Comparing Synchronization Primitives for Varying Synchronization Workloads

Donald Elmore & Ryan McDonnell

***Abstract*—There are a few commonly implemented synchronization primitives that may leave a programmer questioning which to choose. We first took a glance under the hood of three common primitives to analyze low level runtime. We then showed that accepted practices suggest a certain fastest scenario for each primitive and made a prediction accordingly. We developed three testing scenarios to compare these implementations and analyzed the results to confirm or deny expected results as well as examined potential other trade-offs.**

***Index Terms*—mutex lock, condition variable, semaphore, synchronization, contention, locking, thread, pthread, runtime.**

I. Question

AMONG the academic community there is an accepted practice that certain synchronization primitives each have their fastest use case, specifically with mutex locks, condition variables, and semaphores. Mutex locks are used when only mutual exclusion is desired with a minimal number of threads. Condition variables are used when more than just mutual exclusion is needed, but also a number of threads need to wait for a certain condition to be met. Semaphores are a way for a producer to limit the number of consumers for some set of resources. The **question** we will be asking is: are there meaningful differences between synchronization times when choosing a particular synchronization primitive given a particular workload?

II. Model

Synchronization is a necessity when wanting to protect a critical section in code. A critical section is simply a section of code that can only be executed by one thread at a time for the piece of code to have the desired results [1]. Synchronization of threads can be achieved in a multitude of ways depending on the intended implementation. Three of the most common methods (mutex, condition variables, semaphores) will be analyzed throughout this experiment. Of interest is comparing the runtime of these primitives with all being used for the same implementation. Below is a ”under the hood” look at each of the three primitives of interest.

A. Mutex Locks

A mutex is made up of two major parts: a flag indicating whether the mutex is locked or not and a wait queue of the waiting threads. The assembly instructions of a mutex are not the main overhead involved in implementation, that lies with the memory coherency. Change of the flag is a minimal number of instructions and can be completed without a system call. If the mutex is locked then a system call will occur to add the thread that called the lock to the mutex's wait queue. If the wait queue is empty then unlocking is cheap because a new thread does not need the lock. If the wait queue is not empty, then a system call is needed to wake up one of the processes in queue. To summarize, locking an existing unlocked mutex is cheap and unlocking a mutex with no wait queue is cheap. The issue with mutex locks is that a programmer would potentially need to have threads continually polling to check if a condition is met. This could be resource intensive because the thread would be continuously busy.

B. Condition Variables

Condition variables enable threads to wait until a condition occurs. Threads are able to atomically release a held lock and enter the sleeping state afterwards. While they can be used with reader/writer locks, this experiment focuses on their critical section use. Condition variables allow for waking one or waking all waiting threads. Once a thread is awakened it can then reacquire the lock it released when the thread initially went to sleep. A condition variable is used with a mutex, and a given condition variable can only have one mutex associated with it. The mutex that protects a condition must be locked before waiting for the condition. The subroutine atomically unlocks the mutex and will block the caller until the condition is signaled, then when the call returns the mutex will be relocked. Pthread_cond_wait will block a thread, if the condition is never signaled then it will never wake. The low-level performance benefit of a condition variable is that the consumer is able to find when the queue is empty, with no polling necessary. The thread isn't doing anything until a producer adds something into the queue, signaling to the condition variable that something has been placed in the queue [2].

C. Semaphores

At its core, a semaphore is an integer whose value is not allowed to fall below zero. The two main operations that are performed on a semaphore are incrementing and decrementing its value, typically implemented as the post and wait functions. When a thread attempts to decrement a semaphore whose value is zero, the calling thread will atomically block and wait for the semaphore to be incremented. For the case in which the value is above zero, it is atomically decremented. When the

This essay was submitted for ECE 3220 Project 4 for Mr. Robert Underwood.

post function is called, the value of the semaphore is atomically incremented. The thread that called post does not get involved in any following operations that result from the function call. After a call to post that increments the semaphore value from zero, one of the waiting threads will be awoken and allowed to continue execution. Often, two semaphores will be used in producer-consumer type applications. The use of two semaphores allows simple two way communication between producer and consumer. The consumer threads wait on the semaphore that the producer threads post, and vice-versa. One of the obvious benefits of using semaphores is that they can allow multiple consumers to access critical code in a controlled manner, as mutual exclusion is not involved [3].

III. Hypothesis

It is apparent that these three locking implementations have both shared and unique qualities, but how do these qualities effect runtime?

A. Shared qualities

Mutex locks and condition variables both operate on the premise of mutual exclusion. Condition variables and semaphores both do not involve busy waiting for threads that have been blocked [4]. All three of the locking implementations involve some sort of waiting queue for blocked threads, but the state of the waiting threads differs. At a lower level, condition variables and semaphores involve the operating system putting threads into a sleep state until some condition is met. The sleeping threads do not need to do the polling, as they will be awoken by some other thread. This is greatly beneficial for cases in which threads are waiting for long periods of time, as there is minimal CPU overhead.

B. Unique qualities of each of the primitives that could have an effect on runtime testing

1) Mutex Locks: The key performance feature of a mutex lock is that unlocking a locked mutex is very fast, and if the programmer can keep a given piece of code from trying to lock an already locked mutex then a mutex is generally lightweight. This act of the trying to lock an already locked mutex is called *contention* [5] and a well planned program should generally not have much. The idea is that most attempts to lock will not block. The downside of mutex performance is due to this contention. Any waiting threads are not doing anything else, resulting in wasted CPU cycles. The most vital part of mutex performance is waiting time and the waiting processes not releasing their resources.

2) Condition Variables: A condition variable as said before is always associated with a mutex lock to avoid race conditions. The strength of this primitive lies within avoiding busy waiting of all threads that are not currently within a critical section of code [1]. Specifically when only one thread can be in the critical section at a time, unlike a semaphore. This avoiding of busy threads saves wasted CPU cycles and resources.

3) Semaphores: The most unique aspect of semaphores in comparison to the other two presented primitives is the ability to allow multiple consumer threads to access critical code simultaneously. This has obvious benefits to runtime of programs that can take advantage of this. Since the producer threads are signaling the consumer threads to wake, the programmer has a lot of control over the interaction between threads. Another unique aspect of semaphores is the FIFO queue that is typically implemented for thread waiting. This can be a problem in cases where a high priority thread has to wait longer due to low priority threads arriving earlier. So when semaphores are implemented properly, they can result in very optimized CPU usage.

C. Summary

Based on the shared and unique qualities of the three locking implementations, we can make assumptions about what factors would have the greatest and least impact on program runtimes. We will construct an experiment with three tests; each test will be designed such that, based on accepted practices, one primitive should perform the most efficiently. Based on this information, the **hypotheses** for each of the locking implementations is clear, that each of the locking implementations will perform fastest on its respective test. This takes into account that all locking implementations will be run on the same machine with the same test sets each time.

1) Mutex Locks: Mutex locks will perform fastest in applications where threads are not waiting for an extended period of time and only one thread is allowed in the critical section at a time.

2) Condition Variables: Condition variables will perform fastest in applications when threads are required to wait for an extended period of time, but only one thread is still allowed in the critical section of code.

3) Semaphores: Semaphores will perform fastest in applications when a limited number of threads are allowed to access a critical section of code at the same, even if there are other threads that are waiting.

IV. Prediction

A. The **prediction** for mutex locks is as follows:

1. Change of the flag in a lock operation is minimal number of assembly instructions.
2. Mutex locks use busy-waiting, contention is not optimal.
3. The mutex test will be designed in a way to where there is minimal contention and only mutual exclusion.
4. The mutex lock will perform the fastest of the three locking implementations on its test.

B. The **prediction** for condition variables is as follows:

1. Calling threads that are locked out of a critical section can be put in a sleep state.
2. Threads in a sleep state have minimal CPU usage compared to busy-waiting threads.
3. The condition variable test will be designed such that many threads will be waiting for a condition to be met before executing critical code.

4. The condition variable will perform fastest of the three synchronization primitives on its test.

C. The **prediction** for semaphores is as follows:

1. Semaphores allow a limited number of threads to execute critical code simultaneously.
2. The other two synchronization primitives do not allow multiple threads to execute critical code simultaneously.
3. The semaphore test will be designed to where multiple threads can access the critical section at one time with no added issues.
4. The semaphore will perform the fastest of the three locking primitives on its test.

V. Testing

In order to thoroughly test each of the three locking implementations it seemed necessary to develop three unique scenarios. Each one of the scenarios would cater specifically to one of the locking implementations. This would give each primitive its best shot at its theoretical fastest runtime and application, while also comparing the other two primitives. To be clear, this would result in a total of nine tests, one test of each primitive for each scenario (mutex/condition variable/semaphore). In the case that differences of runtime of two primitives in a given application are not meaningful, it will allow for other trade-offs to be analyzed such as ease of implementation. The general testing philosophy was to run each synchronization primitive on each of the three scenarios and record runtime using C++'s chrono high_resolution_clock due to its low overhead to get an accurate representation for comparison. Each of the nine tests were ran ten times each and the results were averaged while eliminating outlying data.

1) Test System Specifications: All of the following tests were run on a fresh Ubuntu 16.04 LTS install with an Intel Core i7-4720HQ CPU at 2.6 Ghz and 16 GB of RAM.

2) Controls for Extraneous Variables: No other processes (aside from native background processes) were running at the time of testing and there wasn't any non-determinism introduced into any of the code used (i.e. rand() calls or something of the sort). Lastly, checks were implemented to make sure that all of the threads did indeed correctly complete the critical section of code and desired output was obtained.

A. Mutex Scenario

For our mutex testing, we used the scenario of incrementing a global variable within multiple threads. This test was chosen for its simple implementation, required mutual exclusion, and small section of critical code. Due to the small size of the critical code, there is minimal contention between threads, as each thread is not in the critical section for long. As stated by Frank Mueller, mutex locks also simplify execution by preventing a thread that's holding a lock from being canceled. As for how we implemented this scenario using condition variables, not much additional code was needed. After starting every thread, a pthread_cond_broadcast() call is made. Compared to the mutex implementation, condition variables are essentially just adding an extra layer in the process, as condition variables are implemented on top of a mutex. For semaphores, we had a similar situation to condition variables. A "producer" thread was created in order to manage the critical section of code properly. Since our condition variable and semaphore implementations are essentially just adding more overhead, the mutex test had the obvious runtime advantage in its testing scenario.

1) Thread Count Specifics: The mutex scenario was ran with ten threads for each primitive to not expose the busy-waiting issue associated with the mutex implementation.

B. Condition Variable Scenario

In our condition variable scenario, a large number of threads are attempting to run some critical section of code. This section contains a large number of instructions, meaning that the waiting threads will be in contention for an extended period of time [6]. This means that both the condition variable and semaphore implementation will have an advantage due to their sleep queue implementations. The mutex lock implementation of this scenario is almost identical to that of the condition variable, aside from the lack of a pthread_cond_wait() call. Similar to the mutex scenario, the semaphore was implemented using a producer thread to create mutual exclusion in the critical section. Therefore, the semaphore will again have added overhead involved. Aside from the obvious effect on runtime that results from mutex locks busy-waiting, the testing runtimes should only differ as a result of the individual implementations of the primitives themselves.

1) Thread Count Specifics: The condition variable scenario was ran with 100 threads, thus having enough threads so that busy-waiting would become a factor.

C. Semaphore Scenario

For the semaphore scenario, we knew that the test should contain a producer that allows a set of consumers to access some critical code. This was implemented using a global array of buffers. The producer thread would be able to allocate (malloc) and store some information on part of the buffer, followed by signaling the consumer that the buffer has been filled. Then one of the consumer threads can do the needed work on that buffer and free the memory. This setup is beneficial for semaphores because the producer can continue to work on empty buffers while consumers are working on the filled ones [3]. Implementing this scenario on the other two primitives required some changes to how the task was completed. For both mutex locks and condition variables, only one buffer could be worked on at any given time. In the mutex lock test, the producer and consumer threads were set up so that they would end up bouncing from one to the other until the task is completed. This behavior makes the very use of this method pointless, however this was expected since mutex locks are not made for this kind of implementation. The condition variable implementation is similar to the mutex implementation, except that the pthread_cond_signal() call gives more control to the producer. This is because the mutex implementation has no guarantee whether the producer or

consumer will grab a lock first, requiring an additional check inside each function. Overall, the semaphore has the clear advantage for expected fastest runtime since it is the only primitive that takes advantage of letting multiple threads work at the same time.

1) Thread Count Specifics: The semaphore scenario was ran with six threads working with five buffers and having 250 producer writes to the buffers.

Mutex Lock Scenario	
Synchronization Primitive	Runtime Avg [ms]
Mutex Lock	10.2068
Condition Variable	10.3968
Semaphore	10.2224

Condition Variable Scenario	
Synchronization Primitive	Runtime Avg [ms]
Mutex Lock	11.5524
Condition Variable	11.5618
Semaphore	11.798

Semaphore Scenario	
Synchronization Primitive	Runtime Avg [ms]
Mutex Lock	13.9852
Condition Variable	13.0676
Semaphore	5.4268

Figure 1. The average runtime of each synchronization primitive given each testing scenario (lower is better).

VI. Analysis

Going into this experiment with intermediate knowledge of locking implementations we had a general idea about how each primitive would behave in each scenario, as can be seen in our hypothesis. Overall, our results were in line with our original hypotheses, however the degree to which the results were displayed was different than original expectations in two of the three scenarios, these being the mutex lock and condition variable scenarios. These unexpected results most likely occurred as a result of the small scale implementations of these tests. However, we can still extract useful information from our results. After completing our testing in each scenario, we could come to a few conclusions:

A. Mutex Scenario Analysis

During the construction of our mutex lock scenario, we expected the mutex implementation to have a somewhat clear advantage. When comparing the results obtained from each of the three implementations, no clear advantage was shown for one primitive. We believe this to be a result of the simplicity and small scale of the implementation. The critical code only consisted of a single increment instruction, so the different primitives all had success with this test. Since the results we obtained for each implementation were so similar, this allows us to question other trade-offs of the scenario. When a programmer is deciding which primitive to use for a scenario of this sort, they should focus on which will be the easiest to implement in their exact case.

B. Condition Variable Scenario Analysis

For our condition variable scenario, the assumption was that condition variables would be the fastest when the critical section contains many instructions. Since condition variables have much less CPU usage during waiting compared to mutex locks, they should have the advantage in this scenario. However, the runtimes we received between the two were very similar. As was the case in the mutex scenario, we believe the main reason for this is the small scale of the implementations. As for the semaphore implementation, we saw only slightly longer runtimes. The increase was not significant enough to make a solid conclusion. Once again, there are miniscule advantages for choosing any specific primitive. However, in certain scenarios it may be easier to implement and understand one primitive over the others.

C. Semaphore Scenario Analysis

When programming the implementations of this scenario, it was clear that the semaphore would have a significant advantage. The scenario was designed around the concept of having multiple threads execute critical code simultaneously. Since semaphores are the only primitive of the three that implements this functionality, the other two implementations had to find different ways to accomplish the task. As is shown in the below graph, the runtime differences are significant. Therefore, semaphores are the obvious choice in programs that can allow multiple threads in critical code simultaneously.

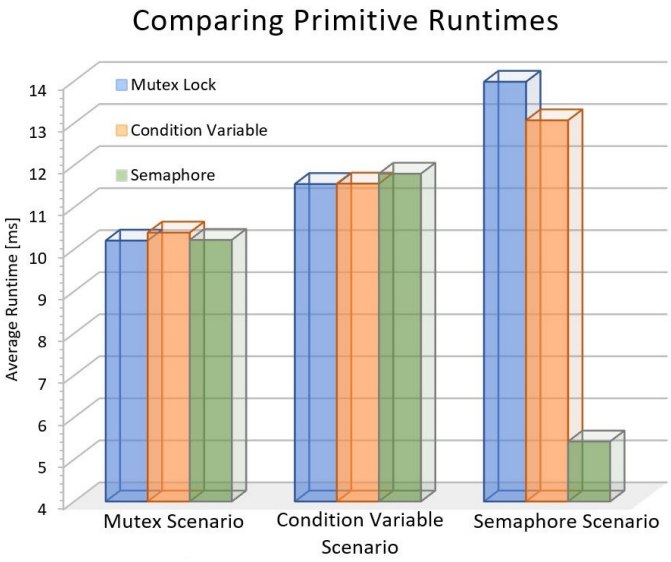

Figure 2. A visual comparison of the synchronization primitive runtimes with each graph cluster corresponding to one testing scenario (lower is better).

REFERENCES

[1] A. Birrell, J. Guttag, J. Horning, and R. Levin, *Synchronization primitives for a multiprocessor: A formal specification*. ACM, 1987, vol. 21, no. 5.

[2] "Using condition variables." [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/com.ibm.aix.genprogc/condition_variables.htm

[3] *Linux User's Manual*, December 2018.

[4] Wikipedia contributors, "Busy waiting — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Busy_waiting&oldid=861266960, 2018, [Online; accessed 7-December-2018].

[5] F. Mueller *et al.*, "A library implementation of posix threads under unix." in *USENIX Winter*, 1993, pp. 29–42.

[6] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 206–212.