



## Multiprocessor Programming Course

Dario Bahena Tapia - Ricardo Zavaleta Vazquez - Isai Barajas Cicourel  
October 2015

# Chapter 1

## Mutual Exclusion

### 1.1 Problem Definition

## 1.2 Peterson Algorithm

### 1.2.1 Particular Case

In this experiment, the particular case we are trying to study is again mutual exclusion where two threads try to use a shared resource.

We are trying to do so in such a way that there is no starvation.

### 1.2.2 Solution

During the lecture we discussed that Peterson algorithm takes the best of other two algorithms (which we called the LockOne and the LockTwo) to create a better one. The LockOne class that we discussed used an approach where mutual exclusion was attempted using a flag which indicates that a given thread has the intention of using a resource. We concluded that it satisfies the mutual exclusion but it can potentially dead lock unless executions are not interleaved.

On the other hand, the LockTwo class achieved mutual exclusion by forcing the threads let the other one execute first. However, this approach also had dead lock problems when one thread ran completely before the other one. So we observe that both algorithms complement each other.

The Peterson algorithm combines these two methods to achieve a starvation-free algorithm. It uses both the *flags* approach and the *victimization* approach.

### 1.2.3 Experiment Description

Now let's discuss how the proposed test case excersices the Peterson algorithm.

The test creates 2 threads that need to be coordinate in order to increment a common counter. Both threads have to cooperate to increase this counter from 0 to 1024. Each of the threads will increase by one the counter 512 times. The expected result is that regardless of the order in which each thread executes the increment, at the end the counter must stay at 1024. If that is not the case, then it means that mutual exclusion did not work.

These are the details of the system we used to run the experiments:

1. Processor: Intel Core i5 @2.5 GHz. 2 Cores.

2. L2 Cache per Core: 256 KB
3. L3 Cache: 3 MB
4. System Memory: 16 GB

### 1.2.4 Sample Results

Figure 1.2 shows the output that we observed in the netbeans window:

```
compile-test-single:
Created dir: /Users/ricardo/Desktop/ch02/Mutex/build/test/results
Testsuite: mutex.PetersonTest
parallel
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.108 sec

----- Standard Output -----
parallel
-----
test-single:
BUILD SUCCESSFUL (total time: 5 seconds)
```

Figure 1.1: Output of the junit test

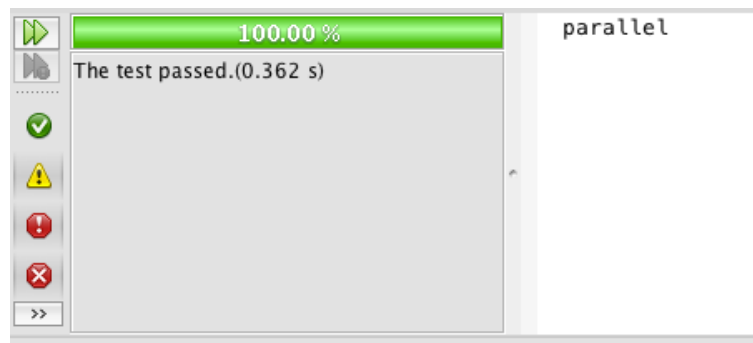


Figure 1.2: The Peterson test passed

### 1.2.5 Interpretation

The results in the proposed system were all OK. In all cases, the threads were able to cooperate to increase the counter to 1024.

One thing that is worth mentioning is that this algorithm has a limitation. The limitation is that it only works for two threads. In a later experiment we will discover other algorithms that allow the coordination of more than two threads.

## 1.3 Bakery Test

### 1.3.1 Particular Case

In this experiment, we are dealing with the problem of mutual exclusion with a number of participants  $> 2$ .

We have already seen that Peterson Algorithm works for two threads. It guarantees mutual exclusion and is starvation free. Therefore, the algorithm is deadlock free. Now let us focus in an algorithm that can be used to coordinate more threads.

### 1.3.2 Solution

The algorithm that we will play with is called Bakery. Its name comes from the fact that it is similar to the protocol used in bakeries where one enters the store and picks a number that indicates the order in which each client will be attended.

In the algorithm, the fact of entering the store is done by setting a flag. After that, the thread calculates the next number in the machine.

The algorithm then chooses the next number to be attended. When that happens, the thread is allowed to enter the critical section.

One thing that we ought to mention is that threads can have the same number assigned. If that is the case, the next in the line is decided using a lexicographical ordering of the threads ids.

### 1.3.3 Experiment Description

Now let us explain how the experiments work. In this case we have 8 threads increasing a counter from 0 to 1024. Each thread will increase the counter 128 times. At the end, the counter must remain in 1024. If that is not the case, then there is a problem with the mutual exclusion.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

### 1.3.4 Sample Results

Figure 1.3 and 1.4 shows the output that we observed in the netbeans window:

```
compile-test-single:  
Testsuite: mutex.BakeryTest  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 sec  
  
test-single:  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 1.3: Output of the junit test

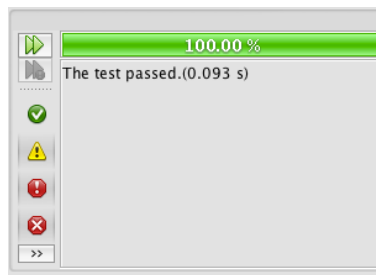


Figure 1.4: The Bakery test passed

### 1.3.5 Interpretation

The results in the proposed system were all OK. In all cases, the 8 threads were able to cooperate to increase the counter to 1024.

## Chapter 2

# Foundations of Shared Memory



## 2.1 Safe Boolean MRSW Register

### 2.1.1 Particular Case

In class we saw the difference between Safe, Regular and Atomic Registers. In this exercise we are experimenting with a Safe Boolean Register. In particular, we are trying a register that allows one single writer whose written value can be examined by multiple readers. We need to remember that Safe registers are those that :

- If a read does not overlap with a write, then the read returns the last written value
- If a read overlaps with a write, then the read can return any value in the domain of the register's values

### 2.1.2 Solution

One way to achieve this behaviour is using a SRSW Safe register. The code that is provided with the book does this by having a boolean SRSW register per thread, in other words, it is an array of boolean values. When doing a write, the writer iterates over this array and writes the new value. The readers simply access its assigned slot in the array (based on the thread id) and return the stored value.

### 2.1.3 Experiment Description

This program provides two test cases. The first one is a sequential test and the second one is a parallel test. The former simply calls a write and then a read from the same thread. This is not rocket science. The reader must retrieve what the writer wrote.

The second test case is slightly more interesting. The writer writes first one value and then another one. After that, 8 reader threads are started and they read the last written value. According to the rules of a safe register, all threads must read the same value because the reads and the writes did not overlap.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.

- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

## 2.1.4 Sample Results

We found out that the tests failed occasionally. The manifestation of such failures were two:

1. The junit framework reports a failure in one of the tests as shown in figure 2.1
2. The junit doesn't report a failure, however the test output shows an *Out of Bounds exception*. This is shown in figure 2.2

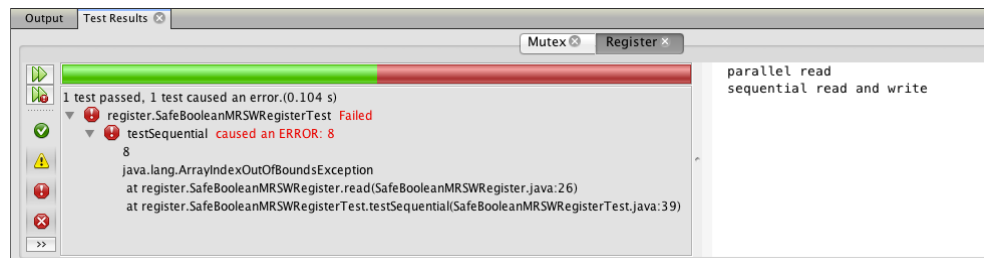


Figure 2.1: First type of failure

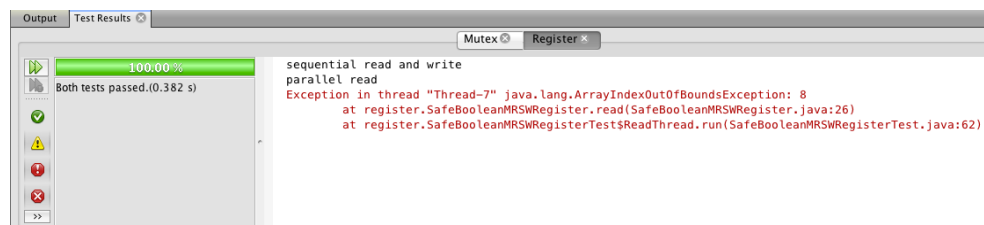


Figure 2.2: Second type of failure

So, the way to fix this problem was to make sure that in each test case the threads ids are reset calling the static method *ThreadID.reset()*. With this hack, the tests passed (See figure 2.3).

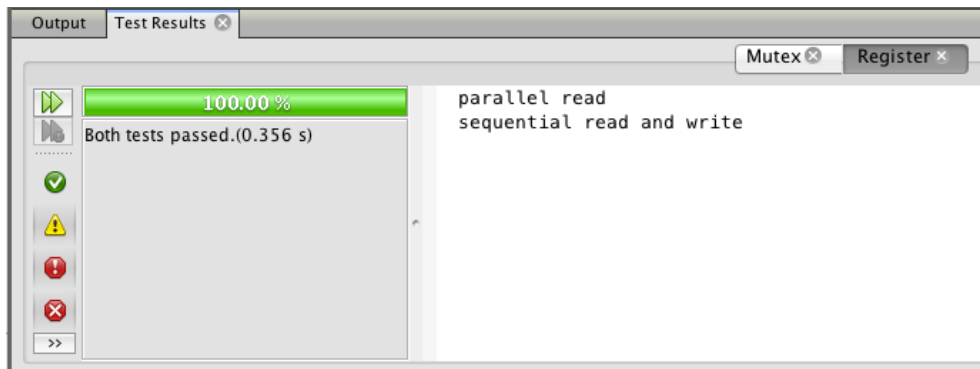


Figure 2.3: Output after fix

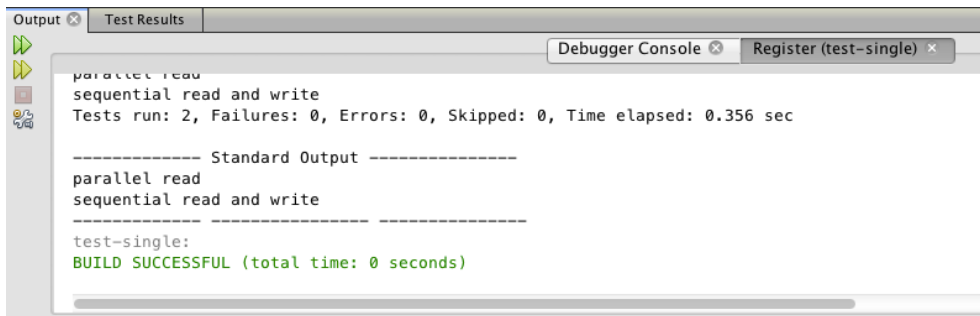


Figure 2.4: Output after fix of threadIds

### 2.1.5 Interpretation

In this experiment we saw a way of implementing Safe Multi-Reader Single-Writer registers. These registers were able of storing boolean values.

Unfortunately, none of the test cases excersice the case where we have concurrent writes and reads. Hence, this experiment only excersiced one of the two rules of a safe register.

## 2.2 Atomic MRSW Register

### 2.2.1 Particular Case

The characteristics of this type of register are:

- It should never happen that  $R^i \rightarrow W^i$
- It should never happen that for any  $j$ ,  $W^i \rightarrow W^j \rightarrow R^j$
- If  $R^i \rightarrow R^j$  then  $i \leq j$

The book proposes to do this by using multiple SRSW registers.

### 2.2.2 Solution

The algorithm then requires a 2 dimensional table. When a writer decides to update the register, it has to update the values in cells  $A[i][i]$ , where  $i$  is the thread id. Apart from writing the value, it has to also update the cell with a timestamp.

In the other hand, when a reader wants to read from the register, it checks the timestamp of the cell  $A[i][i]$ . After that, it has to check other cells in the same column (ie, cells  $A[x][i]$ ) to see if there has been an update in between. That is done by comparing the timestamps. If there is a newer timestamp, the reader has then to update all cells in its row (ie  $A[i][x]$ ). This is the way to indicate subsequent readers which version of the value has the previous reader retrieved.

It is easy to see that the first property is satisfied because there is no way to read a future value. In order to read a value, this value has to be written before.

The second property is also satisfied because when a reader reads the value, it reads the one with the most recent timestamp by scanning the timestamps in its corresponding column. This reader also helps subsequent readers by updating all cells in its row. Forcing property 3 to be satisfied as well.

### 2.2.3 Experiment Description

The two test cases presented demonstrates the same as previous examples so we will not describe the experiment further.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

## 2.2.4 Sample Results

As in previous examples, we identified that the test fails from time to time. Figure 2.5 shows the output when the failure is seen.

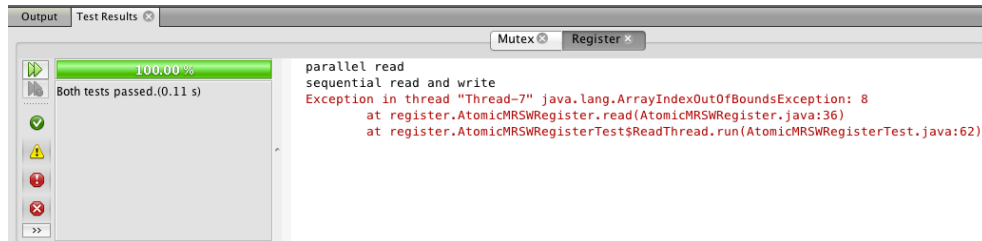


Figure 2.5: First type of failure

Again, the fix for these failures was to reset the thread ids.

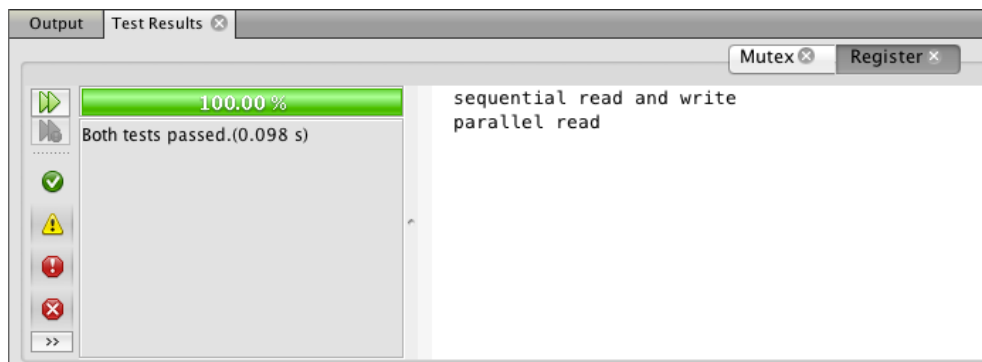
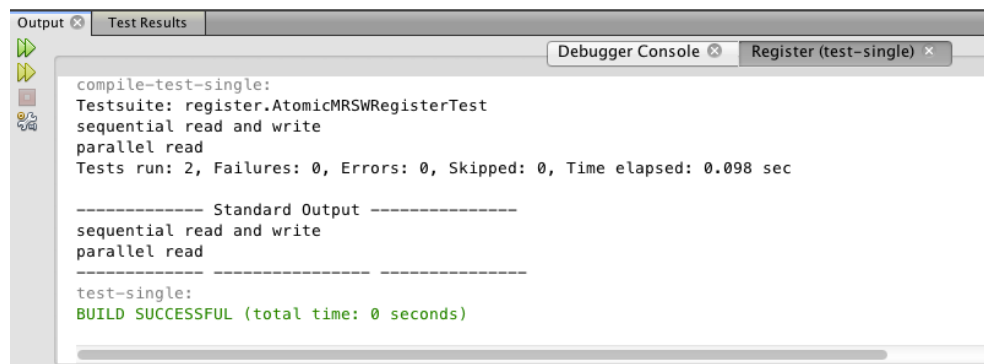


Figure 2.6: Output after fix



The screenshot shows a debugger console window with a tab titled 'Register (test-single)'. The console output is as follows:

```
compile-test-single:
Testsuite: register.AtomicMRSWRegisterTest
sequential read and write
parallel read
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.098 sec

----- Standard Output -----
sequential read and write
parallel read
-----

test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 2.7: Output after fix of threadIds

### 2.2.5 Interpretation

In this experiment we observed a way of implementing MRSW registers. We saw that, at least in the processor we used, the algorithm seems to work correctly.

It would have been great, though, to also have test cases that demonstrate that this particular register satisfies property 3 which is the one that makes it different from the Safe version.

## 2.3 Wait-Free Snapshots

### 2.3.1 Particular Case

In this experiment we are dealing with the problem of creating an algorithm to create a Snapshot of a set of Register in such a way that the algorithm guarantees a Wait-Free operation.

### 2.3.2 Solution

The purpose of a Wait-Free snapshot is to overcome the problems of the Simple snapshot presented before. Let us remember that such snapshot executes sucesive `collect()` operations. Once it achieves a *clean double collect*, it returns the snapshot. Otherwise, it keeps trying.

One of the ideas behind the Wait-Free Snapshot is that when a double collect fails, it is because a update interfered. That means that the updater could take a snapshot right before its update and other threads could use it as their snapshot too.

### 2.3.3 Experiment Description

In this experiment, the test looks a bit different. Let us explain how it is different. In this case we have two experiments:

1. `testSequential`. We spawn a new thread that updates the register with a value of *FIRST* and right after that, it scans the value of the register. The expected result is that this read retrieves the value of *FIRST*
2. `testParallel`. We spawn *THREADS* number of threads (in our case it is 2). Each of them will first update its register with a value of *FIRST* and then with a value of *SECOND*. Each thread then updates a position in a matrix of size *THREADS* $\times$ *THREADS* with the result of doing a *scan()* operation. At the end we compare consecutive rows in the matrix. The comparison is done entry by entry and we should see that for some entries the first entry is greater and for some others it is smaller than the second. What we must see is that some times the first entry is that all entries are equal and we allow the first to sometimes be either greater or smaller than the second one.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

### 2.3.4 Sample Results

For this test, we saw that in every try, both test cases passed.

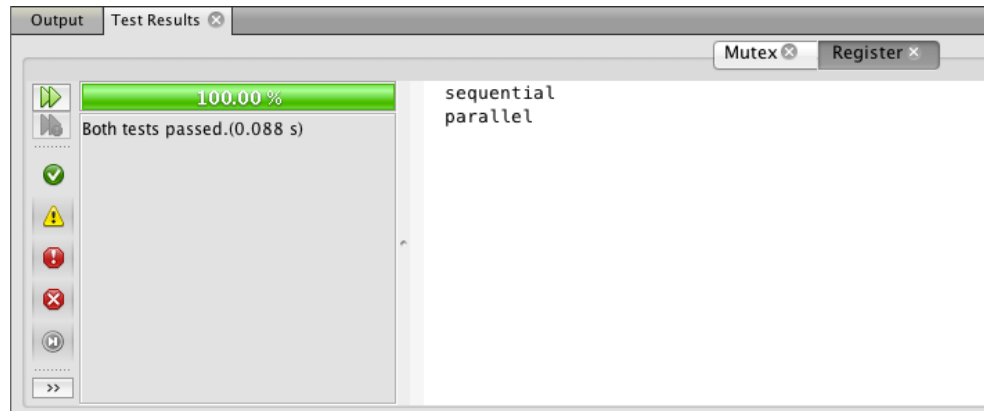


Figure 2.8: Successful execution of the tests for Wait-Free Snapshot

### 2.3.5 Interpretation

In this experiment we were able to observe how a Wait-Free Snapshot can be constructed.



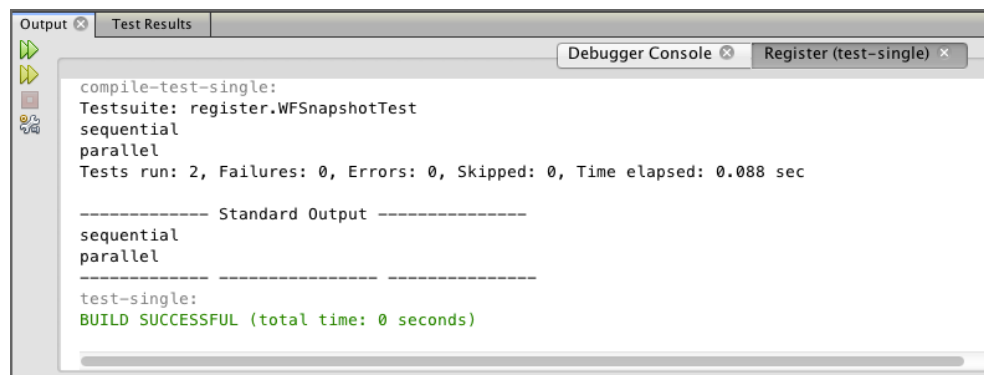


Figure 2.9: Successful execution of the tests for Wait-Free Snapshot

# Chapter 3

## Spin Locks and Contention

### 3.1 Problem Definition

The concept of Spin Lock comes from the following problem: *In the case where a thread tries to acquire a lock on a resource and it fails, what should be done?*

There are at least two ways to react to this problem:

1. Keep trying getting the lock. In this case the lock is called a Spin Lock. The thread is then said to be in a busy wait.
2. Suspend the execution of the current thread and let the others do some work. This action is called *blocking*

In this chapter we will study different ways in which spin locks can be implemented.

## 3.2 Test-And-Set Locks

### 3.2.1 Particular Case

The particular case we are trying to solve is that of mutual exclusion with two participants. The problem with previous approaches for solving this problem (e.g. Peterson Algorithm) is that in the real world the assumption of having a “sequential consistent memory” and a “program order guarantee among reads-writes by a given thread” are not true.

There are at least two reasons for that. The first one is that compilers, in order to improve performance of the program, can in some cases reorder certain operations. As the book clearly mentions, most programming languages preserve execution order that affect individual variables, but not across variables. This means that reads and/or writes of different variables can end up in a different order than the one appearing in the source code once the program is compiled or executed. This can clearly affect our Peterson Algorithm.

The second reason that has to do with the contradiction of sequential consistent memory is due to the hardware design which in most multiprocessor systems allow writes to be written to store buffers. These buffers are written to memory only when needed. Hence, race conditions arise, where one variable might be read before another because the latter was in a store buffer.

The generic way to solve this problem is by using instructions called a *memory barrier* or *memory fence*. These instructions are able to prevent reordering operations due to write buffering. The idea is that these instructions can be called in specific parts of the code to guarantee our proposed order of execution.

The problem in this section is *how can we implement an actually working mutual exclusion algorithm with consensus number of two using memory barrier?*

### 3.2.2 Solution

One solution for this problem is using a synchronization instruction that includes memory barriers. For example, we can use the *testAndSet()* operation which has a consensus number of two and which atomically stores *true* in a variable and returns the previously stored value.

To design an algorithm using this instruction the idea is the following. A value of false is used to indicate that a lock is free. Then, we store a value of true to indicate that we are using the lock. Once we are done, we can set the value back to false to allow other threads take the lock.

In the java code, the *testAndSet()* is done with the *getAndSet()* method. And setting the value to false is achieved with the method *set()*.

### 3.2.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 8 threads
3. Each thread has to increment the counter by one 1024 times
4. At the end, the counter must hold a value of  $8 * 1024$

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

### 3.2.4 Sample Results

For this test, we saw that in every try, it always passed.

### 3.2.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of two threads (consensus number was two) to access a shared resource.

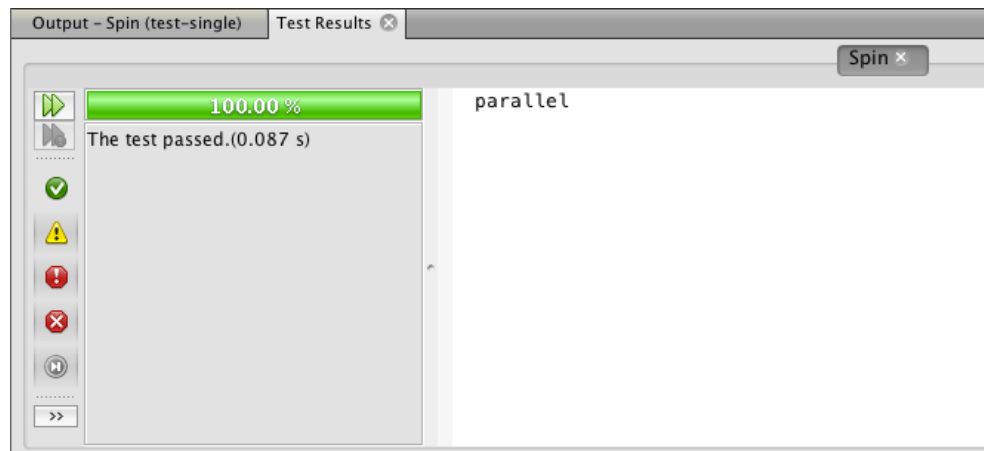


Figure 3.1: Successful execution of the TAS Lock test

```
compile-test-single:
Testsuite: spin.TASLockTest
parallel
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.087 sec

----- Standard Output -----
parallel
-----
test-single:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.2: Successful execution of the tests for TAS Lock

## 3.3 Hierarchical Backoff Lock

### 3.3.1 Particular Case

The specific case we are trying to solve here is that of mutual exclusion with long waiting times on architectures that make use of hierarchical caching. Let us describe more deeply the characteristics of this scenario.

Firstly, it has already been discussed the difference between *spinning* and *backing-off*. The first case consists on doing an active wait in the case where a *lock()* operation is not successful. On the other hand, a *backoff* means that the thread who failed to acquire the lock refrains from retrying for some duration giving competing threads a chance to finish. As mentioned in the book, the idea is to avoid bus traffic caused by threads asking over and over again for the state of the lock.

The term *Hierarchical* comes from the fact that modern cache-coherent architectures organize processors in clusters. These clusters are organized in such a way that intra-cluster communication is faster than inter-cluster communication. So, the idea in this experiment is to find a lock implementation that takes advantage of these access times differences.

### 3.3.2 Solution

One of the solutions proposed in the book consists on adapting the *test-and-test-and-set* lock that was described in a previous chapter to exploit the characteristics of the cluster. The idea behind this implementation is that backoff times of local threads should be shorter than backoff times of remote threads.

### 3.3.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 32 threads
3. Each thread has to increment the counter by one 32 times
4. At the end, the counter must hold a value of  $32 * 32$

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

### 3.3.4 Sample Results

For this test, we saw that in every try, it always passed.

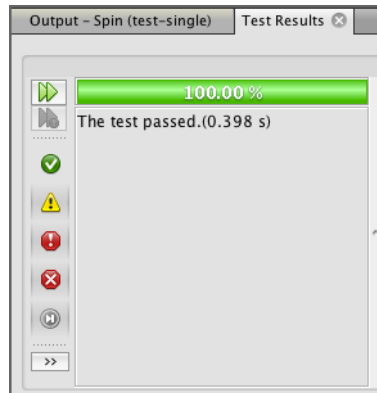


Figure 3.3: Successful execution of the tests for Hierarchical Back-off test

```
compile-test-single:
  Testsuite: spin.HBOLockTest
  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.398 sec

test-single:
  BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.4: Successful execution of the tests for Hierarchical Back-off test

### 3.3.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of multiple threads trying to access a shared resource.

One problem however, has to do with the fact that this locking protocol is not fair in the sense that local threads have more chances of acquiring the lock in the next attempt.

Another thing that we ought to note is that threads spin in the same memory address which causes invalidation of remote cached copies of this value. This problem will be solved yet with another protocol.



## 3.4 CLH Queue Lock

### 3.4.1 Particular Case

Using queue locks solve two problems that appear in simpler locking protocols. Firstly, by using a queue, cache-coherence traffic is reduced because each thread spin on a different memory location; and secondly, by using a queue, threads can know whether their turn has come by inspecting the status of their predecessor in the queue.

The particular case that we want to study in this experiment is *How can we implemente a space efficient Queue Lock?*

### 3.4.2 Solution

One solution for this problem is given by the CLHLock protocol. The algorithm that implement this protocol need two fields local to the thread: A pointer to the current node and a pointer to the previous node. The queue is constructed by having the pointer *previous* pointing to the previous node's *current* field. This field contains a boolean value which is false when the thread releases the lock. When this is the case, the next thread is allowed to acquire the lock.

### 3.4.3 Experiment Description

To demonstrate that this implementation works, the test that was provided does the following:

1. Initiate a shared counter with a value of 0
2. Start 8 threads
3. Each thread has to increment the counter by one 1024 times
4. At the end, the counter must hold a value of  $8 * 1024$

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB

- L3 Cache: 3 MB
- System Memory: 16 GB

### 3.4.4 Sample Results

For this test, we saw that in every try, it always passed.

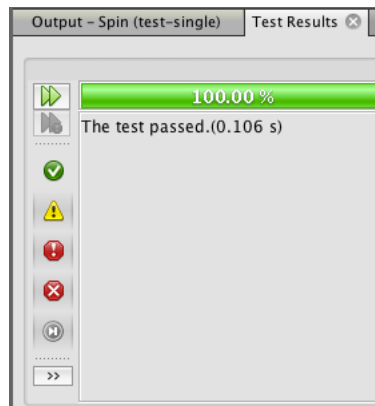


Figure 3.5: Successful execution of the tests for CLH Queue Lock test

```
compile-test-single:
  Testsuite: spin.CLHLockTest
  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.106 sec

test-single:
  BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.6: Successful execution of the tests for CLH Queue Lock test

### 3.4.5 Interpretation

It was shown that the algorithm seems to work and allows synchronization of Multiple threads trying to access a shared variable.

One important thing about this algorithm is that threads spin checking for different memory addresses which avoids invalidation of cached copies. Another important advantage is that it requires a limited number of memory and also provides first-in-first-out fairness.

The case where this algorithm is not good is when, in a NUMA architecture, the *previous* pointer points to a remote location. In that case, the performance of the algorithm degrades.

## Chapter 4

# Monitors and Blocking Synchronization

## 4.1 Count Down Latch

### 4.1.1 Particular Case

The purpose of this experiment to our eyes is to demonstrate how monitor locks and conditions are used. Let us remember both concepts before going further.

The concept of monitor consists on encapsulating three components: data, methods and synchronization mechanism. This allows to have our datastructures (or classes) take care of synchronization instead of overwhelming the user of the API with this task.

Now, the concept of *conditions* comes into play because with them we can signal threads about an event. For example, instead of having threads spinning waiting for a value to become false, we can instead signal threads and let them know that they can now re-check for their locking condition.

### 4.1.2 Solution

The java interface for *Lock* suggests two methods to implement the protocol drafter above. First, there is a method *await()* which basically asks the thread wait till it is signaled about an event. The accompanying method *signalAll* achieves this latter task.

### 4.1.3 Experiment Description

This experiment is a bit different from others showed before. The idea in this experiment is to show how the methods *await()* and *signalAll()* are combined in a working program.

So, the idea of the test is the following. Initially we have 8 threads. We start them as usual but we do not allow them to proceed normally. Instead, we will ask them to wait for a signal. Internally, what we do is decrement a shared variable. When this variable becomes 0, the signal is triggered. Initially this signal is 1, meaning that we only need to decrement the variable once.

After that, each thread will started and wait for the next signal. The variable that controls this other signal is initially set to 8. Each thread will be in charge of decrementing this variable by one. Once it becomes 0, the second signal will be triggered announcing that our test must finish.

These are the details of the system we used to run the experiments:

- Processor: Intel Core i5 @2.5 GHz. 2 Cores.
- L2 Cache per Core: 256 KB
- L3 Cache: 3 MB
- System Memory: 16 GB

#### 4.1.4 Sample Results

For this test, we saw that in every try, it always passed.

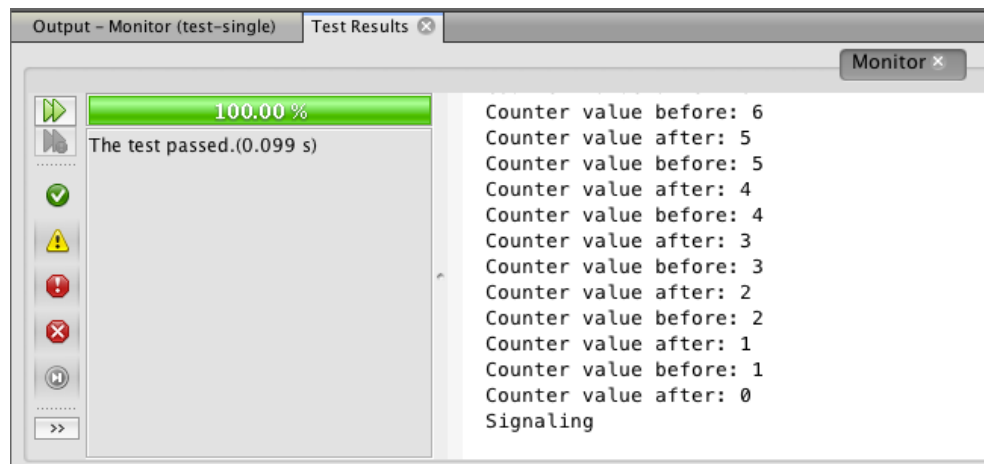


Figure 4.1: Successful execution of the tests for Count Down latch

#### 4.1.5 Interpretation

This experiment showed us the way in which Monitors are used. We saw that we did not need to continuously poll for a flag to acquire a lock. Instead, the monitor sent the threads a signal to indicate them that a particular condition has been met. In our particular test, these signals indicated: (1) that the threads can start; and (2) that the threads must stop.

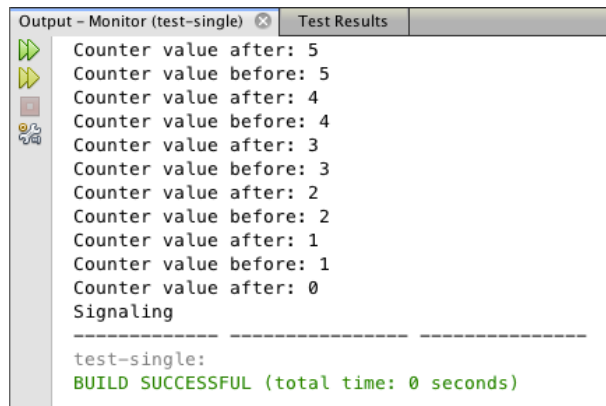


Figure 4.2: Successful execution of the tests for Count Down latch

## Chapter 5

# Linked Lists: The Role of Locking