Cost Evaluation of Synchronization Algorithms for Multicore Architectures

Masoud Hemmatpour
Politecnico di Torino, Italy

Renato Ferrero
Politecnico di Torino, Italy

Filippo Gandino
Politecnico di Torino, Italy

Bartolomeo Montrucchio
Politecnico di Torino, Italy

Maurizio Rebaudengo
Politecnico di Torino, Italy

INTRODUCTION

One of the major issues in modern computer architecture is multicore design. Programmers have been urged to design innovative algorithms by exploiting multicore facilities. Synchronization, i.e., the technique adopted for coordinating threads or processes to have appropriate execution order, is one of the main issues in programming on a multicore processor. In the literature, many synchronization techniques based on hardware and software have been proposed (Petrović, 2014; Yoo, 2013; McKenney, 1998). Modern computers provide special hardware instructions that allow to test and modify the content of a word atomically (e.g., the *cmpxchg* instruction of Intel) which can be used for synchronization of threads (Valois, 1995; Gao, 2007). Software techniques can synchronize threads without any dependency on hardware instructions (McKenney, 1998; Mellor-Crummey, 1991). One important aspect of a synchronization algorithm is its performance, which is evaluated in terms of overhead. In this study, the term cost is used to address an overhead of synchronization algorithm. The state-of-the-art approaches strive to increase the performance by reducing the cost of the synchronization. To the best of the authors knowledge a study to analyze the possible costs of synchronization mechanisms is absent. So, this chapter investigates the costs in the main steps of a synchronization mechanism. Moreover, since memory access is one of the most important costs in synchronization mechanisms, a discrete time Markov chain model of memory access cost is presented to evaluate the memory access overhead.

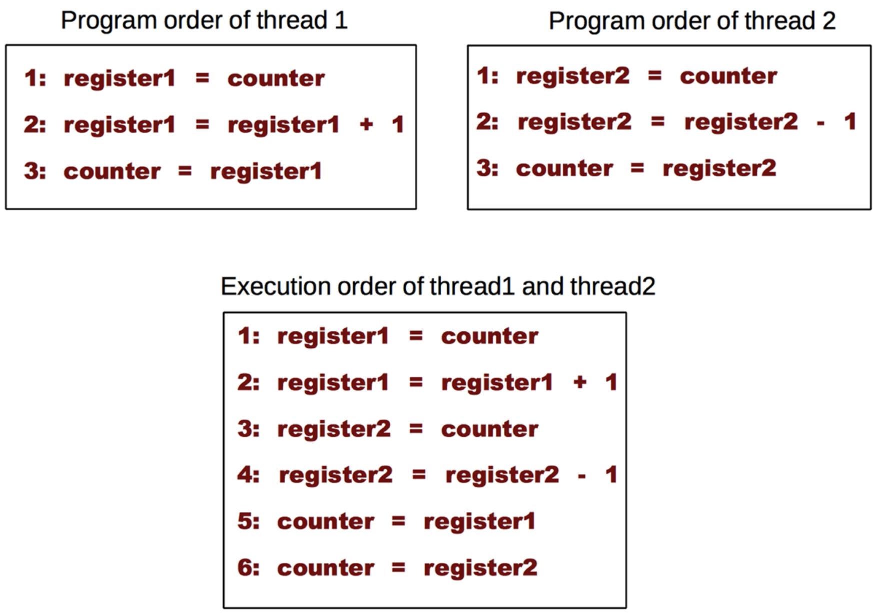
The remainder of this chapter is organized as follows. The primitives of the main synchronization algorithms are described in Section 2. Then, a theoretical evaluation of each cost and experimental results are presented in Section 3. Finally, some conclusions are described in Section 4.

BACKGROUND

When threads are working simultaneously on a shared object, their synchronization should be managed properly, otherwise the instructions of different threads interleave on the shared object in a wrong way. For example, Figure 1 shows the program order of two threads that are working on

DOI: 10.4018/978-1-5225-2255-3.ch346

Figure 1. Incorrect execution of the instructions order



Program order of thread 1

```
1:  register1  =  counter
2:  register1  =  register1  +  1
3:  counter  =  register1
```

Program order of thread 2

```
1:  register2  =  counter
2:  register2  =  register2  -  1
3:  counter  =  register2
```

Execution order of thread1 and thread2

```
1:  register1  =  counter
2:  register1  =  register1  +  1
3:  register2  =  counter
4:  register2  =  register2  -  1
5:  counter  =  register1
6:  counter  =  register2
```

the shared object counter (Silberschatz, 2006). Since one thread is incrementing the counter and another one is decrementing it, at the end, the counter is expected to have the initial value. However, as Figure 1 illustrates, there is a possible execution order of instructions that leads to an incorrect result.

Synchronization mechanisms are used to avoid the problematic interleaving instructions. The part of the code that accesses to the shared object is called critical section. The critical section should be protected by synchronization primitives to avoid concurrent access to the shared object:

- Ticket lock is a synchronization mechanism to guarantee fairness execution to all the threads. Ticket lock is a kind of spinlock that keeps checking to see if a lock is available to acquire the lock. Ticket locks are conceptually two bytes, one indicating the current head of the queue, and the other one indicating the current tail. The lock is acquired by atomically noting the tail and incrementing it by one (thus adding it to the queue and noting the position), then waiting until the head becomes equal to the initial value of the tail (Piggin, 2008).
- Filter lock is a synchronization algorithm that works for multiple threads. The filter lock creates N – 1 waiting lists, called levels. A thread must pass through all the levels before entering the critical section. As shown in Figure 2, at level 0 at most N threads are waiting. In level 1, at most N - 1 threads are waiting. The procedure continues till level N - 1, that corresponds to the critical section, in which only one thread is enabled to enter. Each level should satisfy two properties: first, among the threads that are trying to enter into a level, at least one thread succeeds. Second, at least one thread is blocked in the level (Herlihy, 2006).
- Readers -writer lock (rwlock) is intended a solution for solving the readers-writers problem, where a resource is shared among readers and writers. The readers-writers problem happens when a reader starts to read the first item of the shared resource, then stores it in a local variable.