Evaluating effects of active-wait synchronizations primitives

Vany Ingenzi
Undergraduate Computer Science Student
vany.ingenzi@student.uclouvain.be

***Abstract*—This report is about a study project on synchronization primitives. First, we analyze the execution of three multithreading synchronization problems using POSIX synchronization primitives and interpret the results. Then we implement active-wait synchronization primitives using inline assembly, next we use these primitives in the same multithreading synchronization problems and compare the results.**

***Index Terms*—semaphores, mutexes, multithreading, synchronization POSIX, assembly, x86, concurrency**

I. Introduction

For the course of LINFO1252 Operating Systems given by Prof. Étienne Riviere, we were asked to do a study project about synchronization primitives using three most common synchronization problems that are going to be discussed further. We are going to compare the execution time for each problem's different versions as the number of threads increase. Each problem has three versions :

1. Using POSIX[1] synchronization primitives.
2. Using test-and-set mechanism, synchronization primitives.
3. Using test-and-test-and-set mechanism, synchronization primitives.

The synchronization primitives that are going to be studied and implemented are *locks*[2] and *semaphores*[3].

II. Implementation of our synchronization primitives

A. Active-Wait Vs POSIX implemented wait

For a thread, *actively waiting* means that the thread is still runnable[4], but instead of doing some work, the thread executes an empty loop while waiting for something to happen. However, for the POSIX library, a thread that is waiting on a primitive is in a blocked[5] state. For the primitives of the POSIX library, there is support from the Operating System. For example, POSIX locks use a specific kernel language data structure called *futex* that asks the OS to block a thread till a value at an address is changed [2], etc.

[1]is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

[2]primitives used to guarantee mutual exclusion for a specific part of a code called a critical section, executed by only one thread

[3]primitives used to control access to a common resource by multiple threads

[4]it can be scheduled and given the CPU.

[5]meaning this thread won't be given the CPU time while it is still blocked.

To implement our primitives, we are going to use assembly inline[6] in order to use the instruction, `xchg`. `xchg` is an atomic instruction[7], that exchanges the content of its operands. However, this instruction is expensive in terms of performance, because it will block other processes the access to the BUS[8] so that no other thread on the system can write at the same address as one of the operands.

B. Mechanisms of active-wait locks

We are going to see two different mechanisms that are used to implement the active-wait locks.

1) test-and-set: The test-and-set mechanism idea is to simply try to get the lock continuously till you finally have it. Here's a pseudocode to illustrate our implementation :

```
do {
    try_to_have_lock();
} while(not_have_lock);
```

Listing 1. test-and-set

It is important to understand that we are going to use the `xchg` instruction in the Listing 1, within the function `try_to_have_lock()`.

2) test-and-test-and-set: The disadvantage of the test-and-set mechanism is that it repetitively execute the `xchg` but as we mentioned earlier this operation blocks other processes from using the BUS. Therefore, the test-and-test-and-set mechanism, tries to reduce the number of times the `xchg` instruction is executed. The way it achieves this is by waiting for the thread that has the lock to release the lock, then it tries to attain it. Here's a pseudocode of the test-and-test-and-set mechanism.

```
do {
    while(some_has_the_lock);
    try_to_have_lock();
} while(not_have_lock);
```

Listing 2. test-and-test-and-set

The while loop `while(someone_has_the_lock()` doesn't contain any `xchg` operation, only `try_to_have_lock()` executes `xchg` an operation.

[6]a feature of some compilers that allows low-level code written in assembly.

[7]a schedule can't interrupt the running thread until the operation is over.

[8]The hardware responsible for transfers of data and addresses from the caches to the memory.

C. Implementation of our locks

For our implementation, a lock is represented as an integer of 32 bits. An initialized lock has two possible values :

- 0, meaning it is unlocked.
- 1, meaning it has been locked by a thread.

Now, to guarantee that there's no two threads lock the exclusion primitive at the same time, we used the atomic instruction `xchg`. Therefore, the body of the *locking* function becomes setting the value of the mutex to 1, and the *unlocking* function becomes setting the value of the mutex to zero, both using `xchg`. Here's a pseudocode of our locks implementations.

Note that we use the word *mutex* in the pseudocode, to reference our locks, which are different from the POSIX mutexes primitives.

```
lock(mutex) {
    do {
        put(1, register);
        xchg(register, mutex);
    } while(register != 0);
}

unlock(mutex) {
    put(0, memory);
}
```

Listing 3. Test-And-Set lock implementation

```
lock(mutex) {
    do {
        while(mutex == 1);
        put(1, register);
        xchg(register, mutex);
    } while(register != 0);
}

unlock(mutex) {
    put(0, memory);
}
```

Listing 4. Test-And-Test-And-Set lock implementation

D. Implementation of our semaphores

To implement our semaphores, we are going to use each lock primitive implementation seen in Listing 3 and Listing 4. Our semaphore structure uses two locks (one_access, mutex) and a value initiated by the user. The pseudocode for our semaphore implementation can be found in the Listing 5.

Only one thread can execute the body of the function from `lock(sem.mutex)` ... . Therefore, only one thread can be waiting on `while(sem.val == 0)` ... , meaning if the value of the semaphore changes only one thread is going to wake up, and we are guaranteed that the thread waiting for access is going to be the one that gets access next. The other threads are going to be waiting on the `lock(sem.one_acces)` since the thread that is blocked, still has the `one_acces` lock.

```
wait(sem)
{
    lock(sem.one_access);
    lock(sem.mutex);
    if(sem.val == 0){
        unlock(sem.mutex);
        while(sem.val == 0);

        lock(sem.mutex);
        sem.val--;
        unlock(sem.mutex);
        unlock(sem.one_access);
    }else{
        sem.val--;
        unlock(sem.mutex);
        unlock(sem.one_access);
    }
}

post(sem){
    lock(sem.mutex);
    sem.val++;
    unlock(sem.mutex);
}
```

Listing 5. Semaphore implementation

E. Progress of threads using our primitives

For a continuous infinite demand of a lock and a very large number of threads, we find it hard to prove progress. The reason why we think starvation of a thread is possible in this case it is because ,for the scheduler, the thread isn't starving since it is given CPU time but in the program the thread is only spending the CPU time waiting. Nevertheless, if the number of threads that waits on the lock decreases over time, which is the case for our problems here, then we can guarantee the progress of each thread. This also applies to our semaphore implementation, since it use the lock implementation.

We didn't find a page or an article talking about the guarantee of progress for a similar case using POSIX primitives. Therefore, we made a hypothesis that under the assumption of a fair scheduler that implements a fair policy, POSIX primitives are guaranteed to progress since the scheduler can know which threads are blocked on a mutex and which one of them that has been blocked for long and unlocks the one that gets to pass. In other words because of the support POSIX threads get from the Operating System and the transparency between its primitives and the scheduler, a fair scheduler can guarantee progress of each thread.

III. Synchronization problems

A. Philosophers problem

The problem was designed to illustrate the challenges of avoiding deadlocks, a system state in which no progress is possible. For each philosopher, this is the implemented behavior :

1) Think until the left baguette is available; when it is, pick it up;
2) Think until the right baguette is available; when it is, pick it up;
3) When both baguettes are held, eat;
4) Then, put the right and left baguettes down;

5) Repeat 100 000 times.

In our case, the time of thinking and eating is instantaneous in order to highlight the cost of synchronization operations.

On the implementation side, the baguettes are locks, therefore *picking up* and *putting down* corresponds to *locking* and *unlocking* locks respectively.

B. Producers-Consumers problem

In this problem, we have a producer that puts data in a bounded buffer[9] and the consumer reads the data in the bounded buffer. Our study case has very specific constraints given by the staff regarding this problem :

- The bounded buffer is of fixed-size 8.
- Produced data should never 'overwrite' an unconsumed data.
- The total number of produced/consumed data is 1024.
- The production or processing process is simulated randomly using the command seen at the Listing 1.

```
#include <stdlib.h>
while(rand() > RAND_MAX/10000);
```

Listing 6. Simulated Producing/Consuming time

To control the access to the buffer, we use semaphores that can only let a maximum total of 8 threads consumer and producers. Locks are also used for mutual exclusion on variables.

C. Readers-Writers problem

For the third and last problem, the idea is that we have two groups of threads, *readers* and *writers*. We imagine that they share a database in which they can read and write respectively. The main constraint is that the two group of threads can't access the database at the same time, meaning a reader and a writer can't access the database at the same time. In our implementation, we have other special constraints added on by the staff:

- Each writer has to write the database 640 times.
- Each reader has to read the database 2560 times.
- The reading or writing process is instantaneous.
- Multiple readers can access the database at the same time, but only one writer can access the database.

One may wonder why writers are given a smaller number of iterations relatively to the readers. We think that this constraint was partly chosen because of the pseudocode discussed in the practical classes. For this pseudocode, a writer thread have priority on the readers threads that want to access the database after the reader has demanded access. Therefore, a higher number of writers iterations would heavily block the readers and take much more time. It is important to highlight that there's no starvation for the readers, since eventually they will be given access once the reader ahead has finished his access.

To control the access to the database we use semaphores and locks are used as well for mutual exclusion on variables.

[9]a data structure that uses a single, fixed-size buffer as if it were connected end-to-end.

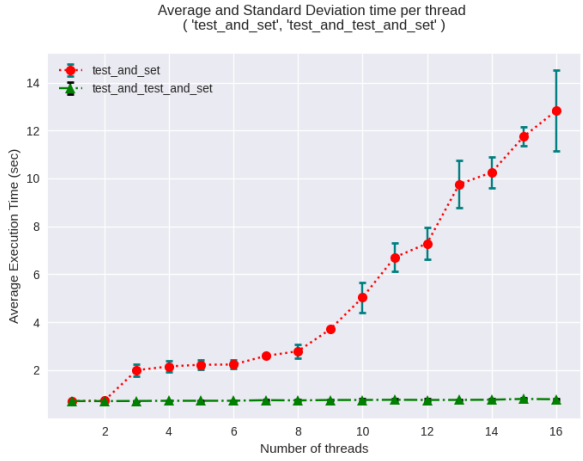

Fig. 1. The execution time mean and standard deviation for a program executing a various number of threads. These threads are continuously locking and unlocking on the same active-wait lock, we compare the Test-And-Test-And-Set mechanism or the Test-And-Set mechanism.

IV. Running Configurations for Experiences

A. Configurations

We are going to run each program with various number of threads and measure the execution time. The range of threads goes from 1 to 16, since we are running these programs on an 8-cores Fedora Virtual Machine. It is important to note that our logical cores[10] are 16.

If the number of threads given to a programmer is an odd numbe, then we add a consumer or a reader, depending on if the program is the producer-consumer problem or reader-writer problem, respectively.

For each problem and each number of threads, we run the program five times.

B. Graphs

For the visual representation, we chose error-bars. For each problem, we are going to observe the average execution time as a function of the number of threads. It is important to note that the vertical lines don't measure the error but the standard deviation, since for each configuration we ran the program five times.

V. Interpretation

A. Philosophers

For the Fig.[2], we observe a linear regression between the number of threads and the average execution time. Furthermore, the standard deviation seems to stay constant with minor difference, except at the very end.

The positive linearity can be explained by the fact that the execution time is mostly the time took for synchronization operations since, as discussed in the [Philosophers problem] subsection, there's no processing time. The synchronization

[10]the number of physical cores times the number of threads that can run on each core through the use of hyperthreading. In our case, 2 threads.

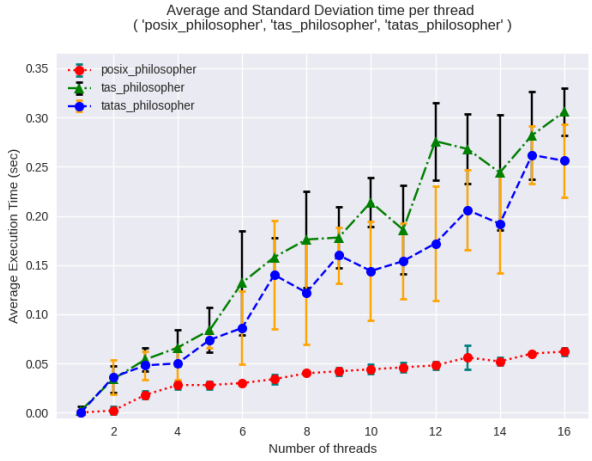

Fig. 2. The execution time mean and standard deviation for the philosopher problem using the three different primitives, with the different number of threads.

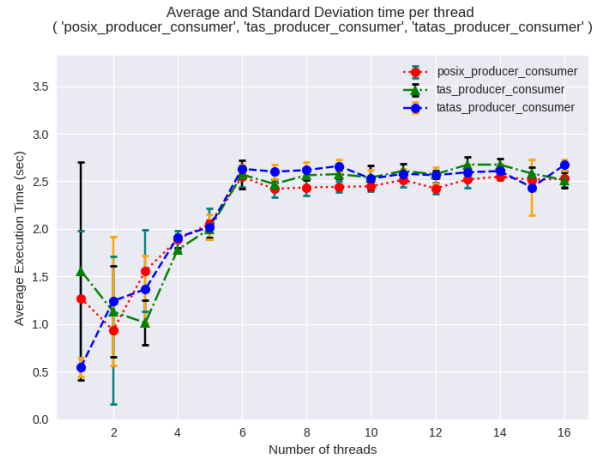

Fig. 3. The execution time mean and standard deviation for the producer-consumer problem using the three different primitives, with the different number of threads.

operations time in this case is the total time that threads wait on the locks, since at most $\frac{N}{2}$ threads have access in their respective critical zones. Therefore, the more the threads, the more amount of time threads spend waiting, which result in more time for synchronization. Thus, having an execution time that increases with the number of threads. Overall, the total amount of work increase with the number of threads, since each thread has to execute 100.000 cycles.

When we compare the average execution time of the three implementation versions respectively. We observe that the active-wait versions takes more execution time overall, and the execution increases more significantly than for the POSIX implementation. This is because since we are using active-wait locks the waiting threads are still chosen by the scheduler, which limits the CPU time of the threads that have access to the critical sections, resulting in more execution time. The Test-And-Test-And-Set (tatas) version performs better than the

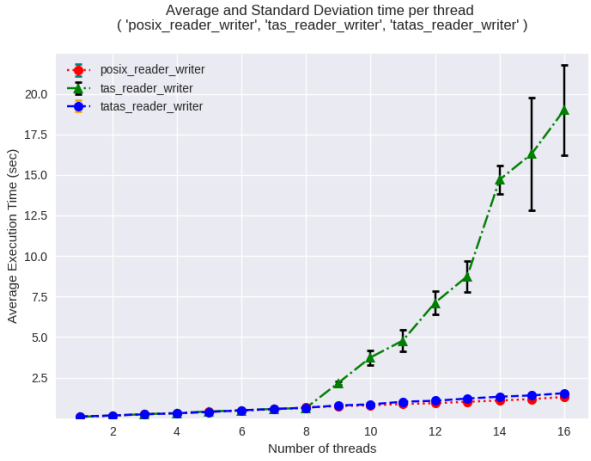

Fig. 4. The execution time mean and standard deviation for the reader-writer problem using the three different primitives, with the different number of threads.

Test-And-Set (tas) since Test-And-Test-And-Set uses less the exchange operation.

B. Consumer-Producer

For the *bounded buffer* problem Fig.[3], we observe an overall constant average execution time ($\approx 2.5sec$) regardless of the number of threads. To be more precise the execution time increase slightly from 2, till 8, and then it alternates around ($\approx 2.5sec$). This observation is identical in all versions.

The standard deviation is so high at the beginning till 8, for each measurement stays constant, this is observable in all implementations.

In order to understand why we have a constant execution time in general, we have to interpret it step after step.

Firstly, all the producers have one common goal to produce 1024 elements and the consumers also have one common goal to consume 1024 elements. So the program ends when all producers and consumers have reached their common goal.

Secondly, above 8 number of threads, it is important to note that when we increase the number of threads, we are increasing the waiting time for threads wanting to pass the semaphores. Nevertheless, remember that there are already exactly 8 threads in the critical section contributing to the common goal. So overall there's still progress in the program.

Finally, we conclude that even though the number of threads increase, at a given time we are going to have exactly 8 threads that are contributing to the common goal, in other words they passed the gateways. Thus, the reason why, even if the number of threads increases, the average execution time is almost the same. It is important to note that the 8 threads working aren't always the same, under the assumption that our scheduler[11] is fair, normally all the threads get to pass the semaphores at least once.

We observe that the active-wait implementations have the same or even a better performance than POSIX implemen-

[11] meaning every thread is given CPU time soon or later.

tation for a number of threads less than the number of the computer's logical cores.

The reason our implementation seems to be more efficient for a number of threads less than 8 is mostly due to the fact that threads using our primitives don't have to use system calls. These system calls block the threads and may take up to a certain number of CPU cycles(time), which is the case for threads using POSIX primitives.

For a number of threads less or equal to 8, every thread is running on an independent core, therefore they are not interleaving. We don't observe a similar phenomenon for the active-wait philosopher's problem and the reader-writer's problem because there's no cooperation.

C. Reader-Writer

The Fig.[4] shows a strong linear function of time in relation to the number of threads for both at Test-And-Test-And-Set implementation and POSIX implementation. However, for the Test-And-Set implementation, the function is linear till 8 threads but afterwards it starts increasing drastically. We also observe that the standard deviations are really low for POSIX and Test-And-Test-And-Set implementation, but they appear to get bigger for the Test-And-Set implementation as the number of threads increase.

To interpret the result, we are going to proceed in steps.

Firstly, as explained in the [Readers-Writers problem] subsection, if a writer is waiting for the critical section, no reader after the waiting writer shall get access to the critical section before the writer does. So there's a favoring of writers over readers.

Secondly, we have strong constraints about accessing the critical section. These constraints are explained in the subsection III-B. When a writer is accessing the database, it is important to notice how these restrictions forces all the other threads to wait on locks and semaphores, resulting in a big amount of execution time. These constraints are the reason why when we compare, with the other two graphs Fig.[2] and Fig[3], this problem is the one that takes the most amount of execution time for the same amount of threads.

Finally, the increase of the number of threads, increases the total amount of work in general since each thread has its individual thread has its own number of iteration. Even more, the increase of the amount of threads increases the waiting time for each thread, given the strong constraints about the favoring writers and maximum number of threads accessing the database.

It is an important thing to notice is that for a number of threads less or equal to the number of cores, the execution time of all versions is the same. On the other-side, when the number of threads is greater than 8, the Test-And-Set implementation execution time increases drastically and for the Test-And-Test-And-Set we have very moderate increase. The reason why we are observing this phenomenon is due to the number of the computer's physical cores. When the number of threads is less or equal to 8, this means each thread is executing on it's core. Thus, the waiting threads aren't using CPU that should be given to other threads that are in the critical sections. Unfortunately, when the number of threads increases, the OS starts to interleave threads on the same cores, resulting in the consumption of CPU by waiting threads that prevent threads in the critical section to do their work.

D. Test-And-Set lock vs Test-And-Test-And-Set lock

In the Fig.[1], we observe a clear gap that grows exponentially between the performance of *test-and-set* and *test-and-test-and-set* mechanisms. This confirms our hypothesis that the operation `xchg` was expensive in terms of performance, since the more we execute it, the less performance we get with *test-and-set*. Thus, why the *Test-And-Test-And-Set* was a better option to use when implementing our active-wait primitives.

VI. Conclusion

Within this study project, I have learned various aspects about synchronization and multithreaded programs.

Firstly, the atomic operation `xchg`. I had briefly covered the atomic operation in other courses but hadn't had the chance to deeply understand and visualize it's negative impact on performance. Nonetheless, I saw it's advantage of making synchronization of threads easier than using other traditional algorithms like the Bakery Algorithm.

Secondly, I learned that before implementing multi-threading programs, it is important to analyze the number of threads because using many threads doesn't always mean that the program is going to be executed faster. Furthermore, the impact of the computer's hardware architecture can be significant.

At last, I've learned a soft skill about analyzing program executions, finding correct and coherent interpretation that contribute to my knowledge or somebody's else. This is a something I dream of doing very often in my career as a researcher.

References

[1] https://uclouvain.be/cours-2021-linfo1252

[2] https://attractivechaos.wordpress.com/2011/10/06/multi-threaded-programming-efficiency-of-locking/