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

Institute of Technology Carlow **0**

Table of Contents **1**

1 Introduction **2**

2 Software Transactional Memory **3**

2.1 Concurrency 3

2.1.1 What is Concurrency? 3

2.1.2 Concurrent Programming 3

2.1.3 Race Conditions 3

2.1.4 Critical Section 4

2.1.5 Mutual Exclusion 4

2.1.6 Deadlock and Livelock 5

2.2 Software Transactional Memory 5

3 STM Implementation **5**

4 Similar Libraries **5**

5 Portability/Platform **5**

6 Benchmarking **6**

7 Comparisons **6**

8 Bibliography **7**

1 Introduction

2 Software Transactional Memory

2.1 Concurrency

2.1.1 What is Concurrency?

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

“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.

2.1.2 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).

2.1.3 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).

2.1.4 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 whereby 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.

2.1.5 Mutual Exclusion

Mutual exclusion is a common method of synchronising access to critical sections of code and, therefore, a solution to the critical section problem. Mutual exclusion is often implemented using a locking mechanism. 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.

Unfortunately, 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).

```
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.

2.1.6 Deadlock and Livelock

2.2 Software Transactional Memory

3 STM Implementation

- Technology options
- C, C++

4 Similar Libraries

5 Portability/Platform

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

6 Benchmarking

7 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].

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.

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