test program G-Thread runs against creating 200 threads and waiting for them to join. All those threads run the same thread function which just updates one integer allocated on the heap. Under the normal pthread program, this will lead to race condition where one of the updates to that local variable may get lost. Since G-Thread runs each thread atomically, there none of those updates will ever get lost . Also as the test shown, G-Thread always retains the expected count in the end without exception across the 50-time run.

   Atomicity Violation

   Atomicity violation usually happens where developers think a piece of code will run atomically but it doesn't. The same test program mentioned in the section above suffices to test the existence of atomicity violation. Same as race condition, since G-Thread runs each thread atomically, the entire thread function proceeds as an atomic section as a whole. The test program again verifies that G-Thread is free of atomicity violation as well.

Order Violation

Order violation happens due to the indeterministic execution of pthreads. By creating two threads with pthread, developers have no idea which thread will run first and it is not safe to make assumption that either thread will run before the other. With G-Thread's sequential semantics, threads are executed in program order so that the second thread created by G-Thread is guaranteed to be committed after the first thread commits. I wrote a test program that only creates two threads with two heap-allocated variables x and y both equal to 1. One thread checks to see that if x equals to 1, then y will be changed to 0. The other thread does the opposite: if y equals to 1, then x will be changed to 1. Normal pthread doesn't guarantee that the final result of this program will be x equals to 1, y equals to 0 or y equals to 1, x equals to 0. It is also possible, due to atomicity violation, that both x and y will be 0. Since G-Thread makes sure that one thread will always finish before the other, the result is deterministic and there both x and y equal to 0 will never happen. This property also helps developers to reason about their programs as they have the full control of the control flow of their programs.

Although heap memory is free of those four categories of concurrency issues, G-Thread is not safe on global variables and transactional I/O as mentioned in the previous section. Also due to the time constraints, the custom memory allocator implemented in G-Thread does not free any memory requested by the user. It will only be an issue for programs that need to use large amount of memory.

2. **How easy or user-friendly is it to use G-Thread?**

   It is fairly simple and easy to use G-Thread. All you need to do to transform your original pthread program to use G-Thread is to include the `libgthread.hh` header file and recompile your program to link against G-Thread library. If you do have locks in your original pthread program, you need to remove them and rest assured that your program will still work without concurrency issues.

3. **How much overhead does G-Thread impose?**

   The overhead of G-Thread is difficult to calculate due to the complexness of this system and many factors need to be concerned. I did two extreme tests to test how G-Thread behaves in the worst and best case scenarios.

   Worse Case Scenario

The test program for this scenario creates 200 threads and waits for them to finish. All of those threads run the same thread function which increments an array of values across multiple pages. Since all those threads access the same set of pages, inevitably during the commit stage, only one of them will be committed successfully and rest of them will be rolled back. Since transactions are rolled back all the time, this is the worse use case of G-Thread. By comparing the runtime of this test program with G-Thread and the runtime of using pthread with locks, G-Thread slows down the program around 20 to 40 times depending on the number of pages touched in each thread function.

Base Case Scenario

In this case, I test G-Thread with a password cracker program. The password cracker program iterates through all possibilities of lowercase alphabet only 6-character passwords and generate their MD5 hashes to match with the hashes provided in the input file. The original program runs on four threads and never touch the heap. Although it does have a global arraylist to check whether all passwords have been cracked, there is no concurrent writes to the same element at any given time. Therefore, even G-Thread isn't safe on global variables, it is in this case. Since this program never touch the heap memory by any means, it is purely a measurement of the overhead for starting up the system and setting up the tracking. The overhead of G-Thread in this scenario is around 8 percent, which is not really ideal and can definitely be improved.

IV. Conclusion and Future Work

In this project, I implemented the system named G-Thread, which provides safe and deterministic execution for threads and keeps multi-threaded programs from common concurrency bugs such as deadlock, race condition, atomicity violation and order violation. Future work can be done for this project involves finishing the global variable and transactional I/O sections of the original system. I should also properly implement the deallocation of memory for the custom memory allocator. In addition, there are definitely many places I can fine tune the system to lower the overhead. For example, G-Thread should be able to know when a transaction needs to be rolled back at the moment that conflicted access has happened and it should rollback that thread immediately rather than waiting for it to finish.