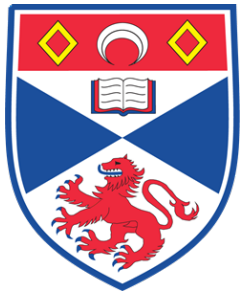
University of St Andrews

CS4204 Coursework 1

Concurrent Data Structure

Author:
150008022

February 24, 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. A create and destroy method 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. [`http://www.alexonlinux.com/do-you-need-mutex-to-protect-int`]

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 [`http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_mutex_destroy.html`].

The lock implementation can be compiled by running *make locking* to produce the executable. [1]

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 `https://github.com/dpw/skinny-mutex`), this already had advantages in terms of memory usage if a large number of the CAccount structs were to be in use.

Compare and swap was used to implement the lock-free concurrent account. Specifically the *__sync_bool_compare_and_swap*

Conclusion

[1] Facebook. create-react-app.