CS 1651: ADVANCED SYSTEMS SOFTWARE
FALL 2013

Project 1 – Synchronization

Introduction

Multicore processors are now a mainstay of commodity computing, from mobile devices to server class systems. This has required Operating Systems to now consider parallel architectures as the predominant substrate of execution, and has given rise to many new approaches and designs. Synchronization poses one of, if not the, largest challenges for these new environments. When multicore CPUs were being introduced poorly designed synchronization limited scalability due to the prevalence of course grained global locks that only allowed a single core to execute a given operation. One example of this can be seen in the process scheduler for Windows Vista that included a global lock in the system wide process dispatch routine and other issues that resulted in large performance degradation with more than 8 cores [1]. If you have followed Linux development then you have probably heard mention of the BKL (Big Kernel Lock) that prevented more than a single core from operating in the kernel at any given time. As a result the number of synchronization operations has exploded as OS designers move towards finer grained locking strategies.

In order to ensure that OS performance continues to scale as the number of cores increases synchronization approaches have taken the spotlight. In this project you will implement a small selection of synchronization primitives used on the x86. This project is broken into multiple parts, and builds on itself as you complete it. In this project you will implement:

- Memory barriers
- Atomic operations
- Barriers
- Spinlocks
- Reader-Writer Locks
- A lock free queue

The project framework can be downloaded from the course website, and assumes that you have a 64 bit Linux environment. The project evaluation will be conducted on a 64 bit version of Fedora 15, and it is your responsibility to ensure that your code functions correctly in that environment. Inside the unpacked project framework you will find a set of files that include a set of evaluation routines (driver.c) and the implementation files for the locking operations (locking.h and locking.c). For this project you will only modify locking.c and locking.h, you may not make any changes to driver.c.

Inside locking.c you will find a set of function declarations that you will need to implement. You are not allowed to change the function prototypes. You may implement additional functions if you so choose, but the functions provided are sufficient to complete the project in its entirety. The amount of code for this project will be relatively small (less than 300 lines of code), so each function implementation will be fairly concise. Some functions include an struct argument that is defined in locking.h. You may modify these structures however you choose. You will not have to manage the structures allocations and deallocations, as that will be handled inside the driver implementation. We recommend you implement this project from the start to the end in order, as there is a loose dependency on the primitives. Meaning, that later primitives can and will make use of earlier operations.

Synchronization primitives utilize the very low-level behavior of the CPU. Implementing them requires that a programmer very carefully manage a CPUs execution directly in order to make sure it does the correct thing. Accordingly the basic synchronization primitives are generally implemented directly in assembly language to ensure that the CPUs operation is correct. For this project you will implement a mixture of inline assembly and C code to achieve the correct behavior.

[1] http://betanews.com/2009/11/17/pdc-2009-scuttling-huge-chunks-of-vista-architecture-for-a-faster-windows-7/

Memory Barriers

The first operation you will implement is a simple memory barrier. These are used to ensure that the compiler does not reorder or omit certain memory operations as part of its optimizations. A memory barrier does not necessarily interact at all with the hardware, but instead is a signal to the compiler that memory operations that placed before the barrier must occur before any memory operations placed after the barrier. Without barriers the compiler is free to reorder operations in anyway it wants as long as the behavior is consistent according to its own behavioral model.

As an example of why this is useful consider a very basic locking strategy that uses an integer counter located in memory.

```
int global_lock = 0;

void do_something() {

    // make sure that lock is available (global_lock == 0)

    global_lock = 1;

    // do something

    global_lock = 0;

}
```

In this case a compiler is not aware that the operation on `global_lock` has any side effects outside of simply setting its value to 1. Therefore the compiler might decide to reorder the operations so that `global_lock` is set to 1 after the critical region `do something`. Or it might simply remove the operations on `global_lock` altogether because it is pointless from the point of the of the compiler (setting the value to 1, and then to 0 without reading it is the same as just leaving its value as 0.)

In order to prevent the compiler from making these mistakes you will need to implement `mem_barrier` which will tell the compiler that it cannot rely on memory values persisting across its invocations.

Atomic Operations

Next you will need to implement simple atomic operations. Modern multicore x86 processors implement a shared memory model with cache coherency that ensures that each core sees a consistent version of the same memory space as every other core. This means that when one core performs a single memory operation it is immediately visible to every other core in the system. This behavior provides the substrate by which all synchronization is performed on x86 platforms.

Atomic operations are the simplest form of synchronization and provide a building block for many of the higher level synchronization operations. Their necessity is due to the fact that x86 memory operations are very rarely "simple", in that they require multiple memory operations in order to complete. At the basic level a memory operation is either a load or a store that either reads or writes a value to a given memory address, as should be familiar to you if you have taken an architecture course (see RISC vs. CISC). The x86 exposes a complex ISA that is implemented as a RISC ISA under the covers. This means that a single x86 instruction is actually implemented as a set of micro-ops (such as load/store).

The upshot of this is that the cache coherency protocol of the x86 operates on micro-ops but not entire x86 instructions. Therefore if you have a complex instruction (such as an add) that is implemented via a sequency of micro-ops that first reads a memory address, modifies its value, and then stores it back to the same location, the hardware provides no guarantee that another core will not modify the value in the middle of add instruction's execution.

This means that by default, x86 operations are not **atomic**. To make an operation atomic, a programmer must explicitly indicate that desire to the CPU. When the CPU detects this it acquires a lock on the memory bus for the duration of a single instruction. Implementing this functionality requires that the programmer add a special “lock” prefix to the instruction they wish to make atomic.

For this part of the project you will need to implement atomic verisons of both add and substract.

Spinlocks

Spinlocks represent one of the most popular forms of locking, and are used as a general locking primitive to provide straight-forward mutual exclusion. Spinlocks get their name from the fact that historically they utilize a polling or busy-wait approach while waiting for a currently held lock to be released. Many current spinlock implementations try to take a more nuanced approach to waiting for a lock that does not impose the utilization penalties associated with polling. In this project we will only focus on a naive spinlock implementation that busy-waits (polls) while waiting for a lock to be released.

Each spinlock is initialized via `spinlock_init` which passes a pointer to a spinlock state structure that you will fill in. After a spinlock is intialized it maybe be acquired via `spinlock_lock` and released with `spinlock_unlock`. Only a single thread is allowed to acquire a spinlock at any given time, threads that try to acquire an already taken lock must wait until they are able to acquire the lock themselves.

Note that the there is a fairly complicated set of tasks that are required by the spinlock code. It must check to see if a lock is available and then acquire it if its free. This must all be done atomically, to ensure another thread does not acquire the lock between detecting that it is free (memory read) and marking it as acquired (memory write). To achieve this the x86 has implemented a special set of instructions that perform complex memory operations. For this project you will utilize the compare-and-swap instruction `cmpxchg`. You must implement the functionality using the x86 ISA, and are not permitted to use wrappers located in the C runtime.

Barriers

Barriers are a way by which multiple threads wait a specific location in the code until other threads in the system have reached the same location. This primitive allows a programmer to implement a single synchronization point that can be used to ensure that an application executes in lock step behavior. These operations are extremely popular in parallel distributed systems and a predominant feature in many MPI based parallel applications. While they are most often seen in distributed memory systems, they have their use in shared memory environments as well.

As an example of why you would use a barrier consider a scientific application that simulates a physical phenomena over time. The simulation is broken into a series of timesteps that each represent a given quanta of simulated time. The simulation code that executes is exactly the same for each timestep, and modifies a shared buffer that represents the current state of the simulated system at a given point in time. Now imagine what would happen if one core got out of sync with the rest and began simulating a different timestep than the rest of the cores. Because the simulation is using a single shared buffer to store the state data, the simulation would no longer be valid as the buffer no longer represented the correct state at a given point in time. To prevent this from happening, such an application needs to ensure that every core is operating on the same timestep. If one core finishes before the others it needs to wait for the others to finish instead of moving ahead to the next time step before the other cores. Barriers provide a way to ensure this behavior.

For this part of the project you will need to implement a simple barrier primitive. The barrier will be initialized with a certain count which will indicate the number of threads that the barrier will need to wait for. A thread will signal its arrival at the barrier by calling `barrier_wait`, which will be responsible for ensuring the thread does not continue execution until the appropriate number of threads (as specified by the init function) have arrived at the same location. Once all the threads have reached the barrier, they will be allowed to continue. The barrier should automatically “reset” itself: That is a barrier should be able to handle multiple iterations of waiting with only a single initialization.

For reference you may want to read the man page for `pthread_barrier_wait().`

Reader Writer Locks

Reader writer locks are a specialized form of spinlocks designed for a particular use case where reads happen much more often than writes. To see why this distinction is useful, we go back to the behavior of the x86’s cache coherency protocol. Cache coherency ensures that memory operations that modify a memory value are atomic and immdiately visible to every core on the system. However, this key here is that the coherency protocol only comes into play when memory is modified. If memory is simply being read then there is no reason why multiple cores cannot each access the value concurrently without any ramifications. Each core caches the value of the memory locally, and provides immediate access to it without notifying the other cores. For simple behaviors with a single writer thread modifying a single memory word this is all that is needed, as the cache coherency protocol will ensure that any modification is propagated to the rest of the cores which will then use the updated value for subsequent reads. However for anything beyond the basic case, a more complicated locking scheme is needed.

One example of a more complex case is a function that references a memory value repeatedly as part of its execution. The only requirement this function has is that the value does not change in the middle of the function. That is a memory reference at the beginning of the function should always be equal to a memory reference at the end of the function. If it is a read only variable then there is no problem, however what happens if the variable is occasionally modified? We need some way to ensure that the modification happens while the function that reads the value is not running anywhere in the system. It is cases such as these that Reader Writer locks are designed for.

Reader Writer locks provide a way for multiple readers to concurrently access a memory value with the assurance that it will not be modified. This is achieved by requiring each reader to acquire a read lock that prevents modifications to occur while still allowing other readers to access the value. When a modification is required code must acquire a write lock that behaves like a normal spin lock and ensures full mutual exclusion to the writer. This creates a scenario where multiple readers may concurrently acquire read locks until a writer requests a write lock, at which point they must wait until the writer completes and releases the write lock. Conversely a writer must wait for all the current readers to release their locks before the write lock is acquired

For this part of the project you will need to implement reader writer locks. You will have to handle lock acquisitions and releases for both readers and writers. While there is some complexity in a reader writer lock's behavior when dealing with read lock requests while a writer is waiting, for this project we will again take the simple approach. Once a writer requests a write lock, any subsequent reader must wait for the writer to acquire and release the lock before they are allowed to acquire a read lock.

Lock Free Queue

Locking is a complicated business and is the largest source of complexity in modern systems. A good rule of thumb for when to use locking is to (1) Don't, unless you absolutely have to, and (2) lock at the finest granularity possible to avoid performance penalties and possible deadlock scenarios. When people are first getting started with concurrency, there is a common tendency to use locks frequently and in complex manners. 99% of the time this is a mistake. You should always try to minimize the amount of time you hold a lock, not just for performance but also to avoid complexity in the code. In particular you should avoid as much as possible situations where you call out of a function while holding a lock. Dealing with concurrency requires a certain mentality, and the first step to achieving that mentality is the realization that locking is a necessary evil that should be minimized whenever possible.

As a result of the perils introduced by locks, many people have sought alternatives to the traditional mutual exclusion based locking approaches. One such example is the use of lock-free data structures, in particular queues. A lock free data structure is a data structure that is capable of being operated on simultaneously by multiple threads.

For the last part of this project you will implement a lock free queue, as described in Implementing Lock Free Queues by John D. Valois (available on the course website). You will need to implement the algorithm described in section 3.4 ("A New Lock-Free Queue"). In addition to the implementation code please include a short answer to the question below.

The test function uses multiple enqueue threads and a single dequeue thread.

Would this algorithm work with multiple enqueue and multiple dequeue threads? Why or why not?

Conclusion

As you have seen, all of the synchronization operations you have implemented so far have relied on atomic features provided by the architecture. In the case of the x86 (and most other architectures) these features are implemented via hardware on a system wide level. That is an atomic operation requires locking the memory bus across the entire system, and relies entirely on the system wide cache coherency protocol that manages the caches of all the cores. As core counts continue to increase the this arrangement will quickly encounter fundamental scalability problems as cache coherency introduces more and more overhead. Indeed Intel is currently investigating CPU architectures that have partitioned cache coherency domains. While the primitives we explored in this project will most likely remain relavent for many years to come, new approaches to synchronization are being actively explored and developed. For example, transactional memory is an approach to providing synchronized behavior in a way that avoids explicit locking.

Submission

The project evaluation will be done via a short demo in which each student will be given the opportunity to demonstrate the functionality of their project and describe their implementations. However you must still submit your code over email before the due date, and the demos will use the version of the code that was submitted. Demos will be arranged via appointment and should not take more than 5-10 minutes. You will be required to show that the code is working, describe it implementation, and answer any questions. Demo appointments will be arranged 1-2 weeks before the due date.

To submit your code, please send a tarball containing your locking.h and locking.c files to jacklange@cs.pitt.edu. Along with the code please include a readme file that explains the state of your code (What is working and what is not).

Your grade will be based on the demo, whether your code passes a set of test scenarios (a subset of which are provided in your driver.c file), and a manual inspection of the code itself.