# CS5510: Multiprocessor Programming : Homework V

Sergio Sainz

October 2019

## 1 Part II : Implementation [75 points]

### 1.1 [10 points] Experimenting with Backoff Strategies Implement and evaluate the following Backoff strategies for the Backoff Lock. The Backoff Lock should create the necessary backoff mecha- nism using the Factory Design Pattern, which the provided template illustrates.

- Exponential: The base delay time is calculated as 2a, where a is the number of unsuccessful attempts that have been made.
- Fibonacci: The base delay time is returned as the Fibonacci number corresponding to the current attempt.
- Fixed: The base delay time is calculated as a fixed value determined by the attempt number. The base delay time is calculated as a fixed value determined by the attempt number.
- Linear: The base delay time is equal to the attempt count.
- Polynomial: The base delay time is calculated as ae, where a is the number of unsuccessful attempts that have been made, e is the exponent configured for the strategy. Pick a value for e that you think is acceptable for your experiment.

**Answer**
Below please find the logs for the experiments. Each strategy was run with three thread numbers 8,16,32 using the *normal* test. Below the average number at the end of the log (avg) is the average of the last four executions in *milliseconds*. Iter is the increments done per thread (for all experiments the total increments is 64000).

```
1  [BackoffLock][Exponential][Threads = 8][Iter per thread = 8000]avg: 5.75
2  [BackoffLock][Exponential][Threads = 16][Iter per thread = 4000]avg: 5.5
3  [BackoffLock][Exponential][Threads = 32][Iter per thread = 2000]avg: 4.75
4  [BackoffLock][Exponential][Threads = 64][Iter per thread = 1000]avg: 2.25
5  [BackoffLock][Fibonacci][Threads = 8][Iter per thread = 8000]avg: 3.5
6  [BackoffLock][Fibonacci][Threads = 16][Iter per thread = 4000]avg: 3.75
7  [BackoffLock][Fibonacci][Threads = 32][Iter per thread = 2000]avg: 3.0
8  [BackoffLock][Fibonacci][Threads = 64][Iter per thread = 1000]avg: 2.0
9  [BackoffLock][Fixed][Threads = 8][Iter per thread = 8000]avg: 2.75
10 [BackoffLock][Fixed][Threads = 16][Iter per thread = 4000]avg: 3.25
11 [BackoffLock][Fixed][Threads = 32][Iter per thread = 2000]avg: 2.5
12 [BackoffLock][Fixed][Threads = 64][Iter per thread = 1000]avg: 1.5
13 [BackoffLock][Linear][Threads = 8][Iter per thread = 8000]avg: 264.0
14 [BackoffLock][Linear][Threads = 16][Iter per thread = 4000]avg: 273.25
15 [BackoffLock][Linear][Threads = 32][Iter per thread = 2000]avg: 212.25
16 [BackoffLock][Linear][Threads = 64][Iter per thread = 1000]avg: 89.5
17 [BackoffLock][Poly][Threads = 8][Iter per thread = 8000]avg: 3.0
18 [BackoffLock][Poly][Threads = 16][Iter per thread = 4000]avg: 3.75
19 [BackoffLock][Poly][Threads = 32][Iter per thread = 2000]avg: 3.0
20 [BackoffLock][Poly][Threads = 64][Iter per thread = 1000]avg: 2.0
```

As we can see from line 12, the combination of strategy **fixed** with 64 threads had the fastest average (1.5 milliseconds). Meanwhile **linear** strategy had the slowest average with 8 threads (264).

Reason for this behavior, I believe, is because the slow locks are backing off for too little time for the lock to become available. Meanwhile the locks that perform better backoff for longer time comparatively. For example, strategy linear keeps waiting on multiples of 10 milliseconds, thus it waits 10, then 20, then 30 and so on every ten milliseconds. Meanwhile the fastest strategy waits like this: 1, 3, 10, 20, 50, 100, 200, 500, 1000, 3033, 5000. As you can see the difference with the previous value increases in exponential manner, but pretty soon it goes to level of 5000 milliseconds. You may ask, why is exponential strategy performing worse than fixed strategy then. The answer , I believe is in the sequence of exponential strategy is power of 2, so its sequence is: 2,4,8,16,32,64,128,256,521,1024,2048. As you can see it grows more slowly than the fixed strategy. So meanwhile the waiting time of fifth attempt of fixed strategy is 50, the waiting time of fifth attempt of exponential is 32. Waiting time in the tenth attempt of fixed strategy is 3033 and of exponential strategy is 1024.

Thus, I believe the reason why some strategies are better than others is because some strategies grow backoff time much faster than others and also grow backoff time quickly in their very early attempts (attempt number smaller than ¡5).

## 1.2 [10 points] Barrier

Imagine n threads, each of which executes method foo() followed by method bar(). Suppose we want to make sure that no thread starts bar() until all threads have finished foo(). For this kind of synchronization, we place a barrier between foo() and bar(). **First barrier** implementation: We have a counter protected by a test-and-test-and-set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n. **Second barrier** implementation: We have an n-element array b, all 0. Thread zero sets b[0] to 1. Every thread i, for 0 ¡ i  n1, spins until b[i  1] is 1, sets b[i] to 1, and waits until b[n  1] becomes 2, at which point it proceeds to leave the barrier. Thread b[n  1], upon detecting that b[n  1] is 1, sets b[n  1] to 2 and leaves the barrier. Implement the barriers and measure the barrier time (i.e., time between foo() and bar()) as a function of number of threads, for both barriers, and plot the data. You should complete the implementation of the "barrier" case statement in Benchmark.java class. Evaluate on both your local machine and Rlogin. Compare (in 3-4 sentences) the behavior of these two barrier implementations given in Problem 2 of Section 2 on a bus-based cache-coherent architecture. Explain which approach you expect will perform better under low load and high load.

**Answer**

Please find in 1 the plot for benchmarking test results:

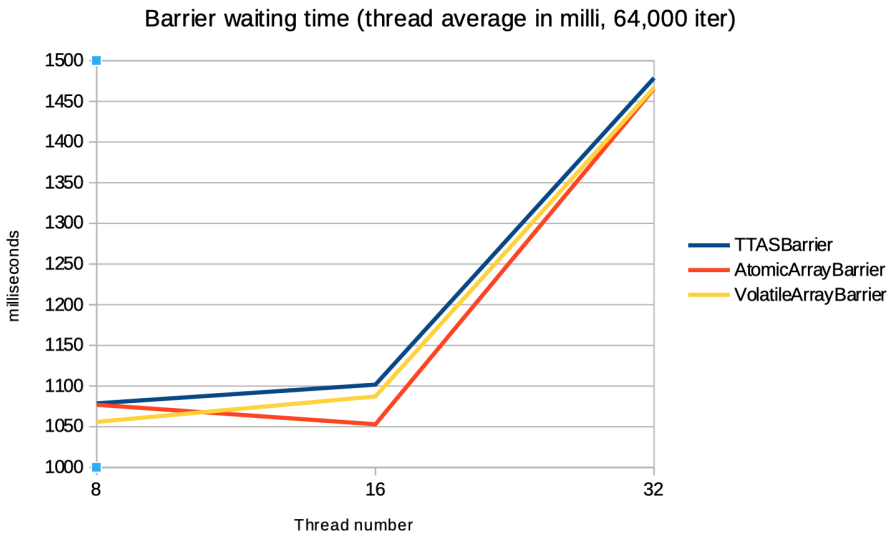


Figure 1:

The slight better performance for the array barriers could be explained due to *cache-coherence* phenomenom. Cache-coherence happens when many threads

are waiting on the same memory location and when memory location is updated, the waiting thread's cached value will be invalidated and therefore they will need to read the new value from main memory. But because they are so many threads, their memory read request will flood the address bus thus causing more delay.
Benefit of array barrier is that the threads are not waiting on the same location of memory exactly, thus cache coherence is avoided somewhat (still there is problem of false-sharing that occurs whenever adjacent locations are also read whenever a memory read happens. Thus cache coherence avoidance is culprit why array barrier is faster.

### 1.3 *[5 points] Lock Evaluation

Implement the following locks below (you may reuse the provided code in the template) and measure the time to execute an empty critical section as a function of number of threads for all locks, and plot the data (similar to Figure 7.4). Perform your evaluation on RLogin. Note: You will actually measure throughput across a few seconds (2-3 seconds), and then you can calculate the average waiting time, which you need to plot. Don't forget about warm-up time. The Benchmark.runEmptyCS.java may be useful for this purpose

1. TestAndSet spin lock
2. TestTestAndSet spin lock
3. Backoff spin lock with the best strategy
4. CLH queue lock
5. MCS queue lock

**Answer** : The results are below figure 2. The first execution (among 5) is removed and thus average is among 4 samples:

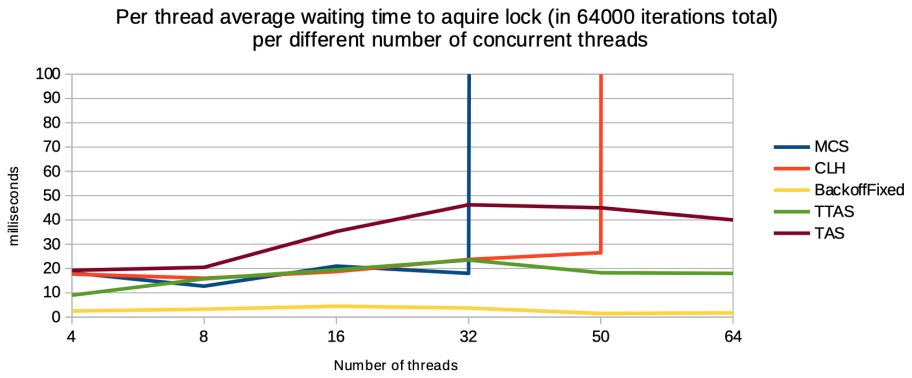


Figure 2:

## 1.4 [15 points] Priority-based Queue Lock

The queue locks provide FCFS fairness, which is a useful property for many applications. But some applications may attach priorities to threads (e.g., some thread may have a higher priority than others). Design and implement a PriorityQueueLock that ensures that the highest priority thread that is waiting for the lock is always granted access to the critical section than all other waiting threads. Note that we don't care about the priority of the thread that is currently holding the lock, which may or may not have a higher priority than the highest priority waiting thread. Thus, whenever the lock is released, the highest priority waiting thread is granted access first. (If you include the priority of the lock holder in the total priority ordering of the lock, then it may require aborting the lock holder, executing roll-back logic, etc., which is outside the scope of this homework.) Beside the lock() and unlock() methods for your lock, also implement a trylock() method that attempts to acquire the lock (respecting thread priorities), and if it can't acquire it within a predefined amount of time (constructor argument in ms), it fails/aborts. trylock() will return a boolean value indicating whether the lock was successfully acquired. For each thread, measure the waiting time before starting critical section execution, and multiply it by its priority (where priority=1 is the highest priority, and priority=5 is the lowest). Obtain the average of your calculated data as a function of number of threads and plot the data (similar to Figure 7.4). Include both PriorityQueueLock (only the lock() method) and FCFS CLH lock in the plot. Benchmark.runNormal may be useful for this purpose. Evaluate on both your local machine and RLogin. Hints: Don't try to implement the PriorityQueueLock the same way as other queue-based locks are implemented. Instead, use the principles be- hind those locks. This is easier if you use Java's PriorityBlockingQueue, and let threads spin on their own node.

**Answer** Please find below the results for the benchmark. We notice that the CLHLock is actually faster than the priority queue lock. I believe reason

stems from fact that priority queue lock has more volatile member variables per node and thus creates more memory barriers. Plot for the rlogin machine is in figure 3. And Plot for the local (laptop) machine is in figure 4.

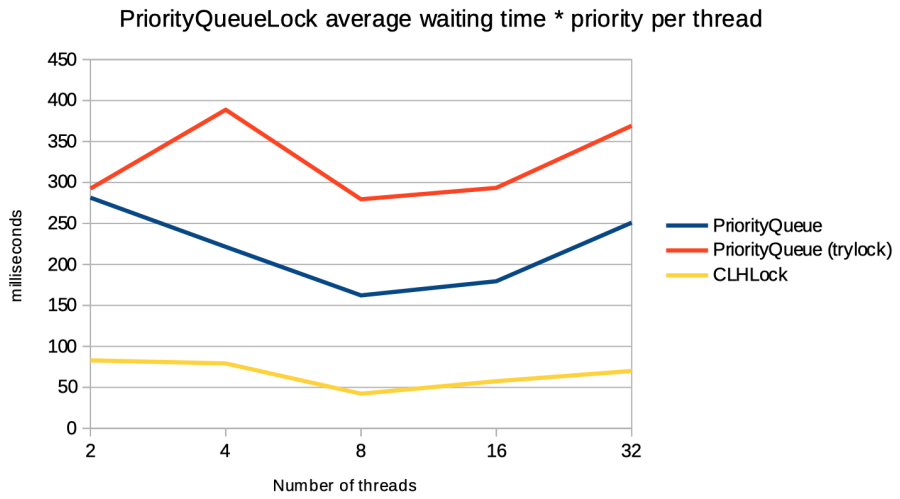


Figure 3:

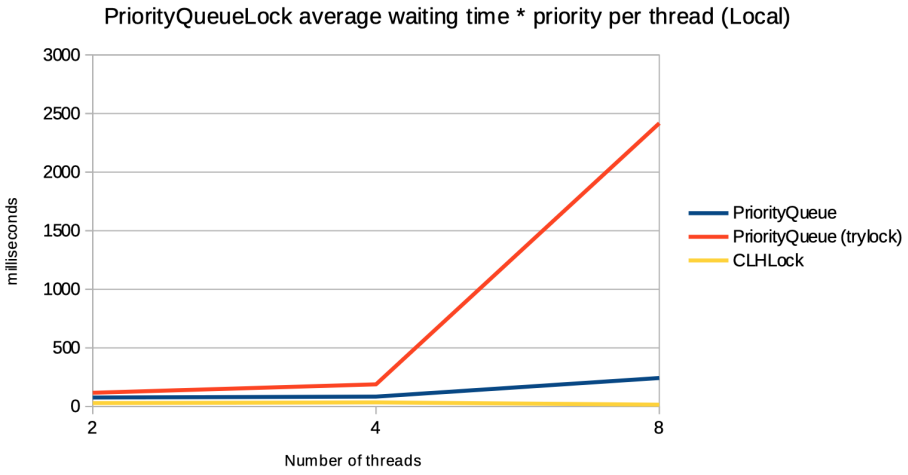


Figure 4:

## 1.5 [15 points] Spin Sleep Lock

Quote from Section 7.1 of course textbook: *Spin-lock algorithms impose trade-offs. By contrast, backoff locks support trivial timeout protocols, but are inherently not scalable, and may have slow lock release if timeout parameters are not well-tuned.* In this problem, your objective is to improve the performance of Spin locks using some clever techniques. Spin locks perform poorly at high contention as numerous threads are contending for the same memory region. Despite, only one thread may succeed. Thus, the idea is to make only a subset of threads spin on a lock, while other threads waiting for the lock are sleeping. We call such a lock SpinSleepLock. Design and implement SpinSleepLock, where at most maxSpin threads are spinning while other contending threads are sleeping. As a lock holder releases the lock, a sleeping thread, if any, is woken up to spin on the lock. Discuss the merits of such a lock and validate your hypothesis via experimentation. Compare your results with the TAS, TTAS, Backoff, ALock, CLS and MCS locks. Be sure to use the best performing backoff strategy for the Backoff Lock. What kind of machines are best suited for such a lock? Rlogin or your laptop? Perform your evaluation accordingly (meaning you don't have to evaluate on both hardware, but only the most effective one). Benchmark.runLongCS.java may be useful for this purpose. Hint: You may find Java object's wait() and notify() methods useful for sleeping and waking up threads.

**Answer**

Please find below the benchmarking comparison in figure 5. As in other benchmarks, it shows the average time it took to complete work before obtain lock and after obtain lock.

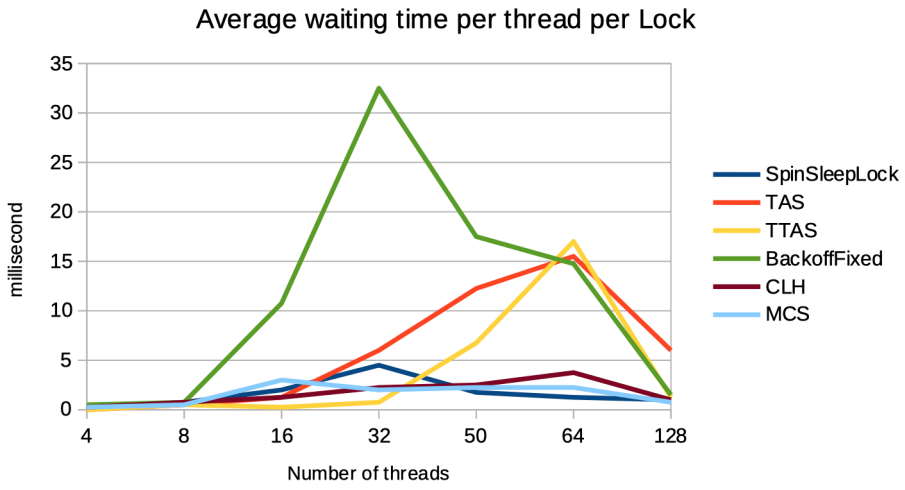


Figure 5: Tested in dogwood with 39 processors each with 10 cores. 640,000 iterations

The reason why spin sleep lock performs better than the MCS, CLH lock (it is based on MCSLock) is because there are no cores spinning and hogging the cores just spinning. Thus it leaves room for the few threads working on the problems to focus on the computations. While the reminder threads are sleep. And it is faster than the backoff lock because the backoff lock is not efficient in waking up the next thread once one thread finishes. Thus, in the backoff lock once a thread finishes it does not inform anyone that their turn is up. It just happens that some threads awake by themselves after some preiod of time and therefore there is some unused time after the working thread finishes its work and before the next thread wakes up. Thus backoff lock is not as fast as the spin sleep lock.

## 1.6 [20 Points] Simple Hierarchical Lock

Of the two hierarchical locks discussed in the textbook, the HBO Lock has the potential to starve threads from other clusters, while the HCLH Lock is more complex.
Implement a simple form of hierarchical lock, SimpleHLock, that is better than HBO Lock, but by reusing existing locks. The core idea of SimpleHLock is to have one local lock per cluster, and one global lock.
The lock works as follows:

- On lock, the first time the thread grabs its local lock and then the global lock.

- On unlock, the thread releases the local lock, and if there is no thread waiting for the local lock, it releases the global lock (or for fairness it releases the global lock after allowing BATCH_COUNT local threads consecutively)
- On lock, if the thread that unlocks the local lock doesn't release the global lock, then the locker thread doesn't need to grab the global lock and just grabs the local one.

It is your responsibility to choose the right lock algorithms for the global and local locks. Evaluate your implementation on RLogin and compare with TAS, TTAS, Backoff, ALock, CLS, MCS, HBO and HCLH locks. Vary the number of clusters between 2 and 4. Implement your own method called runClusterCS in Benchmark.java and add a switch case with "Cluster" as the matching string to invoke this new method. Your benchmark method should take in the number of cluster as arguments (in addition to other arguments you may have) and pass it to the appropriate class(es). Hint: You may find hw4.utils.ThreadCluster class useful for dealing with clusters, but you may need to modify some parts of this class.

**Answer**

In this case decided to implement the global lock using Test And Set lock because at any given time there are at most 3 threads waiting for it (plus the one owning the lock). Thus, TAS, is the most effective without cache-coherence blocking.

On the local locks, I initially thought spin sleep locks as they have highest throughput (lowest waiting time per thread). Nevertheless in practical case the SimpleHLock with spin sleep locks as their local locks did not perform as well as other locks such as TTAS or CLHLock lock in the *dogwood* machine. This machine has 39 cores each with 10 cores. TTAS will not scale up as we increase threads. CLHLock will scale better, thus I pick CLHLock as the kind of lock for local locks.

There are two comparisons below. Comparison in figure ,6, compares only cluster locks SimpleHLock and HBOLock. We can clearly see that SimpleHLock is better because is more efficient of the critical section (once one thread finishes another thread takes over). Meanwhile HBOLock has more underutilization , there is a backoff threshold. And if a thread finishes and no other thread is checking to enter critical section the critical section will remain empty for as long as it takes for another thread to wake up from the backoff.

The second comparison ,7, compares all threads. The SimpleHLock is as efficient as the queue locks such as MCS , CLH and Spin sleep locks. And is faster than TTAS, TAS , Backoff as well as HBO locks. This is in line with the expectation that queue locks cause less cache coherence traffic and escale better.

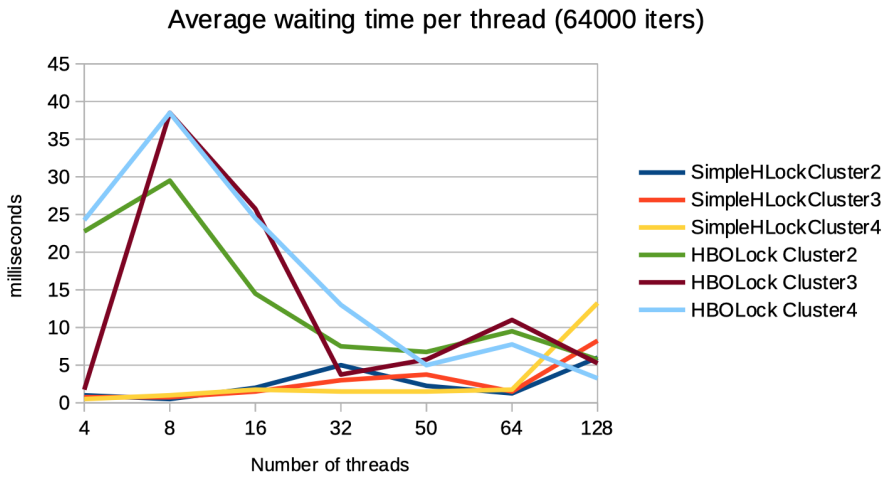


Figure 6: Tested in dogwood with 39 processors each with 10 cores.

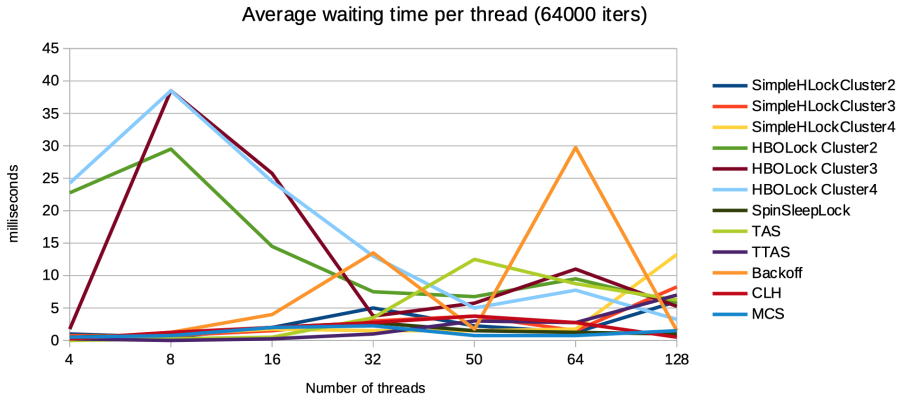


Figure 7: Tested in dogwood with 39 processors each with 10 cores.