Cost of mutual exclusion with spin locks on multi-core CPUs

SÁNDOR JUHÁSZ  ÁKOS DUDÁS  TAMÁS SCHRÁDI
Budapest University of Technology and Economics
Department of Automation and Applied Informatics
Magyar Tudósok krt. 2. QB207
HUNGARY
{juhasz.sandor, akos.dudas, schradi.tamas}@aut.bme.hu

Abstract: Multi-core CPUs promise great performance increase in applications and algorithms that can take advantage of them. Such algorithms, offering high level of parallelism, must protect data structures from concurrent modifications possibly violating data integrity. Synchronization mechanisms offer protection from such erroneous behavior by limiting data modifications to a single thread. In shared memory systems, such as multi-core CPUs, synchronization is cheap as threads can communicate with each other via locks placed in the main system memory. These locks, however, present as performance overhead not only by limiting the level of parallelism but also by changing the characteristics of the application. By investigating various implementations of mutual exclusion mechanisms with the use of spin locks an overview is presented of synchronization options in multi-core systems and the performance of several spin locks is evaluated.

Key–Words: mutual exclusion; spin locks; multi-core CPU; test-and-set

1 Introduction

Mutual exclusion is used to resolve possible concurrency issues when shared resources are accessed in multiprogrammed environments. Data structures that support parallel access protect the integrity of the data by assuring that no concurrent modification of shared data locations is allowed. In order to provide mutual exclusion the threads wanting to access protected data must synchronize with each other. This synchronization is carried out through locks. Locks are flags residing in the main system memory visible to all threads. Before entering a critical section, first the state of this flag is read. If the flag indicates that the lock is free it is set to busy state and the thread proceeds; otherwise it must wait and retry.

Checking the state of the lock and changing it to busy must be a single atomic step. Otherwise due to the race condition it would be possible for more than one thread to enter the critical section at the same time. The fastest and easiest way of assuring atomicity is through hardware support. All x86 architectures implement instructions that serve this purpose, such as the test-and-set and the compare-and-exchange operations. Building on these instructions relatively cheap locks can be constructed with a few assembly instructions [3].

Operating systems and runtime environments also provide various types of locks. Most of these locks differ from test-and-set types of locks by providing additional services, such as suspending threads while the lock is in a busy state. With test-and-set locks, the easiest way to wait for the lock being held is the so called spin-waiting approach, also called active waiting. The thread accessing the lock executes a cycle continuously checking the lock until it can successfully acquire it. Passive waiting, on the other hand, requires operating system support to suspend the thread when the lock cannot be acquired and resume it later when it has been released.

Active waiting is generally thought to be better in case of short critical sections, as the overhead of suspending the thread, executing the scheduler to select another thread for executing, then doing a context switch, can be a significantly bigger overhead than repeating a single test-and-set instruction. Additionally, since multi-core CPUs are cache coherent spinning can be performed in local memory where accessing the lock has very little cost because it resides in one of the CPU caches with short access latency.

Active waiting, however, is not always the best choice for locks. Suspending the thread waiting for entry into a critical section is beneficial for example when the critical section contains I/O or network operations, or similar high latency queries. Operating system locks are also favored in server environments and applications with a high number of active requests, such as web servers. In these algorithms suspending a thread does not diminish the performance of the system because while waiting for the lock the system
can process another request thus still occupying the system resources.

It is crucial that the locks only change the characteristics of the algorithm minimally and present negligible overhead. Systems designed for high parallelism have little contention for the locks [1]. Therefore acquiring the releasing a lock should be very cheap, as they are accessed a lot, but they only serve as safety measure.

In this paper we focus out attention to spin locks for short critical sections. Previous experiments have confirmed that spin locks can deliver much better overall performance in the scenario discussed in this paper.

Simple spin locks can be designed in many ways. Spinning can be done on the test-and-set instruction, on local memory without atomic operations, a delay can be inserted between the retries, or the requests can be queued to assure FCFS order. In this paper several spin lock alternatives are discussed taken from the literature and their behavior and performance is evaluated in current multi-core CPU systems.

The rest of the paper is organized as follows. Section 2 presents six spin locks taken from the literature evaluated in this paper. Section 3 analyses the number of retries needed to acquire to lock, the overall performance of the locks, and the cost of a single lock test. The conclusions are included in Section 4.

2 Related works

This paper examines six spin locks, published by Anderson [2], Graunke and Thakkar [4] and Mellor-Crummey and Scott [6]. These locks have been thoroughly analyzed in the referenced works in multiprocessor system. Our focus, however, are multi-core CPUs, which differ significantly from multiprocessor systems in terms of the memory system. The goal of this paper is to re-evaluate the locking schemes designed specifically for multiprocessor systems in multi-core systems.

The simplest of all locks is the test-and-set spin lock. A bit test-and-set instruction is executed in a loop until the return value indicates that the lock was acquired by this thread. Releasing the locks resets the bit. The test-and-set instruction is supported by all x86 compatible architectures, and is a relatively low cost instruction. The overhead of this lock is the need to access the lock bit in the lowest common level in the memory hierarchy (so that the modification is seen by all), but since in current multi-core CPUs the last level cache is shared by all cores, the instruction is likely to work only in the cache and will not access the main system memory.

If the cost of accessing the lock flag is larger, spinning can be performed in local memory on a copy of the lock flag. Specifically, this is carried out by spinning on the lock flag using a simple memory read and a compare operation. Since multi-core CPUs are cache coherent, this local memory will always be valid. The gain in this method, called test-and-test-and-set is using a much cheaper instruction than test-and-set while the lock is seen busy. But then, to acquire the lock, a second test-and-set instruction is required.

As spinning wastes CPU cycles and increases memory traffic (to continuously load a valid copy of the lock flag from some level of the memory subsystem), a random delay [4] can be inserted before retries. The delay is spent with active waiting, which means, that the scheduler is not involved, the thread does not give up its time slice, only works in local memory for a short time without performing any real work. Reducing the polling is expected to reduce memory contention, although the CPU still performs useless instructions. Randomizing the amount of delay is a good heuristic since the exact time to wait is not known.

Alternative to randomized delay, Anderson [2] suggested to apply an exponential backoff scheme in the delay, as the Ethernet protocol does. The purpose of the delay is the same, to minimize memory contention. And as exponential backoff works in networks, it can also work in resolving lock contention. The potential drawback of this scheme is the case the thread waiting for the lock doubles its waiting time right before the lock holder releases it. In such unfortunate situations a long and unnecessary delay is the result. This, of course, can be managed by applying a maximum to the delay over which limit it is not increased any more.

The last two types of lock are different from the four above as they guarantee FCFS service order meaning that the lock allows entry to the critical section in the same order the threads requested it the first time. Both the test-and-set-based and the delay-based spin locks have random behavior in the previous cases. The thread which notices the free lock first will get the lock. Randomized service behavior makes such a system harder to analyze but spares the administration costs of queuing the requests.

Explicit queue is in fact queuing in shared memory by Anderson (a different name is used to distinguish this from the next queuing solution). Arriving threads are queued in a fixed-size array: the length of the array is the number of threads, and the insertion position is determined by an atomically increased counter modulo the length of the array. When a thread releases the lock it enables the next item in the queue
by changing its flag. The waiting threads perform constant spinning on their flag in the array. The advantage is spinning in local memory and the use of cheap operations (only a single atomic increase of an integer is executed for each request). However, since the flags are in an array, the false sharing effect [5, 7] can increase memory demand.

The last lock is a simple and elegant solution by Mellor-Crummey and Scott [6], called the ticket lock. Instead of an explicit queue the arriving threads acquire a ticket (a number) from an atomically increased counter. When the lock is released, a different counter, the currently served number is incremented atomically. The threads spin on their ticket until it is equal to the currently served number. Requiring a smaller footprint (two integers) this solution in independent of the number of threads in terms of memory requirement.

3 Performance analysis

This section takes three aspects to test and evaluate the locks: the number of retries needed to acquire to lock, the overall performance of the algorithms with the locks, and finally the cost of a lock test.

3.1 Lock retries

The first behavioral element inspected of the locks is the average number of retries needed to acquire them. The reason for this number being relevant is the fact that the number of retries is equal (up to a constant factor) to the number of memory accesses. It must be underlined that not all memory accesses have the same cost, hence the number of retries cannot be a direct measure of performance, but is a relevant parameter. Examining this factor is crucial in understanding the way the locks behave.

All six locks were implemented in C++ using assembly instructions or compiler intrinsics. The following tests were executed on a state-of-the-art Intel Core i7-2600K CPU @ 3.4 GHz with three-level cache hierarchy having 4 physical cores and Hyper-Threading technology enabling 8 virtual cores to execute simultaneously.

The test case used to measure the performance of the locks features a simple data processing scenario where a set of data elements is processed by multiple threads. Each thread works independently of the others to process the data item, but then storing the processed entry requires access to a shared data store protected by the lock.
The lowest number of retries (see Figure 1) can be attributed to the two locks applying delay. This was indeed the goal with the delay inserted between retries, to minimize the number of unsuccessful memory accesses. The exponential backoff has a slightly lower number of retries than the randomized delay. This is naturally due to the increasing amount of delay with this scheme where in the randomized case the delay can be less then in previous retries.

The test-and-set solution, which applies absolutely no delay or any other means of reducing the number of failed retries, closely follows the previous two while using a much cheaper test operation in the test-and-test-and-set lock the number of retries nearly triples. This behavior can be explained by the test-and-test instruction requiring more CPU cycles than a simple memory read reducing the instruction throughput of the thread.

The FCFS locks, ticket lock and explicit queue have a much higher number of retries. In case of these locks the retries contain absolutely no atomic instructions. Such instructions are involved in preparing for the first lock test, but then lock fails and retries need no complex instructions.

### 3.2 Performance

**Ticket lock** having the absolute highest amount lock tests before successful entry into the critical section is the direct opposite of exponential backoff with 35 times more retries and memory accesses with 8 concurrent threads racing for the lock. Let us present whether this translates into real performance difference.

This time the execution time of the algorithms using the locks is measured. The execution time contains both the parallel work of the threads outside the critical sections and the exclusive access within the critical section. Both time spans determine the possible gain with parallelism, as according to Amdahl’s law; the question is whether the overhead of the locks or the changed behavior of the algorithms attributes to significant differences.

The difference in the lock retries between the test-and-set-based locks, with and without delay, and the FCFS locks, in fact, does not translate into significant performance difference (see Figure 2). The lines of the figures are hardly distinguishable, meaning there is no more than a 10% difference between the best (ticket lock) and the worst (random delay).

At high levels of use with 6-8 concurrent threads trying to access the locks a minor difference is observed between the performance of the locks. This means that the locks are in fact present as an overhead in the application and their cost can be minimized, which is good for performance.

### 3.3 Cost of testing the lock

In Section 3.1 the number of lock retries showed very big differences while the execution times in Section 3.2 only presented slight differences in performance. From this it follows that the difference between the locks is the cost of a single lock test. This cost (see Figure 3) is very different for all types of locks.

Please note that the figure does not include random delay and exponential backoff as due to the inserted delay it is not the cost of accessing and testing the lock is what is expensive resulting in lower retries. Thus comparing the cost of accessing the lock measured in execution time would falsely indicate high cost in these two cases.

The lowest cost is attributed to ticket lock and the highest to test-and-set. These results explain why 35 times more lock retries result in nearly the same performance. The difference is the cost of a lock test: a test-and-set instruction executed in a loop costs 35
times more CPU cycles than spinning on local memory using a single memory read and compare operation.

4 Conclusion

Locks are basic building locks of parallel algorithms and applications. There are many alternatives for implementing simple spin locks. The correct type of the lock to use is not trivial. Spinning on some sort of a flag wastes CPU cycles and can cause cache contention for accessing the same cache line often, while more sophisticated locks can increase the mere overhead of acquiring a lock for which there is no contention.

In this paper a detailed evaluation was presented for six types of locks. Most of the locks were developed for multi-processor systems where communication among the threads via the locks is more expensive than in current multi-core CPUs via the caches. The spin locks all feature basically the same performance characteristics: they scale really well with the number of threads and no significant difference can be observed in performance.

The difference between the locks lies in the cost of testing the lock. If the contention for a lock is high, spinning on the lock flag is carried out often. This spinning has considerably different costs for the different types of locks. Our conclusion is that spinning on local memory, thanks to the cache coherency in multi-core systems, is a much cheaper operation.

Access to global memory is much cheaper in multi-core systems, thus avoiding it (spinning in local memory) has no performance gain. The best example for this is test-and-set and test-and-test-and-set, where there is no difference is overall performance. Inserting a delay between the retries does not help either, although it does not worsen the performance.

The best option, though only by a narrow margin, is the ticket lock by Mellor-Crummey and Scott [6], which also has a nice feature guaranteeing first come first served ordering of the requests.

Acknowledgements: This project is supported by the New Széchenyi Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References:


