Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the heart of South Leinster

STM Library Project

Research Manual

OCTOBER 2021

Institute of Technology Carlow

Author: Evin Darling - C00144257

Supervisor: Joseph Kehoe

Course: BSc (Honours) in Software Development

Due: 05/11/2021

Module: Project

Table of Contents

1 Introduction

2 Introduction to Concurrency

The concept of concurrency, its problems, and some of its solutions, should be introduced before discussing STM.

2.1 What is Concurrency?

“A system is said to be concurrent if it can support two or more actions in progress at the same time.” (Breshears, 2009, p. 3). Concurrency is not the same as parallelism, although parallelism is a subset of concurrency. The key difference is that concurrency only requires two or more actions to be in progress at the same time, whereas parallelism requires two or more actions to occur at the same time. For example, a single-core computer can give the impression that it is performing multiple actions at once by frequently interleaving execution time for different processes/threads. In contrast, a computer with multiple processor cores can literally execute two separate actions at the same time.

2.2 Concurrent Programming

Concurrency can be applied to many problems. In this case, concurrent programming will be discussed. This typically involves writing programs that implement multiple threads of execution. Threads of a process can be executed concurrently and share access to the memory of their parent process. Concurrent programs promise performance gains on multicore processor systems. An idealistic expectation of performance gained for a simple application might be that running on a two-core system, you could expect the application to run in half the time (Breshears, 2009, p. 4).

2.3 Race Conditions

A race condition exists when multiple processes or threads have access to shared memory. The reading and writing of memory can cause unexpected issues to arise in multithreaded applications that do not implement a means of synchronising access to the shared data (Netzer and Miller, 1992). Consider a program with an integer variable `count` and two threads running a simple `count++` operation. The expected output value for `count` is 2. However, on register-based architectures count++ will be broken down into three steps: load, add, store. Interleaving of these atomic steps (Figure 1) can cause the output value to be 1. Having code output an incorrect value is obviously problematic and requires a solution.

Process	Instruction	count	Reg(1)	Reg(2)
(Initially)		0	-	-
P1	`LOAD Reg, count`	0	0	-
P2	`LOAD Reg, count`	0	0	0
P1	`ADD Reg, 1`	0	1	0
P2	`ADD Reg, 1`	0	1	1
P1	`STORE Reg, count`	1	1	1
P2	`STORE Reg, count`	1	1	1

Figure 1 Interleaving steps with incorrect output. Adapted from (Ben-Ari, 1990).

2.4 Critical Section

The previous section introduces an idea that is important in the development of concurrent programs. This idea is the critical section problem. In concurrent programs, there may exist sections of code that must only be allowed to be executed by one thread (or some maximum number) at a time to protect against, for example, race conditions. These are known as critical sections. The critical section problem is to design a system where each thread must request permission to enter a critical section (Silberschatz et al., 2013). If we consider `count++`, or more specifically `LOAD`, `ADD`, and `STORE`, in the previous example, to be the critical section of that program, ensuring that only one thread can execute it ensures that `count` will have its newly incremented value stored before another thread is allowed to access it.

2.5 Mutual Exclusion Locks

Mutual exclusion locks, or mutex locks, are a popular method of synchronising access to critical sections of code, likely because they are intuitive and simple (Fraser, 2004). They attempt to solve the critical section problem by enabling threads to lock sections of code. The first thread to reach a critical section will acquire the lock and only release it when it is finished. Other threads that reach the critical section will not be able to proceed until the lock has been released. Locking the `count++` example from previous sections would look something like Figure 2.

```
task():
   lock.lock()     // enter critical section by acquiring the lock
   count++         // execute critical section
   lock.unlock()   // exit critical section by releasing the lock
```

Figure 2 An example of a mutual exclusion lock.

2.6 Problems With Mutex Locks

Locks are not a universal solution to writing concurrent, multithreaded programs. They come with performance issues as they do “not scale well with large numbers of locks and many concurrent threads of execution.” (Fraser, 2004).

A low-priority thread might acquire a lock just before a high-priority thread arrives, locking out the more important thread until it is finished. This is called priority inversion (Herlihy et al., 2020).

Convoying
Deadlock/Livelock

3 Software Transactional Memory

Software transactional memory was introduced by Shavit and Touitou (1997) as an additional synchronisation option that they considered to be more flexible than hardware solutions at the time. Transactions provide an alternative approach to locks, mutexes, semaphores, etc. Usage of STM is simple for the programmer: blocks of code that need to be executed atomically can be converted into transactions.

STM is useful for managing shared-memory data structures where the level of scalability required cannot be achieved with locking mechanisms. The performance of an STM library is an important factor that must be measured during development. (Harris et al. 2010).

3.1 Transactions

A transaction is composed of a sequence of local and shared memory machine instructions. These instructions can be read-transactional, i.e. reads the value of a shared location into a local register, or write-transactional, i.e. stores the value of a local register into a shared location. Transactions are atomic – a transaction will either complete successfully or fail, discarding its changes. (Shavit and Touitou, 1997). A successful transaction commits all instructions in the transaction at once whereas a failed transaction will roll back its changes.

3.2 T-Objects

Transactional objects (“t-objects”)

4 STM Implementation

- Technology options
- C, C++

5 Similar Libraries

6 Portability/Platform

- Linux / Mac / Windows
- Writing cross-platform C/C++ code

7 Benchmarking

8 Comparisons

8 Bibliography

Ben-Ari, M. (1990). *Principles of concurrent and distributed programming*. New York: Prentice Hall.

Breshears, C. (2009). *The art of concurrency*. Sebastopol, CA: O'Reilly.

Fraser, K. (2004). *Practical lock-freedom* [online]. University of Cambridge, Computer Laboratory. Available from: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.html [accessed 2 November 2021].

Harris, T., Larus, J.R. and Rajwar, R. (2010). *Transactional memory*. 2. ed. San Rafael, Calif.: Morgan & Claypool.

Herlihy, M., Luchangco, V., Shavit, N. and Spear, M. (2020). *The art of multiprocessor programming*. Second. Philadelphia: Elsevier, Inc.

Netzer, R.H.B. and Miller, B.P. (1992). What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), pp.74–88.

Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10(2).

Silberschatz, A., Galvin, P.B. and Gagne, G. (2013). *Operating system concepts*. Ninth edition. Hoboken, NJ: Wiley.