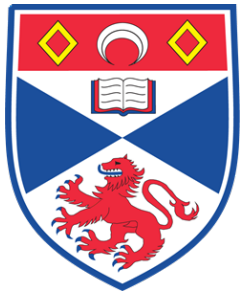
University of St Andrews

CS4204 Coursework 1

Concurrent Data Structure

Author:
150008022

March 1, 2019



Goal

The goal of this practical was to implement and compare a locking and lock free version of a concurrent account data structure.

Part I
Language and Structure

To allow for thorough analysis, the C language was chosen for this practical. This would allow both data structure implementations to use fairly low level operations whilst maintaining a reasonable level of readability and ease of development. Since the gcc compiler can also provide the resulting assembly code, this could also be analysed when comparing lock free and locking implementations. Comparing the effects of compiler optimisation on the locking code could also be considered.

The interface for the account is provided in *account.h*. Since C doesn't support classes, the methods described in the practical specification were altered to also take an account pointer as an argument. Create and destroy methods were also provided to allocate and free any memory used by the account struct.

Part II
Locking

The locking account struct involved two fields, the account balance and the account lock. This lock would have to be obtained in order for a method to make changes to the balance. The account creation method would initialise the mutex with the default attributes (PTHREAD_PRIO_INHERIT and PTHREAD_RECURSVIE_DISABLE). This provides blocking behaviour, where the thread will wait until it's able to obtain the lock. The mutex initialisation is included in the creation method to avoid multiple threads trying to initialise the lock later when it becomes needed.

Each method for interacting with the account struct first obtains the lock.

This is necessary to ensure that the balance variable will not be written to while reading, or written to by multiple threads simultaneously. The implementation is analagous to using the Synchronised keyword on each method in Java as essentially only one thread can operate on the object at any one time. Initially, a lock was not required for reading as it was assumed that the read would be atomic in the sense that a write could not happen simulatenously and return the half-written value at the relevant memory address. However, further reading showed that this behaviour was undefined, since this kind of atomic read for word-sized variables (i.e. int) was completely dependent on the compiler and architecture of the system [1].

For the destruction method, it is assumed that this would be called once all threads are finished operating on it. Realistically, there should be checks for success after calling the *pthread_mutex_destroy* method as destroying a mutex that is currently held by another thread can produce undefined behaviour [2].

The locking mechanism has the potential to suffer from the stampede issue discussed in lecture, and so another implementation was provided that used expoential backoff to try and mitigate this issue. This was done by adding a function for obtaining the lock, which would initially wait a given time before trying again, and double this delay after each failure. A maximum was set to avoid ridiculous wait times which when reached reset the delay back to the initial value.

Since the lock is obtained and released within each method, it is not possible for a thread to hold multiple locks, or wait on another lock while holding one. Because of this, deadlock is not possible.

The lock implementation can be compiled by running *make locking* to produce the executable, and the backoff implementation by running *make locking_backoff*.

Part III
Lock Free

The lock free account structure remained the same, but without the mutex. Since mutexes are surprisingly large on linux systems (24 bytes on 32-bit machines, and 40 bytes on 64-bit machines [3], this already had advantages

in terms of memory usage if a large number of the CAccount structs were to be in use.

The builtin *atomic_compare_exchange_weak* builtin from *stdatomic.h* was used to provide lock free concurrent access to the balance. A simple while loop would check if the value in expected matches the desired value, and try again if not. Since the atomic function would write the actual value to the expected variable if the values did not match, the loop could try continuously.

Possible issues with this implementation is the potential waste of resources caused by the constant spinning.

The lock free implementation can be compiled by running *make lockfree*.

Part IV
Tests

All experiments were run on a machine with the following specs:

Model Name	Intel(R) Core(TM) i7-8750H CPU
Clock Speed	2.20GHz
Architecture	x86_64
CPU(s)	12
Thread(s) per Core	2
Socket(s)	1
Core(s) per socket	6
OS	Ubuntu 18.04

1 Correctness

To test the correctness of the different implementations, each of them were run with an increasing number of threads. Each thread knew its index in the pthread_t array, and so even indices would withdraw from the account and odd would deposit. All threads would repeat this action 1000 times before terminating.

Listing 1: Withdraw using Atomics

```
void deposit(CAccount* account, int amount) {
    int expected, desired;
    int* pBalance = &(account->balance);
    expected = *pBalance;
    do {
        desired = expected + amount;
        atomic_compare_exchange_weak(pBalance, &expected, desired);
    } while (expected != desired);
}
```

The locking implementations showed no errors, and the end values were as expected. However, the lockfree implementation had issues, as the end value of the account was very inconsistent.

The erraneous lockfree implementation of deposit using the `atomic_compare_exchange_weak` function is shown in code listing 1:

Out of curiosity, the same method was written to use the deprecated `__sync_bool_compare_and_swap` builtin, which is the version used in the submission. The compiler is meant to fall back to __sync_ when 'neccessary'. Comparing the assembly code generated by both implementations, some differences were observed. Both used the `lockcmpxchgl` instructions. The `cmpxchgl` instruction is responsible for the compare and swap. The documentation mentions that for multiple processor systems, the `lock` prefix must be used to ensure that the compare and exchange is atomic [4].

2 Performance

To test performance, two aspects were considered:

- Wait time: how long does a requesting thread spend waiting to carry out an operation.
- Number of retries: how many iterations of attempting to access the resource before success.

These metrics were compared as the number of competeting threads were increased.

Increasing the number of threads far past the number of threads that can be processed simultaneously would increase the effect of the scheduler on the results. Therefore equation 1 was used to calculate the maximum number of simulateonously processable threads. For the architecture described earlier, $nthreads_{max} = 12$. Up to 24 threads were used in the experiment out of curiosity, though the calculated value would be considered during analysis.

$$nthreads_{max} = threads\ per\ core \times cores\ per\ socket \times number\ of\ sockets \quad (1)$$

Since the backoff lock could only use nanoseconds as the smallest unit, it was not possible to use values for maximum and initial delay that would be able to compete with the other lock implementations.

2.1 Setup

In order to capture the data needed, the functions would have to be slightly changed to capture the behaviour needed, and the process of recording data would have to be as unintrusive as possible.

This was achieved by passing an `Attempt` struct to each of the `CAccount` methods so that the number of attempts could be recorded. This would only add a few extra clock cycles after each attempt to gain the lock.

The CPU clock time before and after the function call was used to obtain the time taken for the function to execute, and provide a rough value for the wait time.

Before each experiement, memory was allocated for all the `Attempt` structs required for each thread, and freed afterwards. The max, min, and average values would be obtained for each thread, and printed to the console. To store the results, stdin was redirected to a csv file.

2.2 Analysis

To analyse and create plots of the data, a python script was used. The independant variable in each graph would be the number of threads, and this was plotted against the average, max, and minimum number of retries and wait times.

A scatter plot was created of each dependant variable against the independant variable. Plots merging the results from the different locking mechanisms were also created. In order to improve the visualisations, the wait

times were converted from seconds to nano seconds, as the former appeared as a flat line when plotted.

Once plotted, it was realised that the minimum wait time and number of tries was constant for all threads, and so this was omitted from the results.

2.3 Results

In each plot, each point corresponds to the measurement made by an individual thread when using a specific locking mechanism, i.e. when using two threads, there will be two data points for each locking mechanism at $x = 2$.

Figure 1 shows that there is a large performance difference between each mechanism. The backoff implementation had the largest variance as the number of threads increased, and the lockfree implementation had the best performance overall. Locking showed a linear growth in wait time, whilst lockfree showed logarithmic growth.

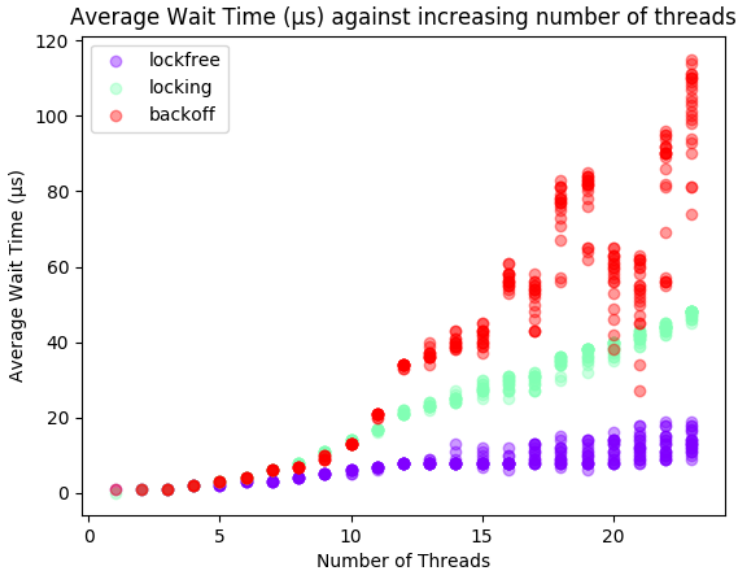


Figure 1: Average wait time of all locking mechanisms

Since the larger unit of delay used in the backoff implementation affected the scale of the plot in Figure 1, Figure 2 was also made. By only considering up to $x = 12$, we can see that the lockfree mechanism produces the lowest

wait time once more than 4 threads are involved. The effect of the scheduler can also be seen once $x > 12$, as the variance of the average wait times for both locking mechanisms increases.

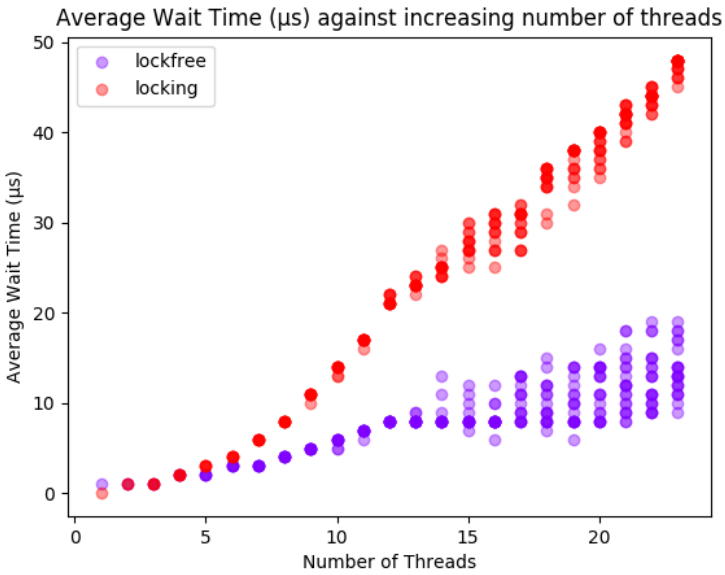


Figure 2: Average wait time of locking and lockfree locking mechanisms

Whilst the lockfree mechanism appears to reduce the time spent waiting to access a resource, the mutex has the advantage of allowing a thread to sleep, and allowing other threads to continue execution whilst it does so. Since the lockfree implenentation requires constant polling of the resource, it is likely to be less energy effecient, and more demanding on the CPU. Though the backoff implementation would reduce these negatives, it would suffer the trade-off of likely increasing average wait times.

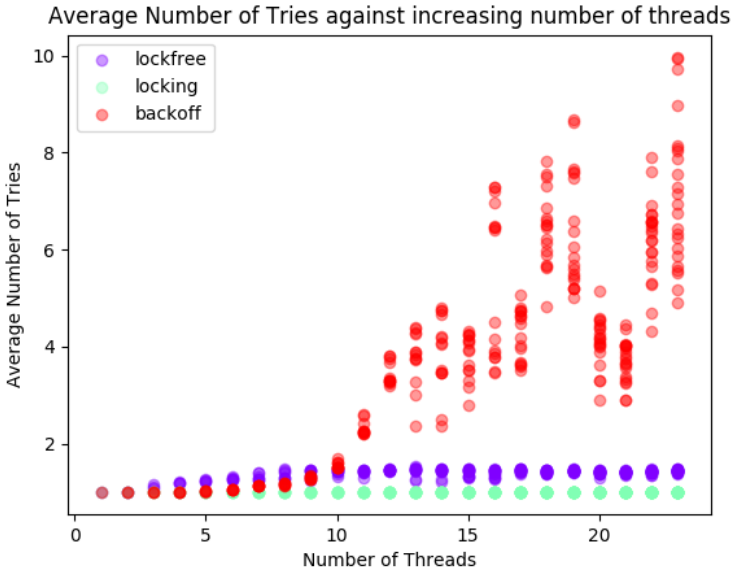


Figure 3: Average number of tries of all locking mechanisms

Conclusion

References

[1] Alexandar Sandler. Do you need a mutex to protect an int? `http://www.alexonlinux.com/do-you-need-mutex-to-protect-int`, 2008. Accessed: 2019-02-20.

[2] IEEE Std. pthread_mutex_destroy. `http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_mutex_destroy.html`, 2004. Accessed: 2019-02-21.

[3] dpw. skinny-mutex. `https://github.com/dpw/skinny-mutex`. Accessed: 2019-02-21.

[4] Intel. Intel® 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4. `https://software.intel.com/en-us/download/`

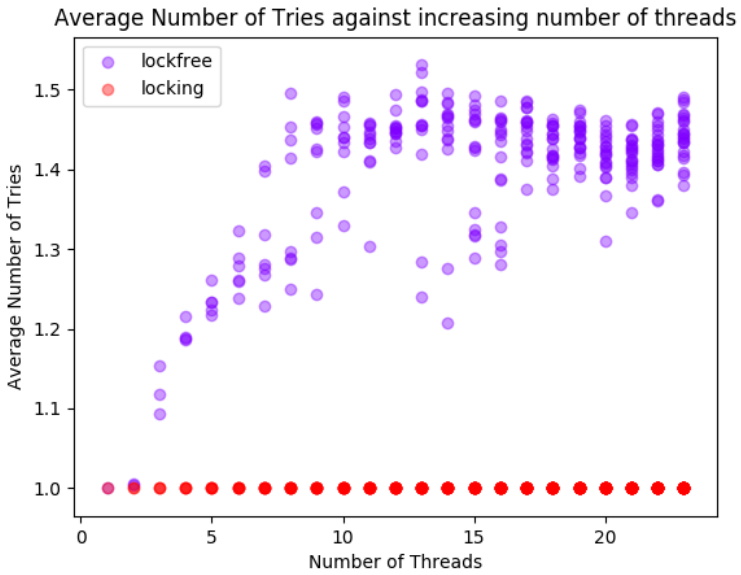


Figure 4: Average wait time of locking and lockfree locking mechanisms

intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3
Accessed: 2019-02-22.