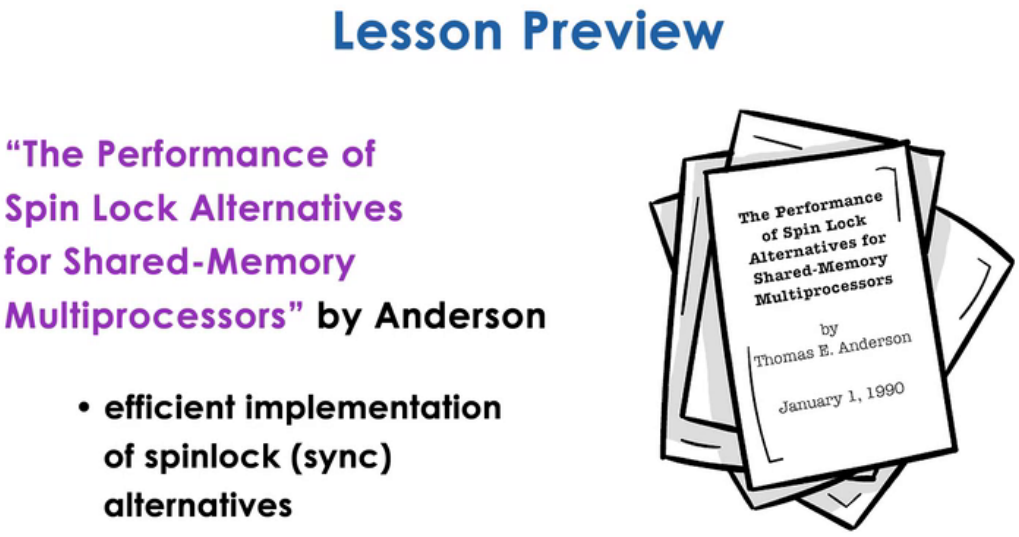
P3L4 - Synchronization Constructs

01 - Lesson Preview

Instructor Notes

- Anderson, Thomas E.
- "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors".
- IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990.



Up to this point, we mentioned synchronization multiple times while discussing other operating system concepts. In this lesson, we will primarily focus on synchronization. We will discuss several synchronization constructs, what are the benefits of using those constructs, and also we will discuss what is the hardware level support that's necessary in order for those synchronization primitives to be implemented.

As we cover these concepts, we will use the paper "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors" by Thomas Anderson. This paper will give you a deeper understanding of how are synchronization constructs implemented on top of the underlying

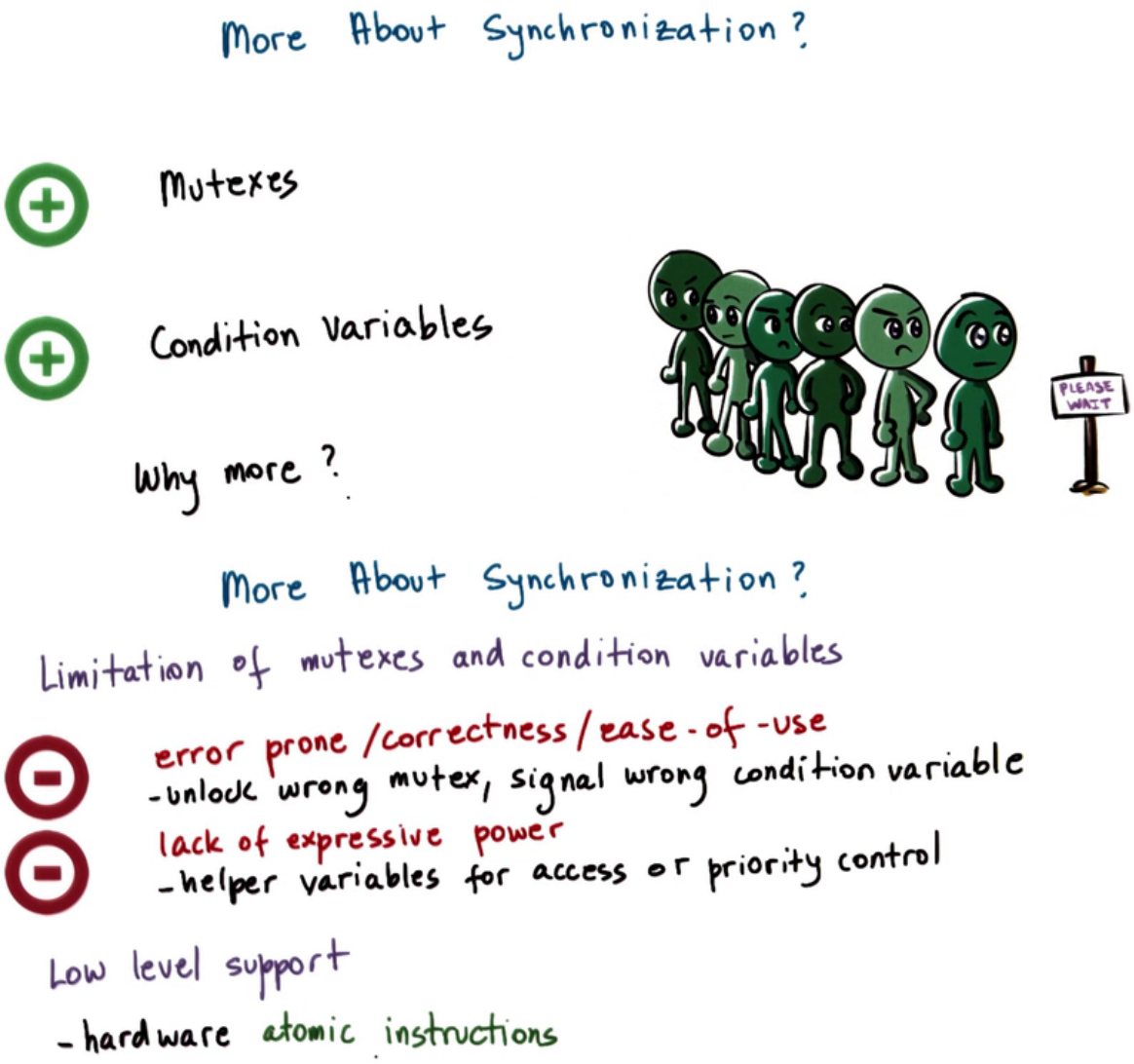
hardware and why they exhibit certain performance strengths. There is a link to the paper in the instructors notes.

02 - Visual Metaphor

Let's illustrate synchronization and synchronization mechanisms using our toy shop example. Synchronization in operating systems is like waiting for a coworker in the toy shop to finish so that you can continue working. When an elf is waiting on a co worker to finish, several things can happen. The elf can continuously check whether the other elf is done or they can ask the coworker to signal them when they're done. At any rate, the time that an elf spends waiting will hurt the performance of the toy shop. For instance, the elf may repeatedly check the other elf whether it's completed the work, asking questions like are you done now, what about now, are you still working. This can have negative impact on the elf that is working because it will delay its processing, he will be annoyed. The second approach of hey, I'm done you can can come do your work now is maybe a little bit more laid back but this other guy may have gone off for lunch in the meantime so it may take it longer to actually come back and proceed with the execution

with the processing of the toy. So regardless, in both of these cases, the workers are wasting some productive time during this wait period. Similar analogies to these exist in synchronization in operating systems. Processes may repeatedly check whether it's ok for them to continue using a construct called spinlocks that's supported in operating systems. We already talked about mutexes and condition variables and we saw that they can be used in order to implement this behavior where a process waits for a signal from another process before it can continue working. Regardless of how we wait and which synchronization mechanism we use, this will affect performance. This can result in CPU cycles that are wasted when we're performing this checking or performance may be lost due to certain cache effects when we're signaling another process that was periodically blocked to come back and start executing again.

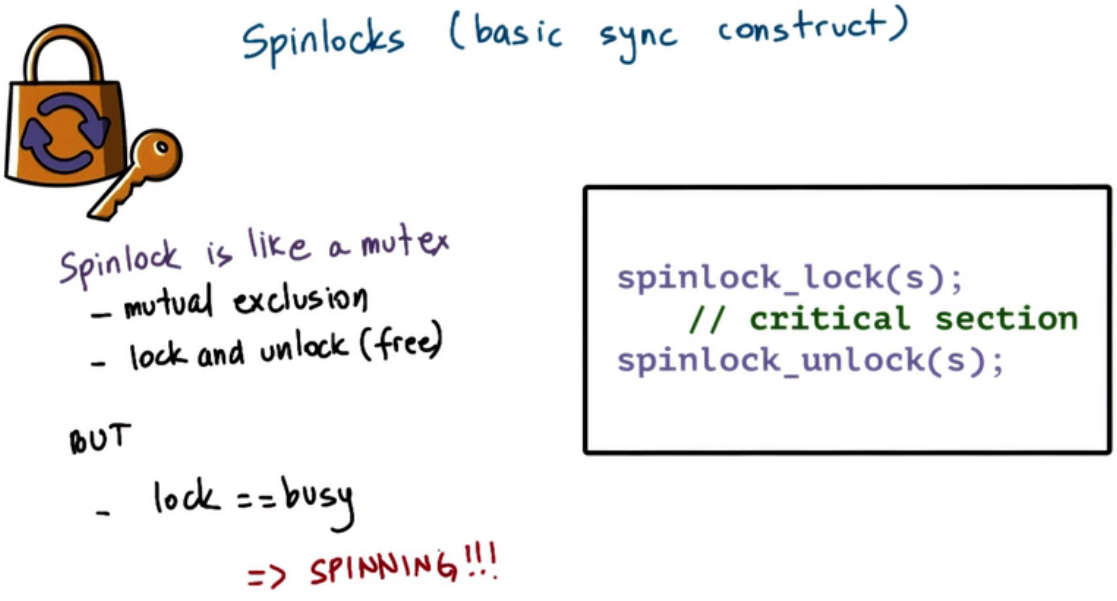
03 - More About Synchronization



We spent already quite a bit of time talking about dealing with concurrency in multithreaded programs. We explained mutexes and condition variables both in generic terms based on

Birrell's paper and also their specific APIs and usage in the context of pthreads. You're probably also even well under way with writing your pthread programs using these constructs. So you're probably asking yourselves now why do we have to talk more about synchronization. We spent quite a bit of time already talking about it and gained experience with it. Well, as we discussed mutexes and condition variables, we mentioned a number of common pitfalls. This included things like forgetting to unlock the mutex or signaling the wrong condition variables. These issues are an indication that the use of mutexes and condition variables is not an error proof process. What that means is that these errors may affect the correctness of programs that use mutexes and condition variables and in general it will affect the ease of use these two synchronization constructs. Also, in the examples that we discussed, we had to introduce some helper variables when we needed to express invariance to control readers/writers access to a shared file. We also need helper variables to deal with certain priority restrictions given that mutexes or condition variables don't inherently allow us to specify anything regarding priority. This implies that these constructs lack expressive power. We cannot easily express with them arbitrary synchronization conditions. Finally, mutexes and condition variables and any other software synchronization construct requires lower level support from the hardware in order to guarantee the correctness of the synchronization construct. Hardware provides this type of low level support via atomic instructions. These are some of the reasons why it makes sense to spend more time talking about synchronization. What we will do, we will look at few other constructs, some of which eliminate some of the issues with mutexes and condition variables and also we will talk about different types of use the underlying atomic instructions to achieve efficient implementations of certain synchronization constructs.

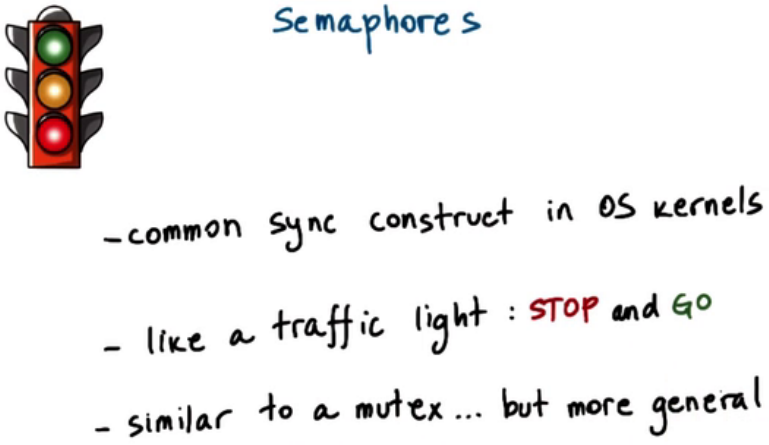
04 - Spinlock



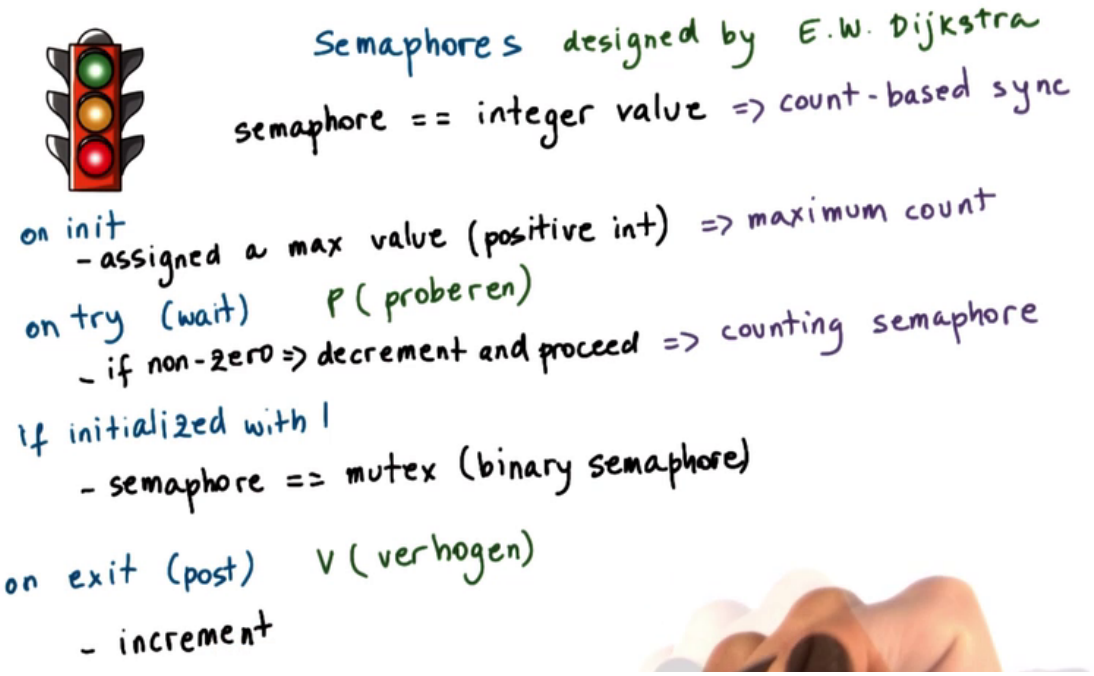
One of the most basic synchronization constructs that's commonly supported in operating systems is Spinlocks. In some ways, spinlocks are similar to a mutex. The lock is used to

protect a critical section to provide mutual exclusion among potentially multiple threads or processes that are trying to perform this critical section and it has certain constructs that are equivalent to the lock and unlock constructs that we saw with mutexes. The use of these lock and unlock operations is similar to the case of the mutexes. If the lock is free then we can acquire it and proceed with the execution of the critical section. And if the lock is not free we will be suspended at this particular point and unable to go on. The way spinlocks differ than mutexes however is that when the lock is busy, the thread that's suspended in its execution at this particular point of time isn't blocked like in the case of mutexes, but instead it is spinning. It is running on the CPU and repeatedly checking to see whether this lock became free. With mutexes, the thread would have relinquished the CPU and allowed for another thread to run. With spinlocks, the thread will spin, it will burn CPU cycles until the lock becomes free or until the thread gets preempted for some reason, for instance maybe it was spinning until ultimately it's time slice expired or potentially a higher priority thread becomes runnable. Because of their simplicity, spinlocks are a basic synchronization primitive and they can be used to implement more complex, more sophisticated synchronization constructs. Because they're a basic construct we will revisit spinlocks a little bit later in this lesson and we will spend some time discussing different implementation strategies for spinlocks.

05 - Semaphores



The next construct we will talk about is semaphores and these are a common synchronization construct that have been part of operating system kernels for a while. As a first approximation, a semaphore acts as a traffic semaphore. It either allows threads to go or it will block them, it will stop them from proceeding any further. This is in some way similar to what we saw with a mutex. It either allows a thread to obtain a lock and proceed with the critical section or the thread is blocked and has to wait for the mutex to become free. However, semaphores are more general than just this behavior that can be obtained through the mutex and we will take a look at what exactly a semaphore is a bit more formally next.



Semaphores are represented via an integer number, some positive integer value. When they're initialized, they're assigned some maximum value, some positive integer. Threads arriving at the semaphore will try it out. If the semaphore's value is non-zero, they will decrement it and proceed. If it is zero, then they will have to wait. This means that the number of threads that will be allowed to proceed will equal this maximum value that was used when the semaphore was first initialized. So as a synchronization construct, one of the benefits of semaphores is that they allow us to express count related synchronization requirements. For instance, 5 producers may be able to produce an item at the same time. The semaphore will be initialized with the number 5 and then it will count and make sure that exactly 5 producers are allowed to proceed. If a semaphore is initialized with 1, then its behavior will be equivalent to a mutex, it will be a binary semaphore. All threads that are leaving the critical section will post to the semaphore, signal the semaphore. What that will do it will increment the semaphore's counter. For a binary semaphore, this is equivalent to unlocking a mutex. Finally, some historic tidbits. Semaphores were originally designed by Dijkstra. He was a dutch computer scientist and also a Turing award winner. The wait and post operations that are used for semaphores were referred to as P and V and P and V come from the dutch words Proberen which means to test out, to try and Verhogen which means to increase. So if you ever see a semaphore used with operations P and V, you will now know what that means.

06 - POSIX Semaphores

POSIX Semaphore API

```
#include <semaphore.h>

sem_t sem;
sem_init(sem_t *sem, int pshared, int count);
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

Here are some of the operations that are part of the POSIX semaphore API. Semaphore.h defines the sem_t sem type. Initializing a semaphore with the sem_init call. This takes as a parameter a semaphore data type variable and also it takes the initialization count and a flag. This flag will indicate whether the semaphore is shared by threads within a single process or across processes. The sem_wait and sem_post operations take as a parameter the semaphore variable that was previously initialized.

07 - Mutex Via Semaphore Quiz

Mutex via Semaphore

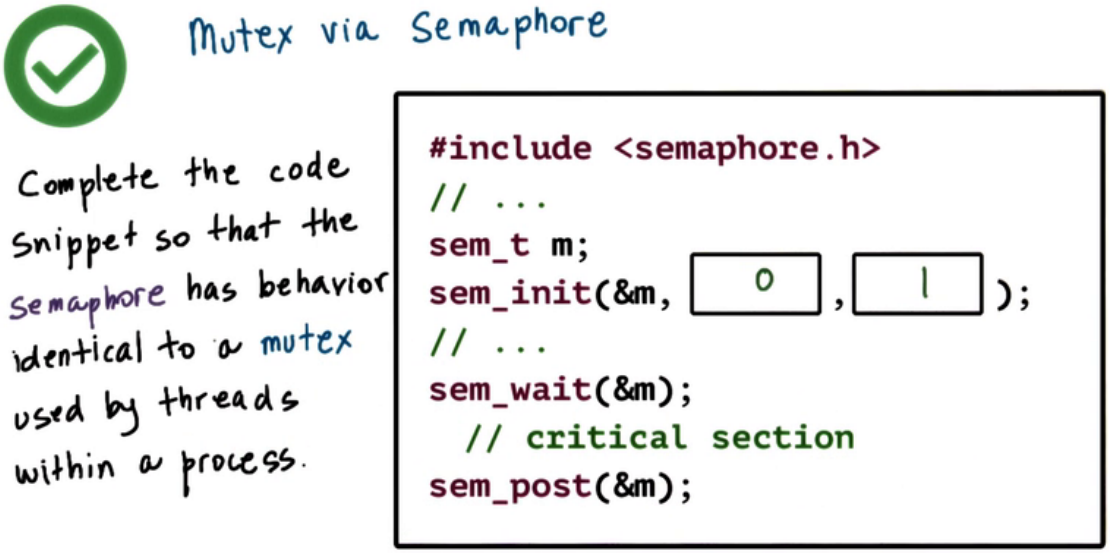
Complete the code snippet so that the semaphore has behavior identical to a mutex used by threads within a process.

```
#include <semaphore.h>
// ...
sem_t m;
sem_init(&m, [      ], [      ]);
// ...
sem_wait(&m);
  // critical section
sem_post(&m);
```

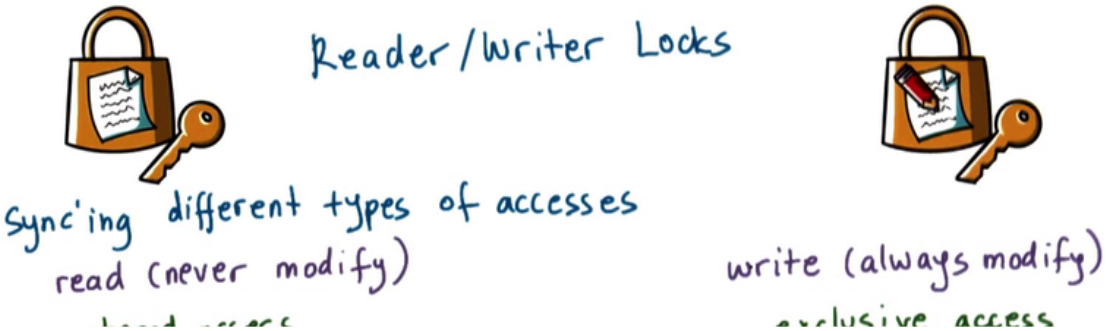
Quiz Help

If you need help, then use a [sem_init() man page] reference.

08 - Mutex Via Semaphore Quiz



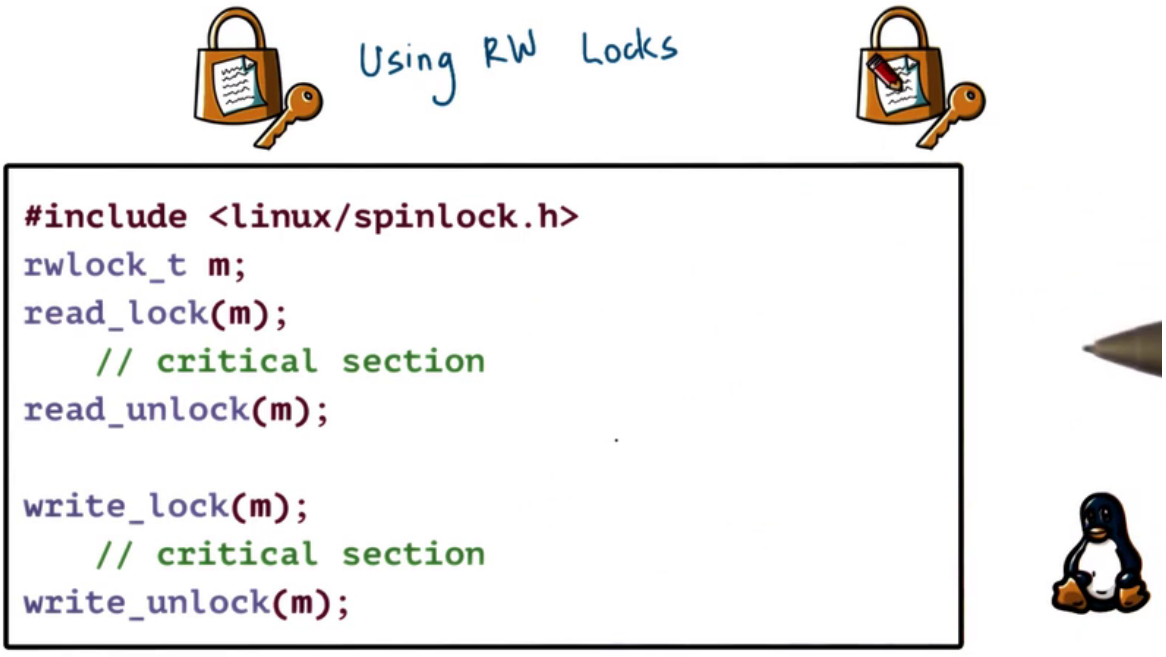
09 - Reader Writer Locks



When specifying synchronization requirements, it is sometimes useful to distinguish among the different types of accesses that a resource can be accessed with. For instance, we commonly want to distinguish those accesses that don't modify a shared resource like reading vs those accesses that do modify shared resources like writing. For the first type of accesses the

resource can be shared concurrently. For the second type of accesses, we require exclusive access. For this reason, operating systems and language runtimes as well support so called reader/writer locks. You can define reader/writer locks and use them in a way that's similar to a mutex however you will specify the type of access read vs write that you want to perform and then underneath the lock behaves accordingly.

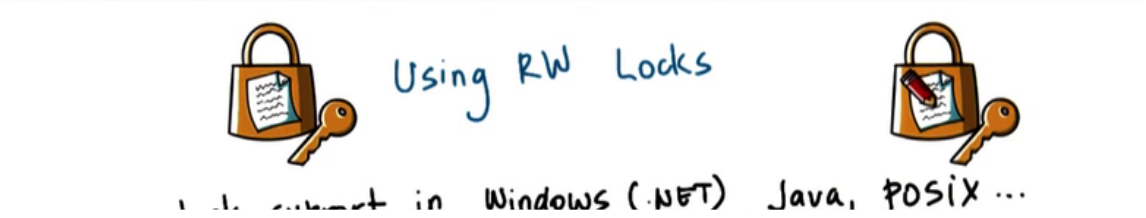
10 - Using Reader Writer Locks



Instructor Notes

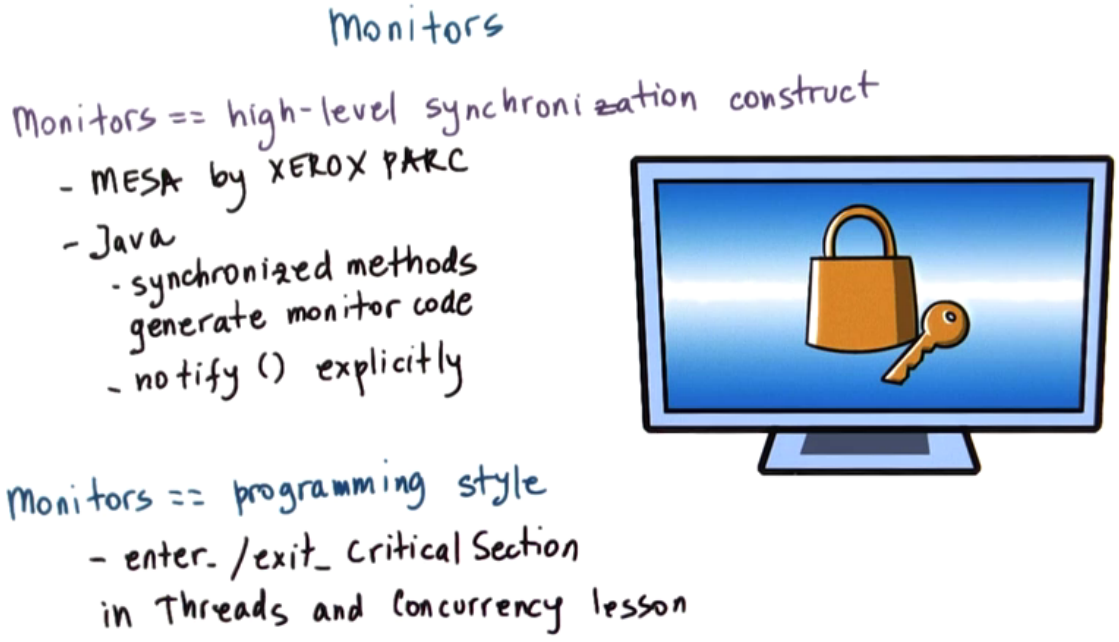
If you would like to explore more about the rwlock_t, then check out /include/linux/rwlock.h.

In Linux, you can define a reader/writer lock by using the provided data type for reader/writer locks. To access a shared resource using this reader/writer lock, you use the appropriate interface read_lock or write_lock. Which one you use will clearly depend on the kind of operation that you want to perform on the shared resource. The reader/writer API also provides the corresponding unlock counterparts for both read and write. A few other operations are supported on reader/writer locks but these are the primary ones. And if you would like to explore more then take a look at the .h file we are providing a link to it in the instructor's notes.



Reader/writer locks are supported in many operating systems and language runtimes. In some of these contexts, the reader/writer operations are referred to as shared locks and exclusive locks. However certain aspects of the behavior of the reader/writer locks are different in terms of their semantics. For instance it makes sense to permit recursive read lock operations to be invoked but then it differs across implementations on exactly what happens on read unlock. In some cases, a read unlock may unlock every single one of the read lock operations that have recursively been invoked from within the same thread. Whereas in other implementations a separate read unlock is required for every single read lock operation. Another way in which implementations of reader/writer locks differ is in their treatment of priorities. For instance in some cases, a reader, an owner of a shared lock, may be given priority to upgrade the lock, so from a reader lock to convert to a writer lock, compared to a newly arriving request for a write lock, or an exclusive lock. In other implementations, that's not the case. So, the owner of the read lock will first release it and then try to reacquire it with write access permissions and contend with any other thread that's trying to perform the same operation. Another priority related difference across reader/writer lock implementations is what kind of interactions there are between the state of the lock, the priority of the threads, and the scheduling policy in the system overall. For instance, it could block a reader, so a thread that otherwise would have been allowed to proceed if there is already a writer that has higher priority and that is waiting on that lock. In this case, the writer is waiting because there are other threads that already have read access to the lock and if there is a coupling between the scheduling policy and the synchronization mechanism, it's possible that a newly arriving reader will be blocked, it will not be allowed to join the other readers in the critical section because of the fact that the waiting writer has higher priority.

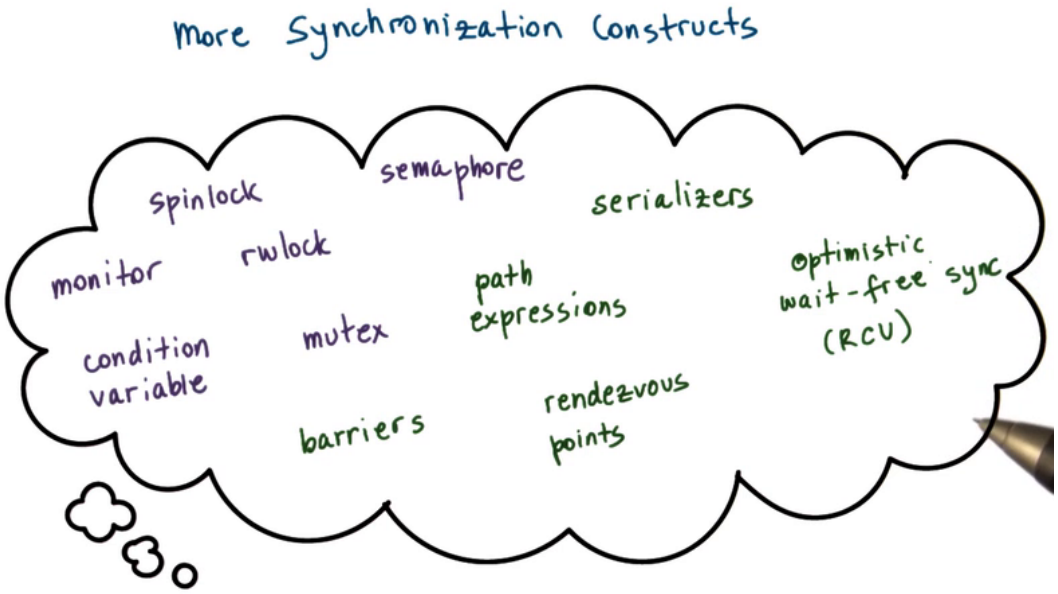
11 - Monitors



One of the problems with the constructs that we saw so far is that they require developers to pay attention to the use of the pairwise operations lock, unlock, wait signal and others. And this as you can imagine is one of the important causes of errors. Monitors on the other hand are a higher level synchronization construct that helps with this problem. In an abstract way, the monitors will explicitly specify what is the resource that's being protected by the synchronization construct, what are all the possible entry procedures to that resource, like for instance if we have to differentiate between readers and writers, and also it would explicitly specify any condition variables that could potentially be used to wake up different types of waiting threads. When performing certain types of access on entry, all of the necessary locking and checking will

take place when the thread is entering the monitor. Similarly, when a thread is done with the shared resource and it's exiting the monitor, all of the necessary unlock operations, checks, any of the signaling that's necessary for the condition variables, all of that will happen automatically, will be hidden from the programmer. Because of all of this, monitors are referred to as a high level synchronization construct. Historically, monitors were included in the MESA language runtime developed by Xerox PARC. Today, Java supports monitors too. Every single object in Java has an internal lock and methods that are declared to be synchronized are entry points into this monitor. When compiled, the resulting code will include all of the appropriate locking and checking. The only thing is that notify has to be done explicitly. Monitors also refer to the programming style that uses mutexes and condition variables to describe the entry and exit codes from the critical section. And this what we described in the threads and concurrency lesson with the enter critical section and exit critical section sections of code.

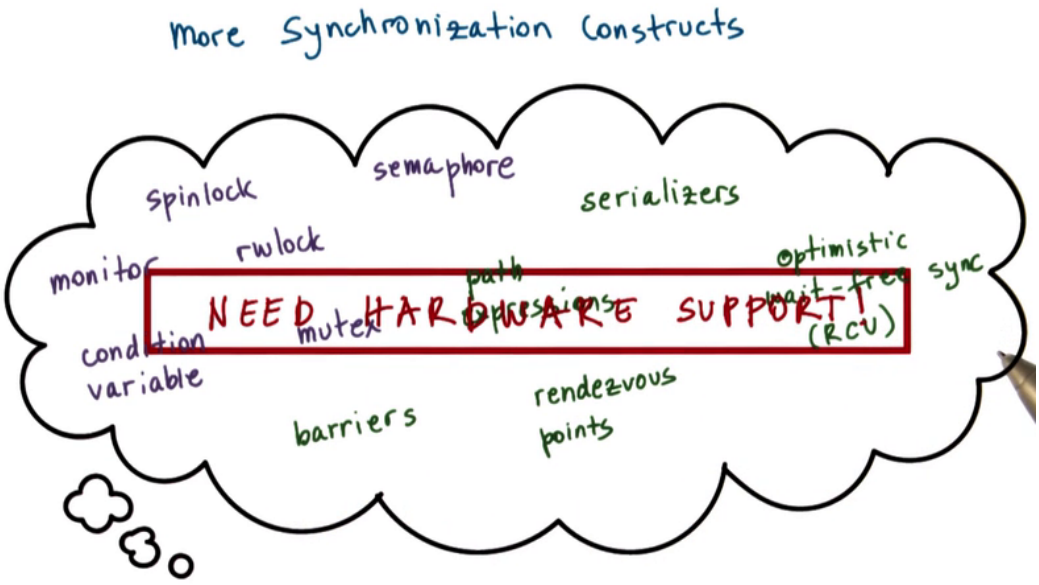
12 - More Synchronization Constructs



Serializers - make it easier to define priorities and then also hide the need for explicit signaling and explicit use of condition variables from the programmers. ***Path Expressions*** - require that a programmer specify the regular expression that captures the correct synchronization behavior. So as opposed to using locks or other constructs, the programmer would specify something like many reads or a single write. And the runtime will make sure that the way the operations are interleaved, that are accessing the shared resource, satisfy that particular regular expression. ***Barriers*** - like a reverse from a semaphore, will block all threads until *n* threads arrive at this point. ***Rendezvous Points*** - waits for multiple threads to reach a particular point of execution. ***Optimistic wait-free sync*** - trying to achieve concurrency without explicitly locking and waiting. Example is ***(RCU) Read Copy Update*** that's part of the Linux kernel.

In addition to the multiple synchronization constructs that we already saw, there are many other options available out there. Some like serializers make it easier to define priorities and then also hide the need for explicit signaling and explicit use of condition variables from the programmers. Others like path expressions require that the programmer specify the regular expression that captures the correct synchronization behavior. So as opposed to using locks or other constructs, the programmer would specify something like, many reads or a single write and the runtime will make sure that the way the operations are interleaved, that are accessing the shared resource, satisfy that particular regular expression.

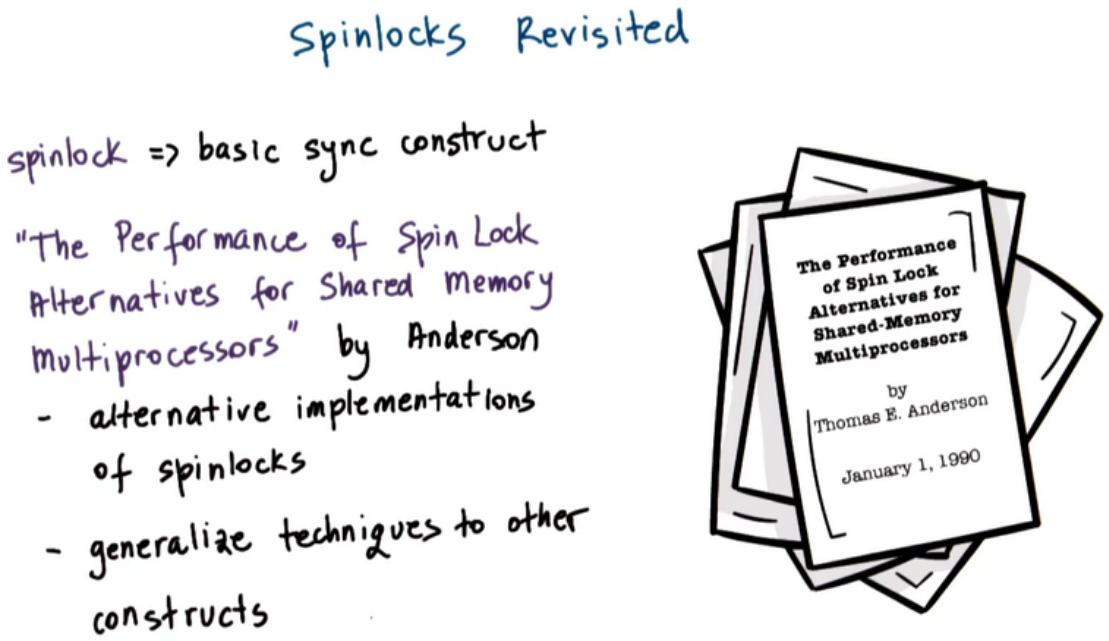
Another useful construct are barriers and these are almost like a reverse from a semaphore in that if a semaphore will allow any threads to proceed before it blocks, a barrier will block all threads until n threads arrive at this particular point. Rendezvous points is also a synchronization construct that waits for multiple threads to meet that particular point in execution. Also for scalability and efficiency, there are efforts to achieve concurrency without explicitly locking and waiting. These approaches all fall into a category that we refer to as wait-free synchronization and they're optimistic in the sense that they bet on the fact that there won't be any conflicts due to concurrent writes and it's safe to allow reads to proceed concurrently. An example that falls into this category is the so called read copy update lock (RCU lock) that's part of the linux kernel.



One thing that all of these methods have in common is that at the lowest level, they all require some support from the underlying hardware to atomically make updates to a memory location. This is the only way they can actually guarantee that the lock is properly acquired and that the state change is performed in a way that is safe and doesn't lead to any race conditions and that all threads in the system are in an agreement of what exactly is the current state of the synchronization construct. We will spend the remainder of this lesson discussing how a

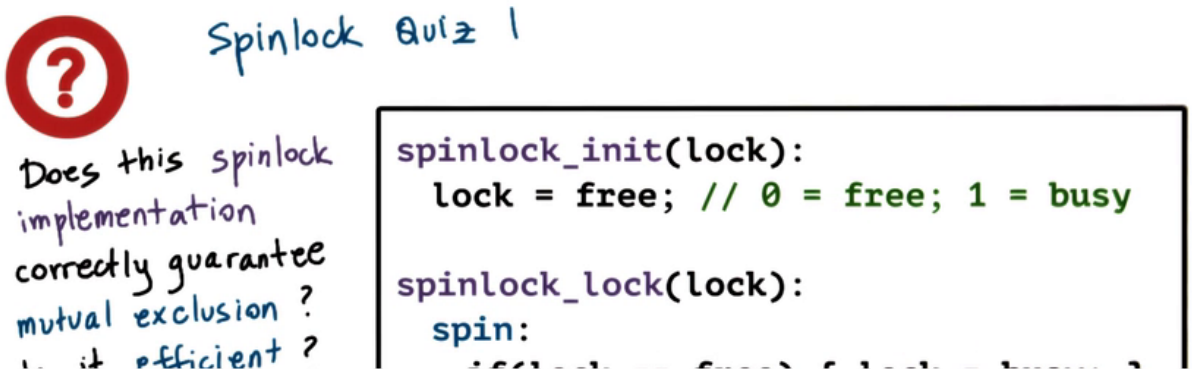
synchronization construct can be built using directly the hardware support that's available from the underlying platform and we will specifically focus on spinlocks as the simplest construct out there.

13 - Sync Building Block Spinlock



We said that spinlocks are one of the most basic synchronization primitives and that they are also used in creating some more complex synchronization constructs. For that reason, it makes sense to focus the remainder of this lesson on understanding how exactly spinlocks can be implemented and what types of opportunities are available for their efficient implementation. To do this, we will follow the paper "The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors" by Tom Anderson. The paper discusses different implementations of spinlocks and this is relevant also for other synchronization constructs that use spinlocks internally. Also some of the techniques that are described in this paper that concern the use of atomic instructions generalize to other constructs and other situations.

14 - Spinlock Quiz 1

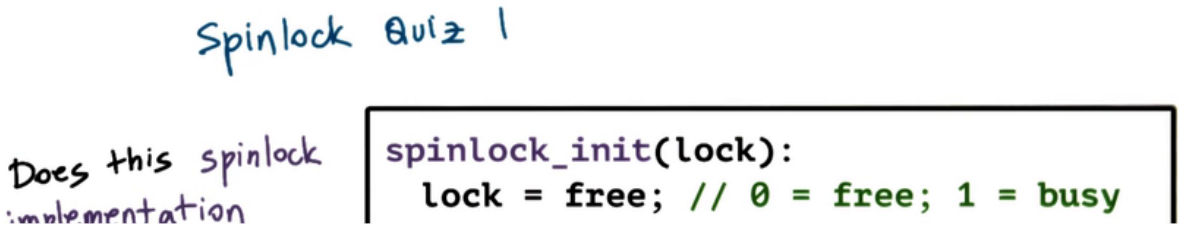


Instructor Notes

Steps of Interaction with a Spinlock

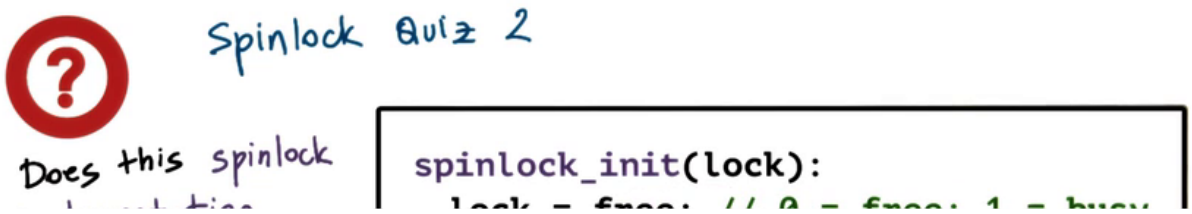
1. The lock needs to be initialized to free
2. To lock the spinlock, check if the lock is free...
    - If the lock is free, then we can change its state (grab the lock)
    - If the lock is not free, then we must keep spinning
3. Finally, we can release the lock by setting it to free

15 - Spinlock Quiz 1



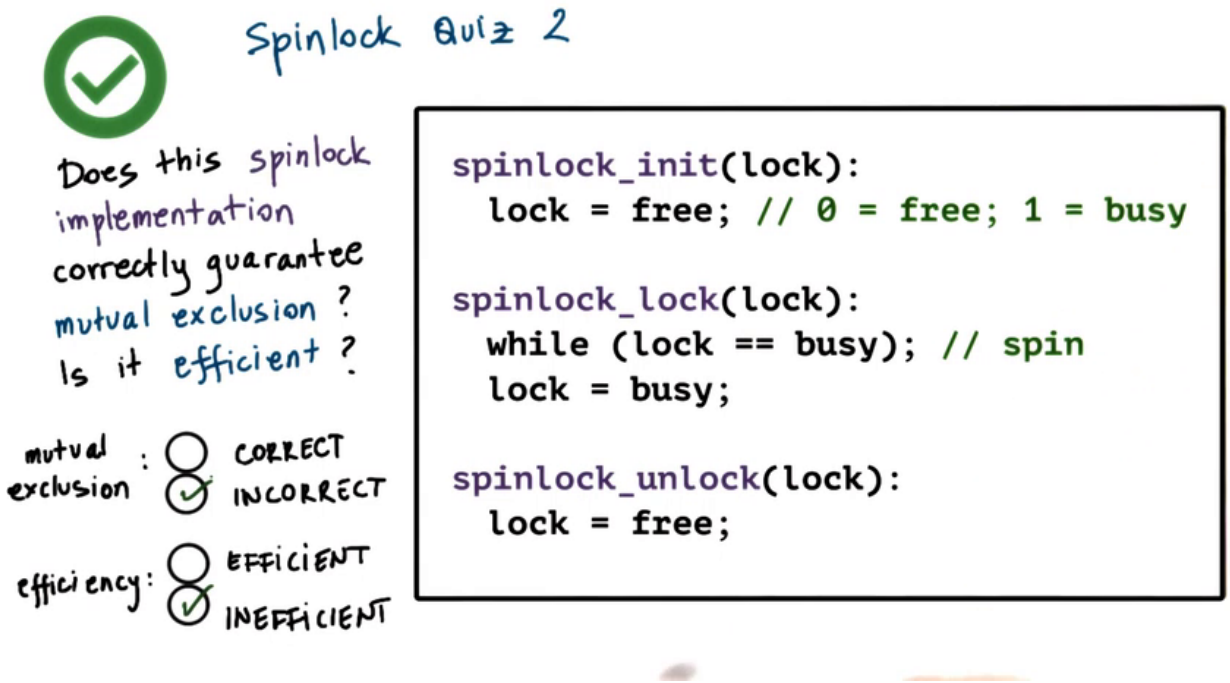
Before we talk about the correctness of the mutual exclusion, let's take a look at the efficiency factor. So, this goto spin statement… as long as the lock is not free, this means that this cycle will repeatedly be executed and this will waste cpu resources. Therefore from efficiency standpoint, this is not an efficient implementation… it's inefficient. But efficiency aside, this solution is also incorrect. In an environment where we have multiple threads that execute concurrently or multiple processes, it is possible that more than one thread will see at the same time that the lock is free and they will move on to perform this lock=busy operation at the same time. Only one thread will actually set the lock value to busy correctly. The other one will simply overwrite it and then it will proceed, it will think it has correctly acquire the lock. As a result, both processes or both threads can end up in the critical section and that clearly is incorrect.

16 - Spinlock Quiz 2



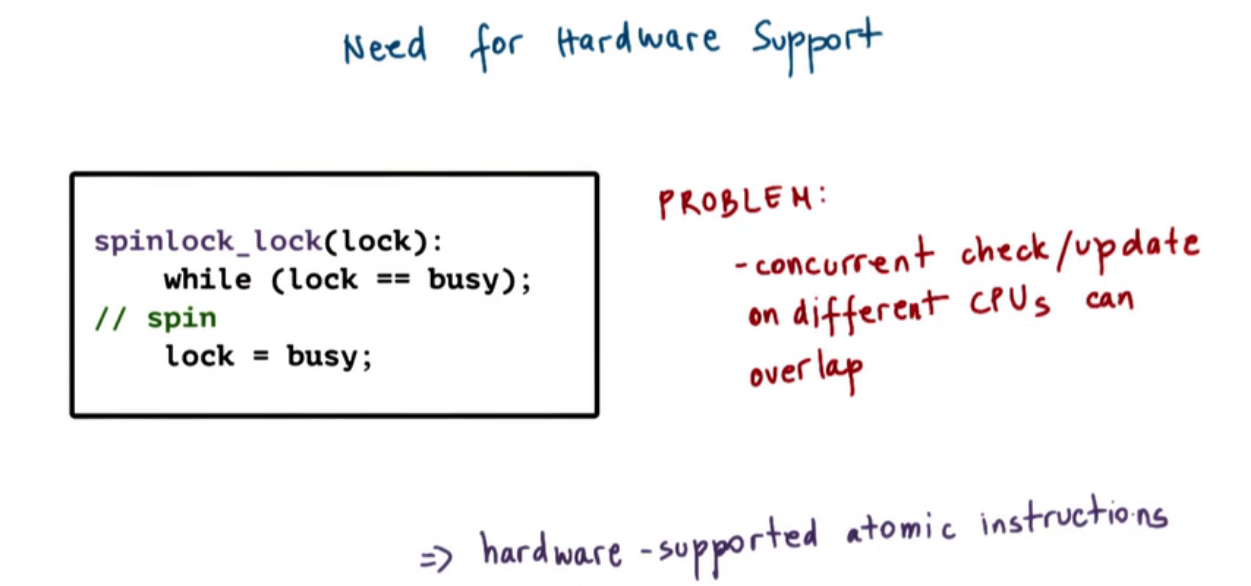
Here is another slightly different version of the same implementation that avoid the goto statement. As long as the lock is busy, the thread will keep spinning, it will remain in this while loop. At some point when the lock becomes free, the thread will exit from this while loop and it will set the lock value to busy so as to acquire it. Now answer the same question as in the previous quiz. Is this implementation of spinlock correct in terms of its ability to guarantee mutual exclusion and also is this an efficient implementation?

17 - Spinlock Quiz 2



Again as in the previous case, this implementation will be both inefficient and incorrect. The inefficiency comes from the fact that again we have continuous loop that is spinning as long as the lock is busy. Now the implementation is incorrect because although we did put this while check here, multiple threads again will see that the lock is free once it becomes free, they will exit this while loop and will move on here and try to set the lock value to be busy. If these threads are allowed to execute concurrently, there is absolutely no way purely in software to guarantee that there won't be some interleaving of exactly how these threads perform this check and these set operations and that a race condition will not occur here. We can try to come up with multiple purely software based implementations of a spin lock and we'll ultimately come to the same conclusion that we need some kind of support from the hardware in order to make sure that some of this checking and setting of the lock value happens atomically via hardware support.

18 - Need for Hardware Support



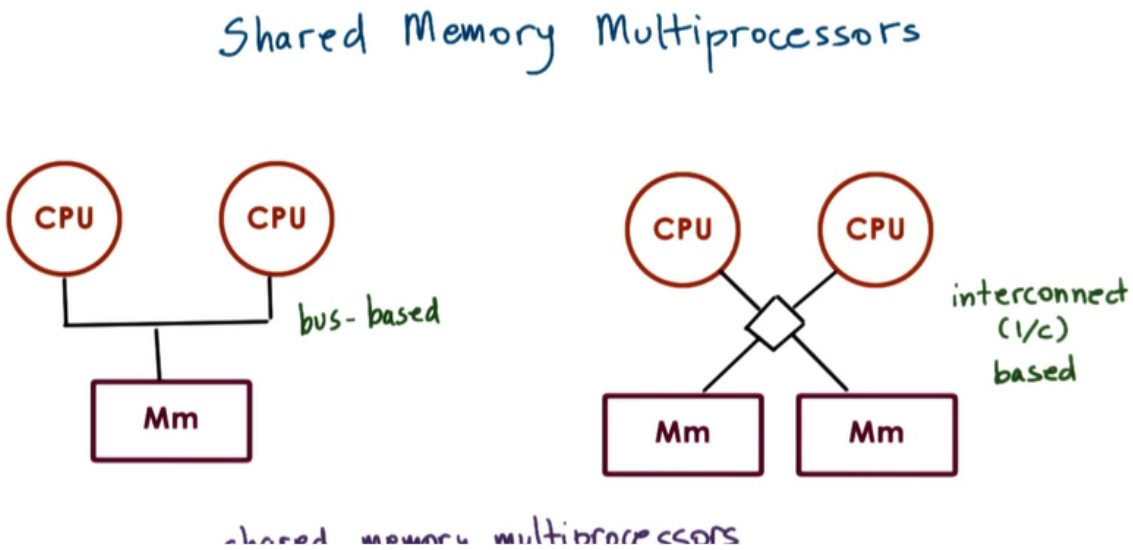
So we need to get some help from the hardware. Looking at the spinlock example from the previous video, we somehow need to do the checking of the lock value and the setting of the lock value to happen indivisibly, atomically, so that we can guarantee that only 1 thread at a time can successfully obtain the lock. The problem with this implementation is that it takes multiple cycles to perform the check and the setting and during these multiple cycles threads can be interleaved in arbitrary ways. If they're running on multiple processors, their execution can completely overlap in time. To make this work, we have to rely on support from hardware supported atomic instructions.

19 - Atomic Instructions

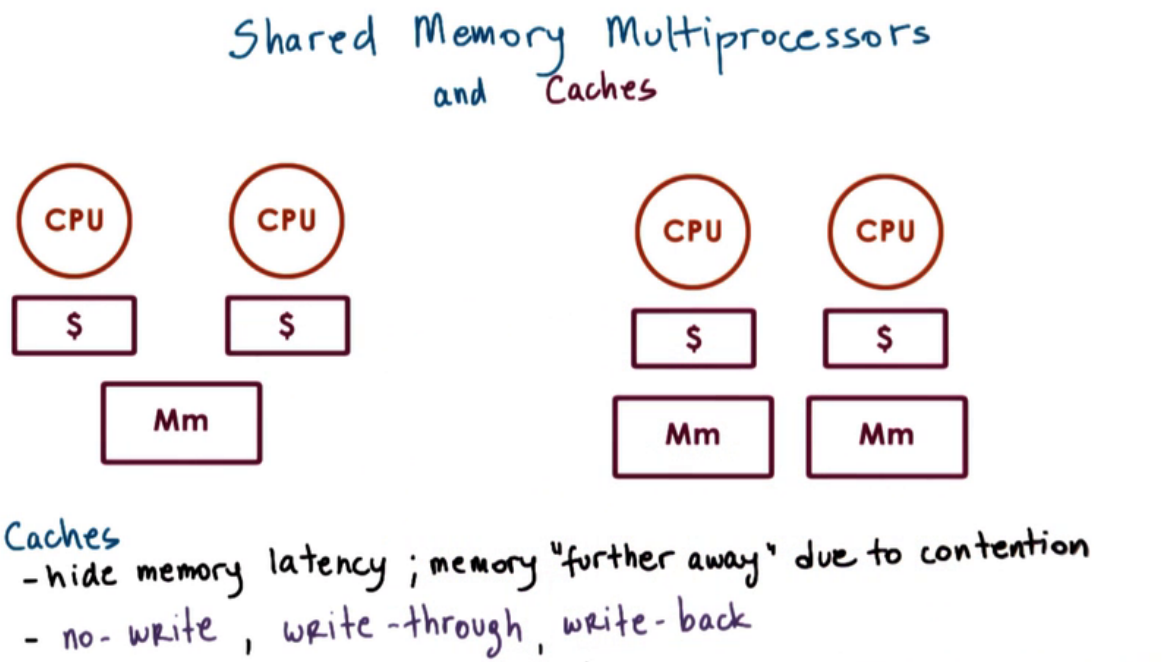
Each type of hardware or hardware architecture will support a number of atomic instructions. Some examples include test_and_set or read_and_increment or compare_and_swap. Different instructions may be supported on different hardware platforms. Not every architecture has to support every single one of the available atomic instructions out there. As you can tell from the names of these instructions, they all perform some multistep, multicycle operation, but because they're atomic instructions, the hardware makes guarantees that this set of operations that are included in these instructions will happen atomically, so not just halfway, the entire operation or none of it, that it will happen in mutual exclusion, meaning that only 1 instruction at a time will be allowed to perform the appropriate operation and that the remaining ones will be queued up, will have to wait. So if we think about this, what this means is that the atomic instructions specify some operation and this operation is the critical section and the hardware supports all of the synchronization related mechanisms that are required for that operation. If we look at our spinlock example using the first atomic test and set operation, the spinlock implementation can look as follows. Here test_and_set atomically returns or tests the original value of the memory location that's passed as parameter, lock in this case, and sets the new value of this memory location to be one. This happens atomically. Remember 1 to us indicates that the lock is busy. When we have multiple threads that are contending for this lock, when they're trying to execute this spinlock operation, only 1 needs to successfully acquire the lock. The very first thread that comes to execute the test_and_set, for that thread test_and_set will return 0 because the original value of the lock was zero, originally the lock was free. That thread will therefore exit the while loop because test_and_set will return 0 (free) and that is different than busy (1). That is the only thread that will acquire the lock and proceed with the execution. All of the remaining

threads that try to execute test_and_set, for them test_and_set will return 1 because this first thread already set the value of lock to be 1. Therefore, those remaining threads will just continue spinning into this while loop. Notice that in the process, these other threads still repeatedly resetting the value of the lock to 1. So as they're spinning through the while loop every single time they try to execute this test_and_set operation this sets the value of the lock field to be 1 again to be busy however that's ok. The very first thread when it executed test_and_set, that already set the value of the lock to be busy and these other threads are not really changing the fact that the lock is indeed locked. Which specific atomic instructions are available on a given platform varies from hardware to hardware. Some operations like test_and_set are pretty prevalent. Others like read_and_increment may not be available on all platforms. In fact, we may have version of this where in some cases there is availability of an atomic operation that atomically increments something but it does not necessarily return the old value. In other cases there may be atomics that read_and_decrement as opposed to read_and_increment. In addition there may be differences in the efficiencies with which different atomic operations execute on different architectures. For this reason, software such as synchronization constructs that are built using certain atomic instructions has to be ported, we have to make sure that the implementation actually uses one of the atomic instructions that's available on the target platform. Also we have to make sure that the implementation of software of these synchronization constructs is optimized so that it uses the most efficient atomics on a target platform and so that it uses them in an efficient way to begin with. Anderson's paper presents several alternatives on how spinlocks can be implemented using available hardware atomics and we will discuss these in the remainder of this lecture.

20 - Shared Memory Multiprocessors



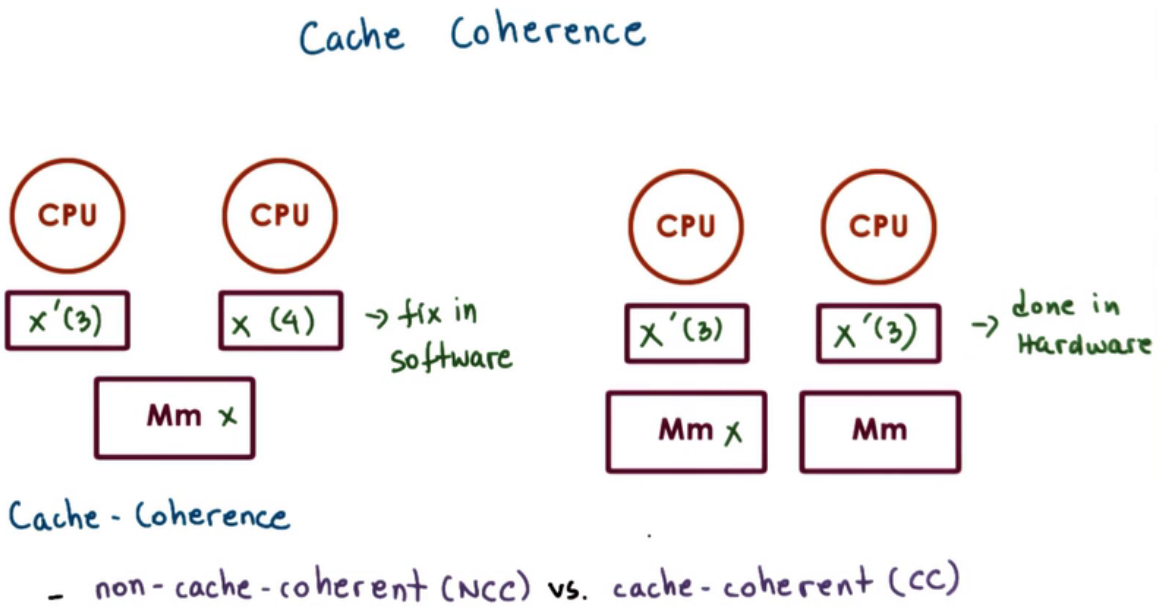
Before moving on with the discussion of the spinlock alternatives that are presented in Anderson's paper, let's do a refresh on multiprocessor systems and their cache coherence mechanisms. This is necessary in order to understand the design tradeoff and the performance trends that will be discussed in this paper. A multiprocessor system consists of more than one cpu and some memory that is accessible to all of these CPUs. The shared memory may be a single memory component that's equidistant from all the CPUs or there may be multiple memory components. Regardless of the number of memory components, they are somehow connected to the CPUs and in this figure I'm showing an interconnect based connection and this is more common in current systems or a bus based connection which was really more common in the past. Also note here I'm drawing the bus based connection to apply to a configuration where there is only a single memory module, however the bus based configuration can apply to both of these situations and vice versa. The one difference is that in the interconnect based configuration I can have multiple memory references in flight where one memory reference is applied to this memory module and another one to the other memory module, whereas if I have a bus based configuration in this case, the shared bus only one memory reference at a time can be in flight regardless of whether these references are addressing a single memory module or spread out across multiple memory modules. So the bus is basically shared across all the modules. Because of this property, the memory is accessible to all CPUs, these systems are called shared memory multiprocessors. Other terms used to refer to shared memory multiprocessors are also symmetric multiprocessors or for short SMPs.



Blank Video >3:09

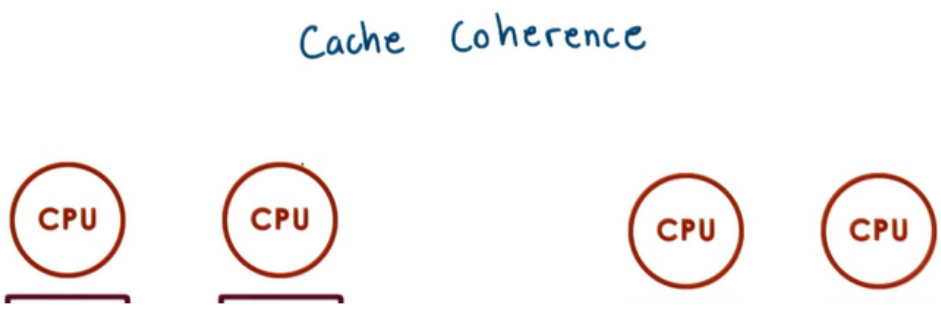
In addition, each of the CPU's in these kinds of systems can have caches. Access to the cache data is faster so caches are useful to hide the memory latency. The memory latency is even more of an issue in shared memory systems because there is contention on the memory module. Because of this contention, certain memory references have to be delayed and that adds to the memory latency even more so than before. So, it is as if the memory is further away from the CPU's because of this contention effect. So when data is present in the cache, the CPU's will read data from the cache instead of memory and that will have an impact on performance. Now when CPU's perform a write, several things can happen. First, it may not even allow a write to happen to the cache. A write will directly go to memory and any cached copy of that particular memory location will be invalidated. Second, the CPU write may be applied both to the cached location as well as directly in memory. So this technique is called write-through, we write both in the cache as well as in memory. And finally, on some architectures, the write can be applied in cache but the actual update to the appropriate memory location can be delayed and applied later. For instance, when that particular cache line is evicted. We call this technique write-back.

21 - Cache Coherence



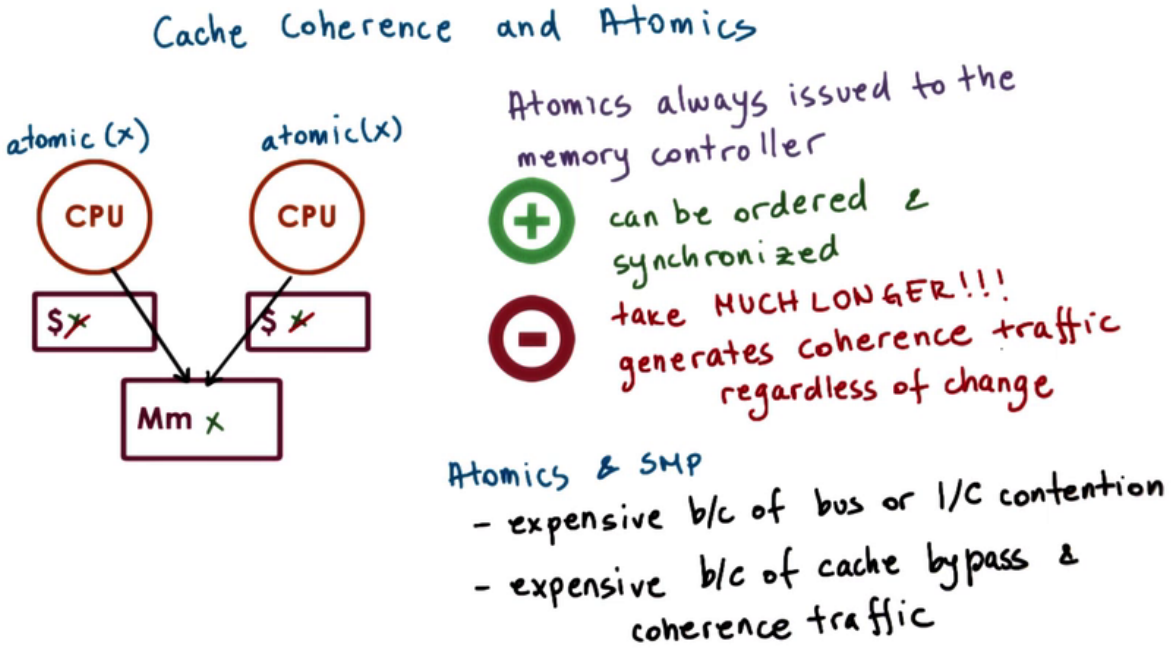
One challenge is what happens when multiple CPU's reference the same data, x in this case. The data can appear in multiple caches or in this case since we have multiple memory modules, x is actually present in one of the memory modules but both of the cpu's referenced it and so it appears in both of their caches. On some architectures, this is a problem that has to be dealt with purely in software. Otherwise the caches will be not coherent. For instance, if on one CPU we update x to be 3, the hardware is not going to do anything about the fact that the value of x in the cache of another CPU is 4. This will have to be fixed in software. These are called non-cache-coherent (NCC) architectures. On other platforms however, the hardware will take care of all of the necessary steps to make sure that the caches are coherent. So when one CPU updates x to be 3, the hardware will make sure that the cache on the other CPU is also updated. These are called cache coherent (CC) platforms.

The basic methods that are used in managing the cache-coherence are called write-invalidate and write-update. Let's see what happens with each of these methods when we have a situation where a certain value is present in all of the caches on these two different platforms. In the write-invalidate case, if one CPU changes the value of x, then the hardware will make sure that if any other cache has cached that particular variable x, that value will be invalidated. Future references on this CPU to that same value x will result in a cache miss and will be pushed over to memory.



The memory location clearly has to be updated based on one of the methods write through or write-back.
In the write-update case, when one CPU changes the value of x to x' then the hardware makes sure that if any other cache has cached that same memory location, its value gets updated as well. As the name of this method suggests. Subsequent accesses to this memory location from the other CPU will result in a cache hit and will return the correctly updated new value.
The tradeoff is that with write invalidate, we actually post lower bandwidth requirements on the shared interconnect in the system. This is because we don't actually have to send the full value of x, just its address so that it can be invalidated on other caches. Plus, once a cache line is invalidated, future modifications to the same memory location will not result in subsequent invalidations, that location is already invalidated. So if the data isn't needed on any of the other cpu's anytime soon, it is possible to amortize the cost of the coherence traffic over multiple changes, basically x will change multiple times over here before it's needed on the other CPU and it's only going to be invalidated once. That's what we mean by this amortized cost. For write-update architectures, the benefit is that the data will be available immediately on the other CPU's that need to access it. We will not have to pay the cost to perform another memory access in order to retrieve the latest value of x. So then clearly programs that will need to access the value of x immediately after it's been updated on another CPU will benefit from support for write-update. Note that you as a programmer, you don't really have a choice in whether you will use write-update or write-invalidate. This is a property of the hardware architecture and whichever policy the hardware uses you will be stuck with it.

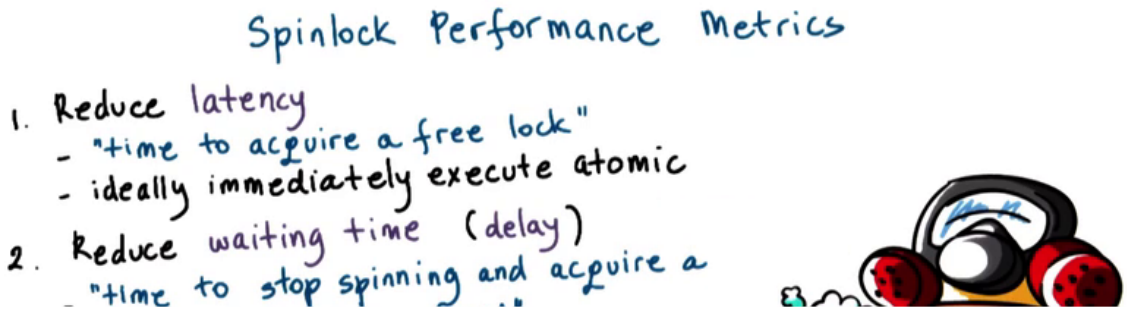
22 - Cache Coherence and Atomics



One thing that's important to explain is what exactly happens with cache coherence when atomic instructions are used. Recall that the purpose of atomic instructions is to deal with issues that are related to the arbitrary interleaving of threads that are sharing a CPU as well as threads that are concurrently executing across multiple CPUs. Let's consider the following situation. We have two CPUs and on both of these CPUs we'll need to perform some atomic instruction that involves the memory location of x. And this x has been cached in both of these CPUs. The problem then is how to prevent multiple threads on these different CPUs to concurrently access the cached values of x. If we allow the atomic instructions to read that updated cached value of the memory reference, that's the target of the atomic instruction, there can be multiple problems. We have multiple CPUs with caches and we don't know where that value has been cached, we have write-update vs write-invalidate protocols, we have latency on the chip. Given all of these issues, it's really challenging if a particular atomic is applied to the cache on one CPU to know whether or not on another CPU another atomic is attempted against the cached value on that CPU. For that reason, atomic operations bypass the caches and they always directly access the memory location where the particular target variable is stored. By forcing all atomics to go directly to the memory controller, this is going to present a central entry point where all of these references can be ordered and synchronized in a unique manner. So none of the race conditions that could have occurred if we allowed atomics to access the cache, they just won't occur in this situation. This will solve the correctness problem but it will raise another issue. Atomics will take longer than other types of instructions. They will always have to access memory and they will also contend on memory. In addition, in order to guarantee atomic behavior, we have to generate the coherence traffic in either update or invalidate all of the cached copies of this memory reference. Even if the value of this memory location doesn't

change with the atomic operation, we still have to perform this step on forcing coherence traffic, so either invalidating or forcing the same value to be reapplied to the cache. Regardless of whether or not this location changes. This is necessary in order to stay on the side of safety and to be able to guarantee correctness of the atomic operations. In summary, atomic instructions on SMP systems are more expensive compared to on a single CPU system because there will be some contention for the shared bus or the shared interconnect. In addition, atomics in general are more expensive because they will bypass the cache in these kinds of environments and they will trigger all of the coherence traffic regardless of what happens with the memory location that's the target of the atomic instruction.

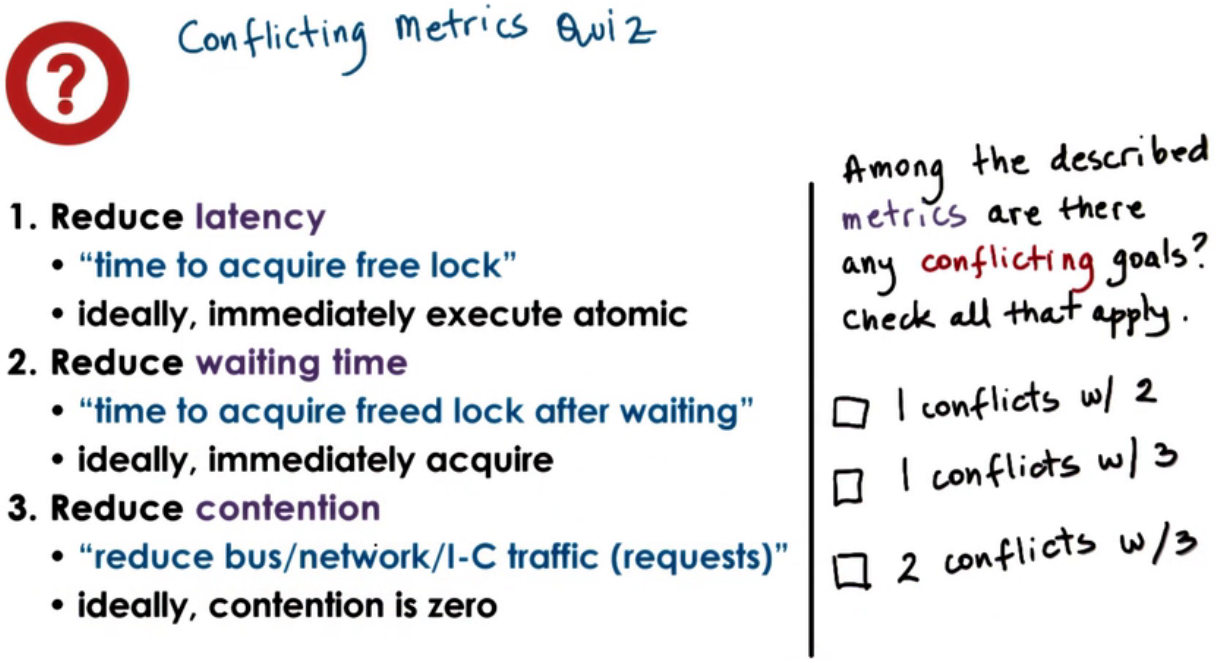
23 - Spinlock Performance Metrics



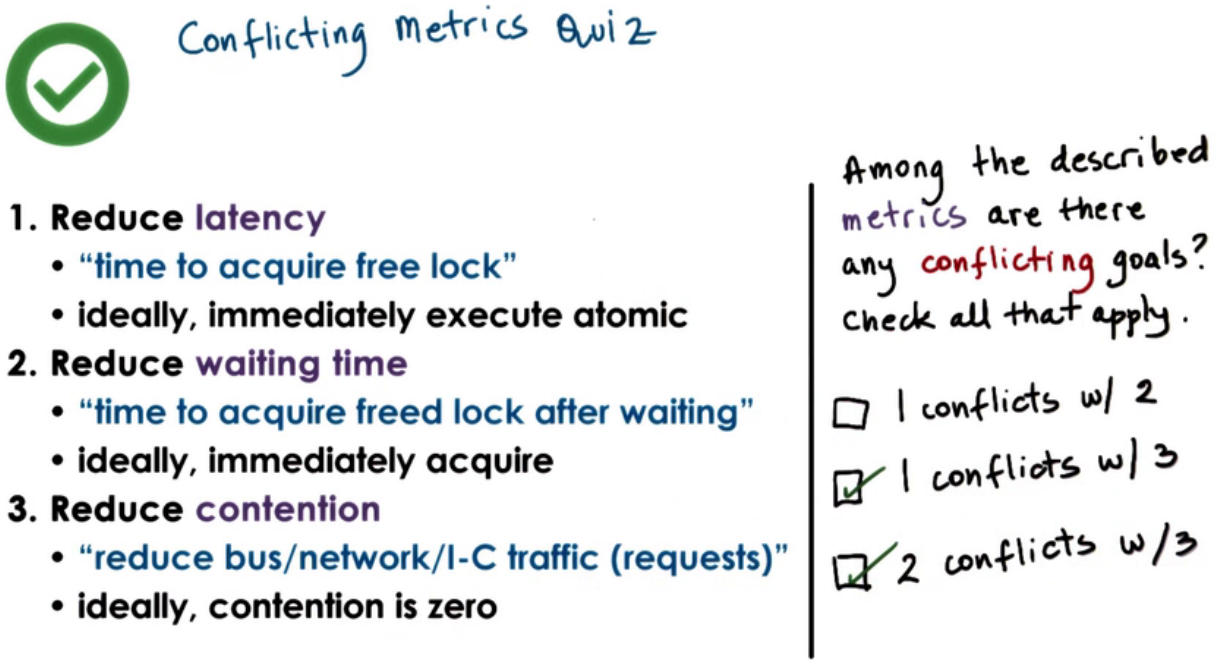
With this background on SMP's, cache coherence, and atomics, we're now finally ready to discuss the design and performance trends of spinlock implementations. The only one thing that's left to decide are what are the performance metrics that are useful when reasoning about different implementations of spinlocks. To determine this we should ask ourselves what are our objectives. First we want the spinlocks to have low latency. By latency, we're referring to how long does it take for a thread to acquire a lock when it's free? Ideally we want the thread to be able to acquire a free lock immediately with a single instruction and we already established that spinlocks require atomic instructions so the ideal case will be that we just want to be able to execute a single atomic instructions and be done. Next we want the spinlock to have low delay, or to have low waiting time. What that means is that when a thread is spinning and a lock becomes free, we want to reduce that time it takes from the thread to stop spinning and to acquire that lock that has just been freed. Again ideally, we would like for the thread to be able to do that as soon as possible, as soon as this lock is freed for a thread to be able to acquire it, so to execute an atomic. And finally we need a design that will not generate contention on the shared bus or the shared network interconnect. By contention, we mean both the contention

that's due to the actual atomic memory references as well contention that's generated due to the coherence traffic. Contention is bad because it will delay any other CPU in the system that's trying to access the memory, but more importantly it will also delay the owner of the spinlock and that is the thread, that is the process that's trying to quickly complete the critical section and release the spinlock. So if we have a contention situation, we may potentially be delaying the unlock operation for the spinlock. That will clearly impact performance even more. So these are the three objectives that we want to achieve in a good spinlock design. And the different alternatives that we will discuss in this lesson will be evaluated on these criteria.

24 - Conflicting Metrics Quiz



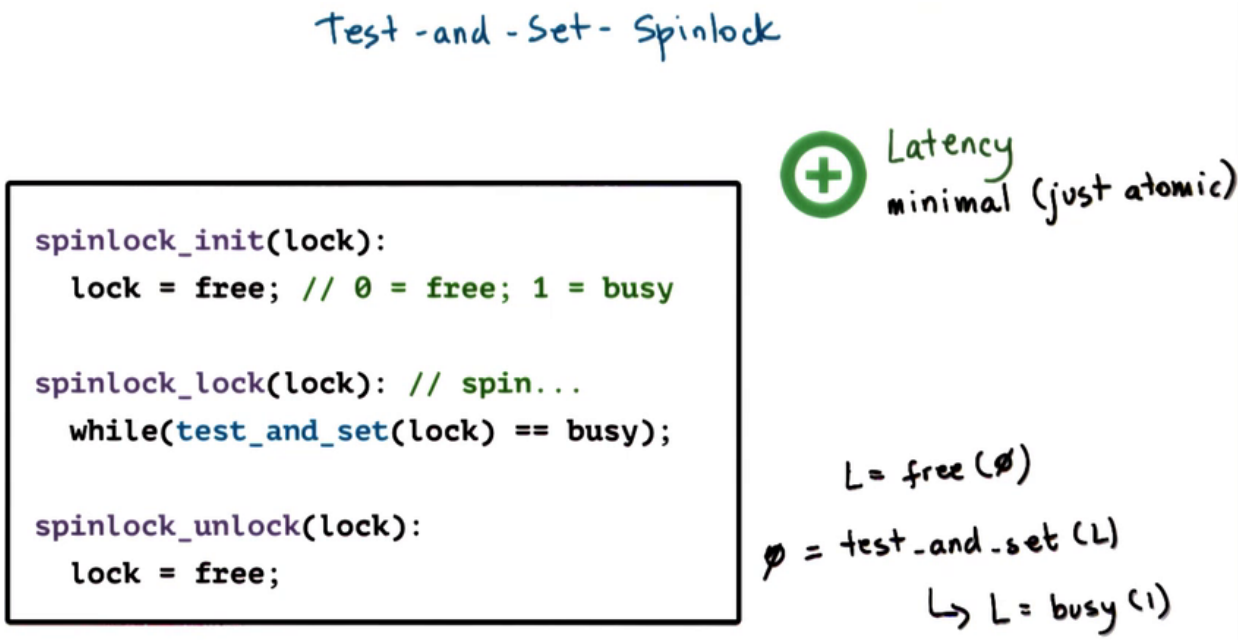
25 - Conflicting Metrics Quiz



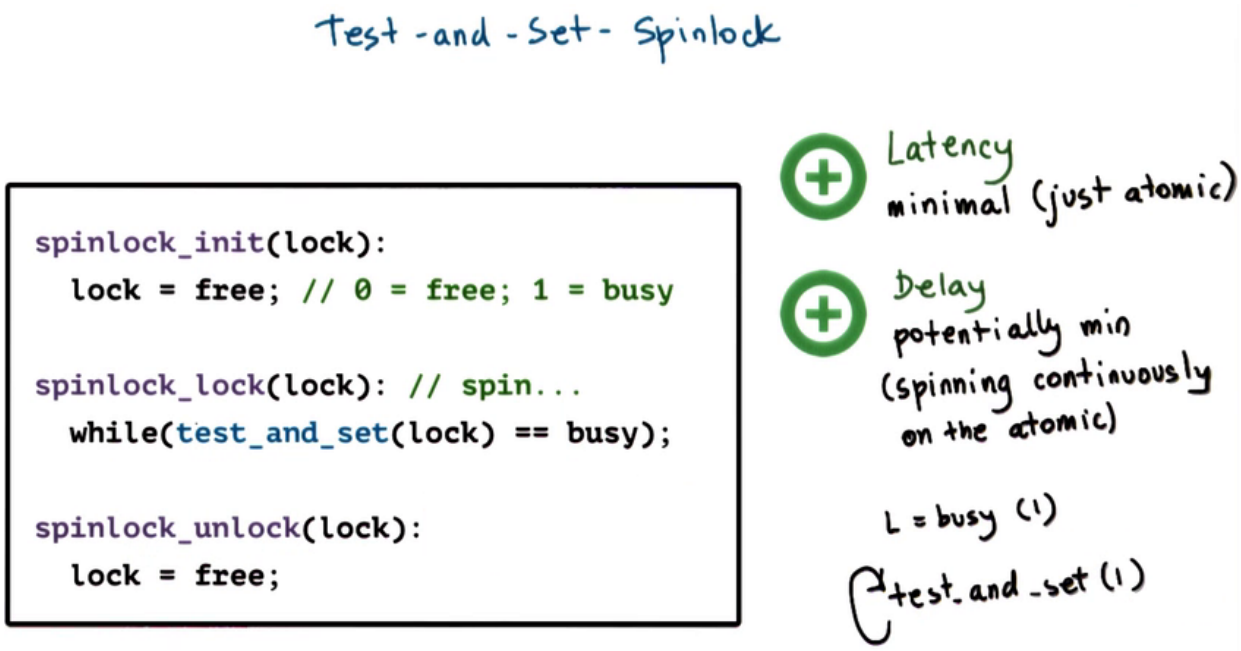
Goal #1 is to reduce latency and this implies that we want to try to execute the atomic operation as soon as possible. As a result, the locking operation will immediately incur an atomic operation and that can potentially create some additional contention on the network. Therefore 1 conflicts with 3.

Similarly with #2, if we want to reduce the waiting time, the delay, then we have to make sure we're continuously spinning on the lock as long as it's not available so that we can detect as soon as possible that the lock is freed and we can try to acquire it. Again this will create contention, so 2 is conflicting with 3. As for any conflicts between reducing latency and reducing waiting time or delay, it is hard to answer this in a general case, it will really depend on from one algorithm to another. So we're not going to mark this as an answer.

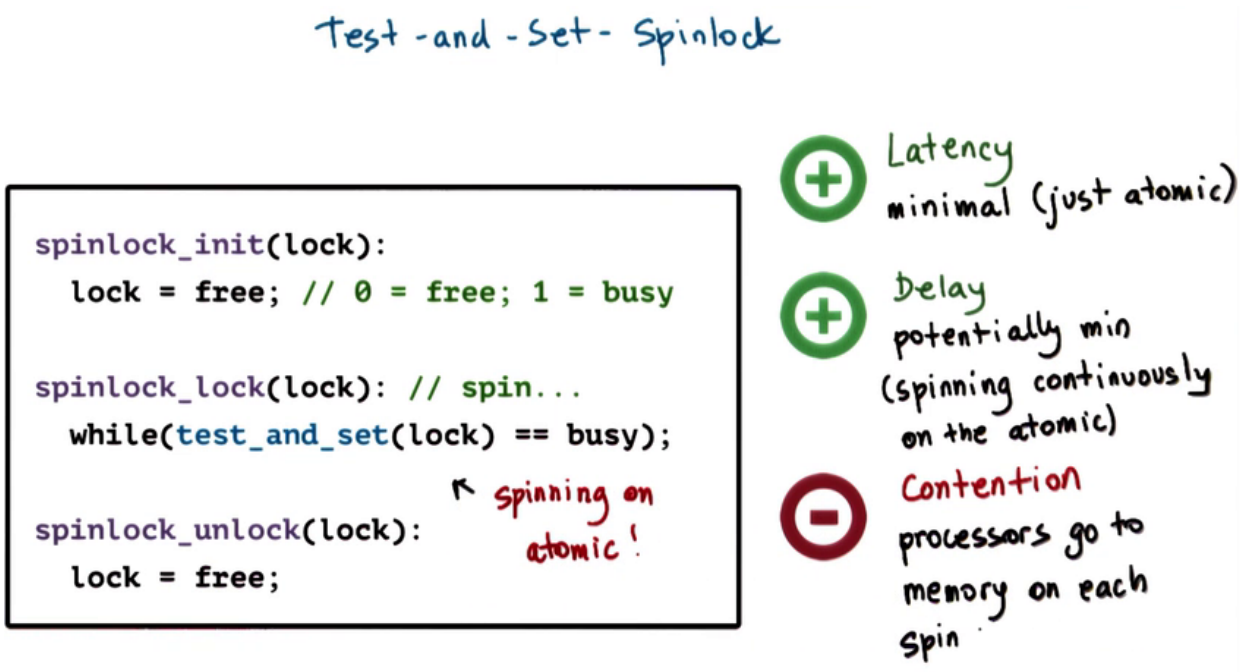
26 - Test and Set Spinlock



Let's look at the simple spinlock implementation we showed earlier in this lesson that uses the test_and_set instruction. In this one and all of the other examples that we will show, we will assume that the lock initialization step sets the lock to be free and that zero indicates free and 1 indicates busy, that the lock is locked. The nice thing about this lock is that the test_and_set instruction is a very common atomic instruction that most hardware platforms support. So as code, it will be very portable, we will be able to have the exact same code run on different hardware platforms. From a latency perspective, this spinlock implementation performs as good as it gets. We only execute the atomic operation and there's no way we can do any better than this. Note that the lock was originally freed, so it was zero, and then as soon as we execute the spinlock_lock operation, we make an attempt to execute test_and_set, the lock is free so this will return zero, as a result of that we will come out of this while loop, and also the test_and_set will change the value of lock to 1 so the lock is busy. So at that point we have clearly come out of the while loop, we're not spinning, and we have the lock and we have marked that the lock is busy.



Regarding delay, this particular implementation potentially could perform well because we see that we're continuously spinning on the atomic instruction. As long as the lock is busy, test_and_set will return 1 (busy) so we will remain in this while loop, however whenever the lock does become free, this or one of the other test_and_set will immediately detect it and come out of the while loop. That single test_and_set operation also will again set the value of lock to 1 so any other test_and_set attempts will result in spinning again.



Now we already said there is a conflict between latency and delay and contention so clearly this lock will not perform well from a contention perspective. As long as there is spinning, every single processor will repeatedly go on the shared interconnect, on the shared bus to the memory location where the lock is stored given that it's repeatedly trying to execute the atomic instruction. This will create contention, delay the processing that's carried out on other CPUs, it will delay the processing that the lock owner needs to perform who's trying to execute the critical section, so therefore it will also delay the time when the lock actually becomes freed. So it's just bad all over. The real problem with this implementation is that it continuously spins on the atomic instruction. If we don't have cache coherence, we will really have to go to memory in order to check what is the value of the lock, but with this implementation even if we do have coherent caches, they will be bypassed because we're using an atomic instruction so in every single spin we will go to memory regardless of the cache coherence. That clearly is not the most efficient use of atomics or of hardware that does have support for caches and cache coherence.

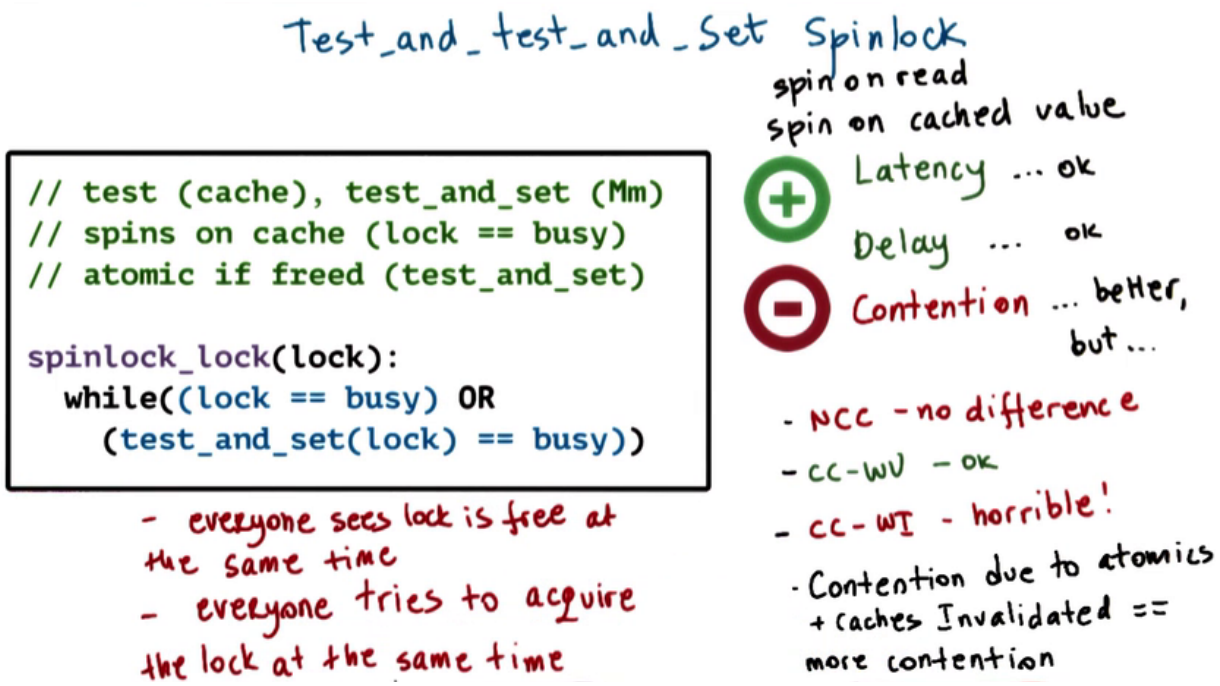
27 - Test and Test and Set Spinlock

```
spinlock_init(lock):
  lock = free; // 0 = free; 1 = busy

spinlock_lock(lock): // spin...
  while(test_and_set(lock) == busy);

spinlock_unlock(lock):
  lock = free;
```

Let's see how we can fix the problem with the previous implementation. If the problem is that all of the CPUs are repeatedly spinning on the atomic operation, let's try to separate the test part, which is checking the value of the lock from the atomic. The intuition is that for the testing we can potentially use our caches and test the cached copy of the lock and only if the cached copy of the lock is indicating that the lock has changed it's value, that it's free, only then do we try to actually execute the atomic operation.

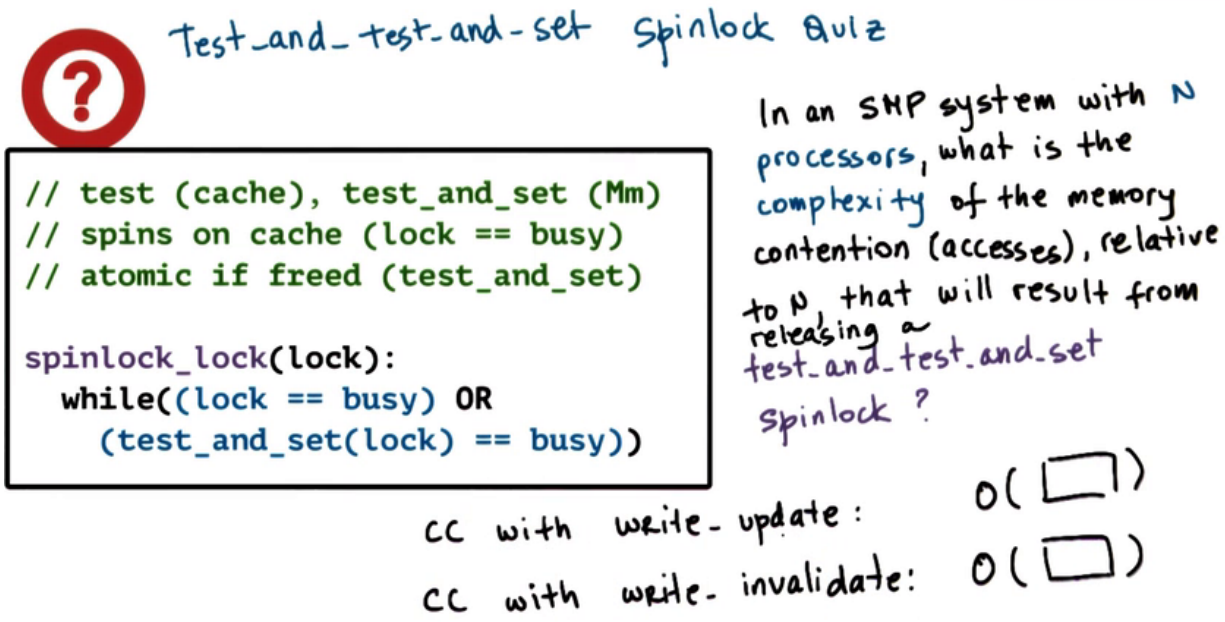


NCC - Non-Cache Coherence
CC-WU - Cache Coherence with Write-Update
CC-WI - Cache Coherence with Write-Invalidate

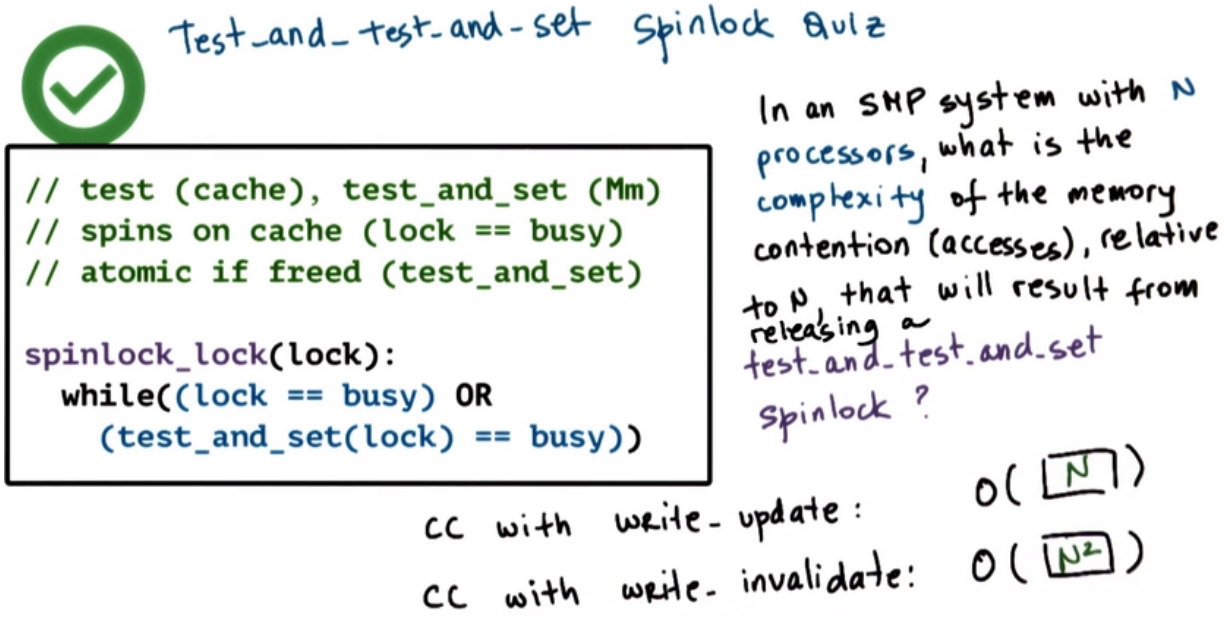
So, here is what the resulting spinlock_lock operation will look like. The first part checks if the lock is busy. This checking is performed in the cached value, so this is the not involving any atomic operation, we're just checking whether particular memory address is set to 1 or 0, so it's busy or free. On a system with caches, this will clearly hit the cache. As long as the lock is busy, we will stay in this while loop, we will not evaluate the second part of this predicate, so if the lock is busy, this is one, this is true, the entire predicate is true so it will go back in the while loop. What that also means is that as long as the lock is busy, as long as this part is true, the test_and_set, the atomic operation will not be evaluated at all, we will not attempt to execute it. Now only when the lock becomes free, so when this part of the predicate, lock==busy, when this is not true, only then do we try to evaluate the second part of this predicate. At that part, the test_and_set operation, the atomic instruction will be executed, or will be attempted at least and then we will see what happens, whether we acquire the lock or not. So what this also means is we will try to make a memory reference since the test_and_set performs a memory reference only when the lock becomes free. This spinlock is referred to as the test_and_test_and_set spinlock. It is also referenced as "spin on read" spinlock since we're spinning on the read value that's in cache or "spin on the cached value" because this is the behavior of the lock. The Anderson paper uses the "spin on read" term to refer to this lock. From a latency and delay standpoint this lock is ok. It's slightly worse than the test_and_set lock because we have to perform this extra check, whether the lock is busy or not that hits the cache, but in principle it's good. But if we start thinking whether or not we really solved the contention problem by

allowing to spin on the cache, we realize that this not quite the case. First if we don't have a cache coherent architecture then every single memory reference will go to memory just like with test_and_set so there's no difference whatsoever. If we have cache coherence with write_update then it's not too bad. The one problem is that all of the processors with write_update will see that the lock becomes freed, so regarding the delay, and so every single one of them will at the same time try to execute the test_and_set operation. So that can potentially be an issue. Now the worst situation of using this particular spinlock implementation is when we have a write_invalidate based architecture. In this case, every single attempt to acquire the lock, not only will it generate contention for the memory module, but it will also create invalidation traffic. Now when we talked about the atomics, we said that one outcome of executing an atomic instruction is that we will trigger the cache coherence, so the write_update or write_invalidate traffic regardless of what the situation is. If we have a write_update situation, that coherence traffic will update the value of the other caches with the new value of the lock. If the lock was busy before the write_update event and if the lock remains busy after the write_update event, great no change whatsoever. That particular CPU can continue spinning on the cached copy. However with the write_invalidate we will simply invalidate the cached copy so it is possible that the lock was busy before the cached coherence event, before somebody executed an atomic instruction, and if the atomic instruction was not successful we're basically continuing to have a situation in which the lock is busy. However, as far as the caches in the system are concerned, that atomic instruction just invalidated their cached copy of the lock. The outcome of that is, they will have to go out to memory in order to fetch this copy of lock that they want to be spinning on, so they will not be able to just spin on the cached copy. They will have to go ahead and fetch this lock value from memory every single time somebody attempts to perform an atomic instruction. So this type of behavior will simply just compound the contention effects and will make performance worse. The reason basically for the poor performance of this lock is that at the same time everybody will see that the lock has changed its state, so that it has been freed, and everybody at the same time will try to acquire this lock.

28 - Test and Test and Set Spinlock Quiz

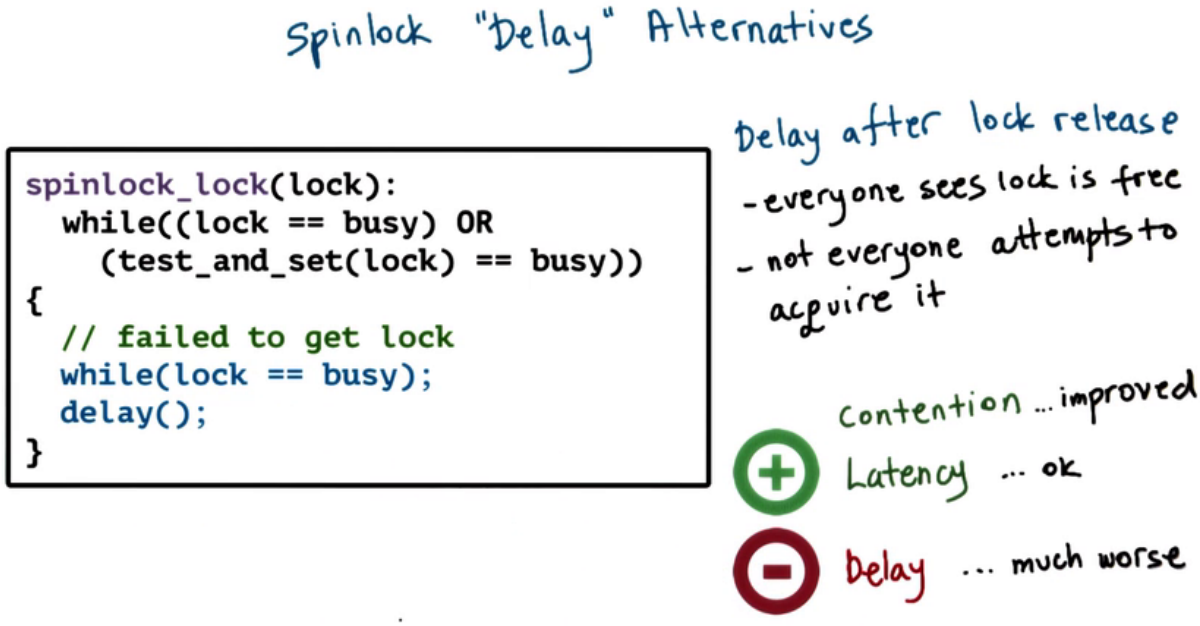


29 - Test and Test and Set Spinlock Quiz

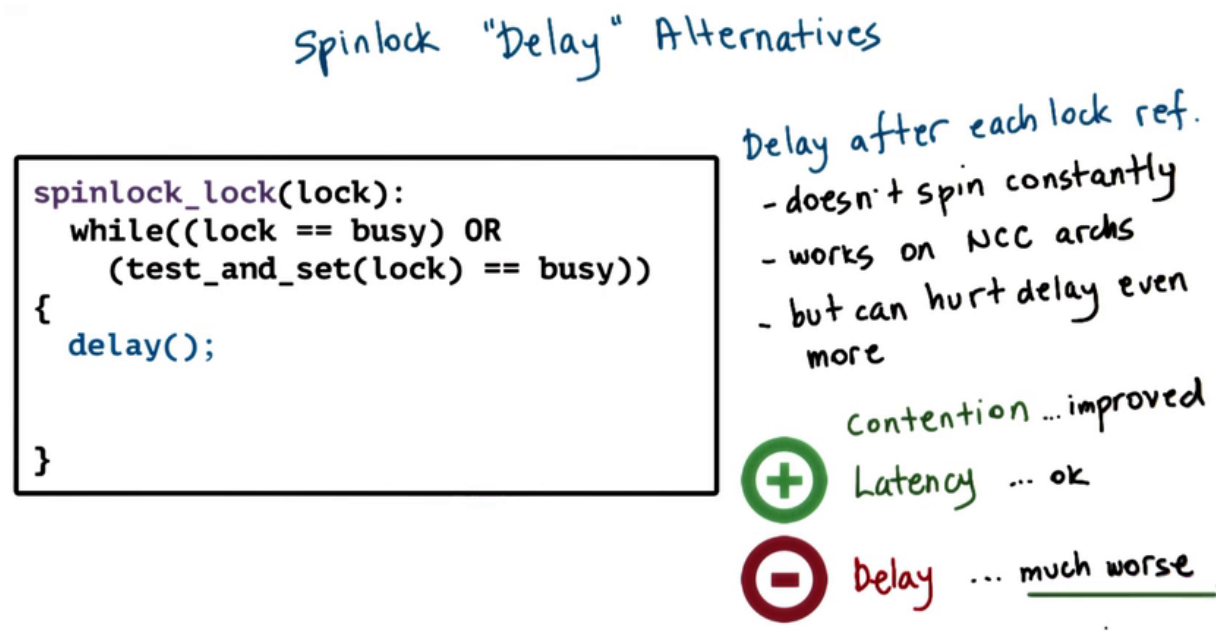


If caches are write_update then all of the processors will be able to see when the lock is released immediately and they will issue a test_and_set operation. So we'll have as many memory references as there will be test_and_set operations so the complexity of the contention is going to be order of all of N. If the caches are write_invalidated then all of the processors caches will be invalidated after that initial lock release. For some processors, by the time they re-read the lock value from memory in order to execute this part of the predicate, the lock will have already been set to busy by another processor, so those processors will try to spin on the newly read cached copy, so back in this portion of the while loop. Other processors however, when they re-read the value of lock from memory, that will happen before any test_and_set has executed so they will see the value of lock is free. As a result, they will try to execute the test_and_set operation. Now only one of these test_and_set operations will succeed, however every single one of them will go ahead and invalidate everybody's caches. That means that that will also invalidate the caches on those processors that did see that the lock was busy. For this reason, the complexity of the bandwidth that gets created, the contention that gets created when the lock is freed using a test_and_test_and_set spinlock is O of N squared.

30 - Spinlock Delay Alternatives

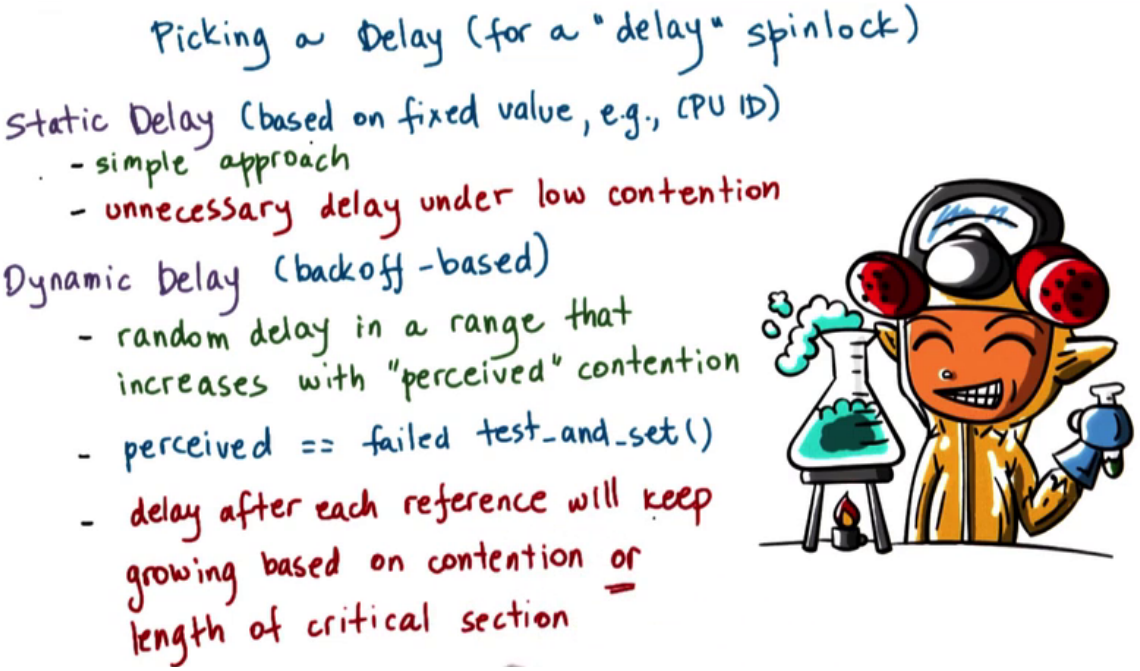


A simple way to deal with the problems of the test_and_test_and_set lock is to introduce a delay. Here is a simple implementation which introduces a delay every single time a thread notices that the lock is freed. So the delay happens after release. When the thread sees that the lock is freed, it will come out of this while loop and then before going back to re-check what the value of the lock is and whether it's indeed freed to execute the atomic operation, the test_and_set, the thread will wait a little bit, will delay. The rationale of this is yes, everybody will see that the lock is free at the same time however with this delay not everybody will try to issue the atomic operation at the same time. As a result, the contention in the system will be significantly improved. When the delay expires, the delayed threads will try to recheck the value of the lock and it's possible that if somebody else in the meantime came and executed the test_and_set, it's possible that they will see that the lock is busy and then they will go in this inner while loop and continue spinning. If the lock is free, the delayed thread will execute the atomic operation however with this delay it's more likely that not all of the threads will try to execute the atomic operation at the same time and also that we're not going to have these situations where threads are repeatedly invalidated while they're attempting to spin on the cached value. That is because after the delay in the second check, a lot of the threads will see that this lock has become busy already and they will not even attempt to execute the test_and_set instruction. So, there will be fewer cases where the lock is busy and somebody's attempting to execute the test_and_set instruction and that was what caused one of the issues with the contention effects in the previous examples. From a latency perspective, this spinlock is still OK. Yes, we do have to perform one memory reference to get the lock first into the cache and then perform the atomic instruction but that's similar to what we saw with the test_and_test_and_set. From a delay perspective, clearly this lock will be much worse because once the lock is freed, then we have to delay for some amount of time and if there is no contention for the lock, then that delay is just wasted time.



Another variant of the delay based spinlocks is to introduce a delay after every single lock reference. So everytime we go through this while loop, we include a delay. The main benefit of this is that it works on non-cache coherent (NCC) architectures. Because basically we're not going to be spinning constantly. In every single spin loop, we will include a delay. So if we don't have cache coherence and we have to go to memory using this delay will help with the reduction of contention of the memory network. The downside of this is clearly that it will hurt the delay much more because we're basically building up the delay even when there is no contention on the network.

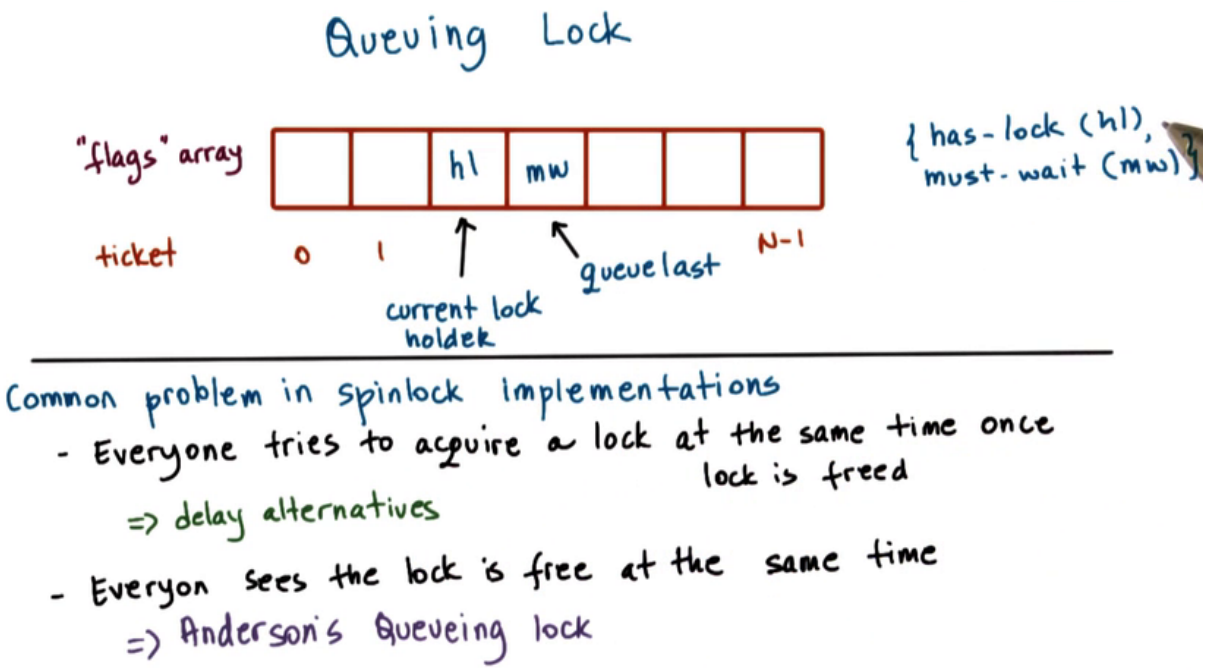
31 - Picking a Delay



One issue with the delay based spinlocks is how to pick a good delay value. Two big strategies make sense. Static delays and a dynamic delay. For a static delay, we can use some fixed information like for instance the CPU id where the process is running in order to determine a delay that will be used for any single process that ends up running on that particular CPU. The benefits of this is that it's a simple approach and under high loads likely this static delays will sort of nicely spread out all of the atomic references so that there's no contention. Clearly the delay will have to be a combination of something that combines this fixed information and the length of the critical section so that one process is delayed one times the critical section, another process is delayed twice the critical section, and so forth. The problem is that this will create unnecessary delay under low contention, so what if we have two processes, the first process is running on CPU with an ID 1, the second process is running on a CPU with an ID 32, and so they're the only two that are contending for the spinlock and yet the second process that's running on the CPU with ID 32 will be delayed substantially regardless of the fact that there's no contention for the lock. To avoid this issue, a more popular approach is to used dynamic delay. And the dynamic delay is such that each process will pick a random delay value based on the current perception of the contention in the system. The idea is that if the system is operating in a mode of low contention then it will choose a dynamic delay value that's within a smaller range, so it will back off for just a little bit, it will delay for just a little bit, and if the system is opening in a mode of large contention, then this delay range will be much larger so with the random selection some processes will back off a little bit whereas other processes will back off, they will delay for quite a bit of time. Theoretically, both of these approaches under high load will result into the same kind of behavior, so dynamic will at high load, will tend to be equivalent to the static delay based approach. The key question is however, in this dynamic delay is, how do we know how much contention is there in the system? That's why we put here the term

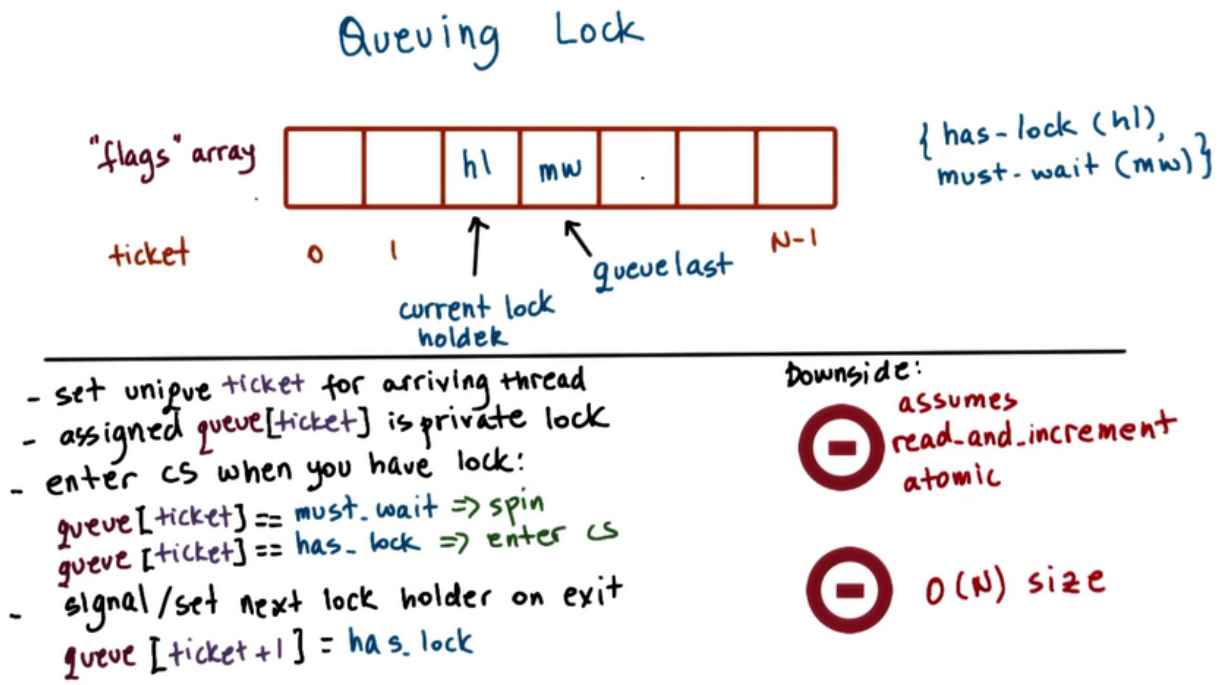
“perceived”. We don’t know exactly what is the contention in the system so each of the processes, the implementation of the spinlock somehow has to infer whether the system is operating in low or high contention so that it can pick an appropriate delay. So a good metric to estimate the contention is to track the number of failed test_and_set operations. If a test_and_set operation fails, the more likely it is that there is a higher degree of contention. The problem with this however is if we’re dealing with after every single lock reference then our delay will keep growing based on both whether there is indeed contention in the system or if simply the owner of the critical section is delayed, it’s executing a long critical section. So then we may end up with the same situation as before. Just because somebody was executing a long critical section while holding the spinlock, that doesn’t mean that we need to bump up the delay. So we need to be able to guard against this case.

32 - Queueing Lock



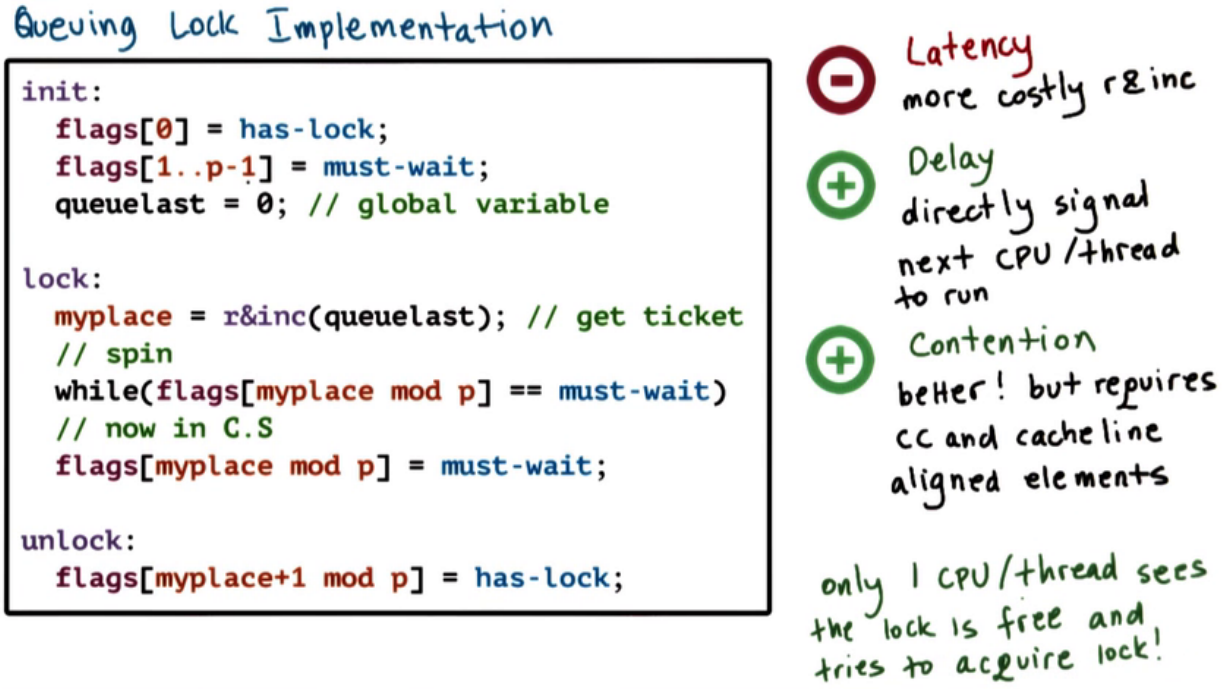
The delay alternatives in the spinlock implementations that we saw in the last morsel address the problem that everybody tries to acquire a spinlock at the same time when that lock is freed. In this paper, Anderson proposes a new lock called a queueing lock and that lock is trying to solve the problem that everybody sees that the lock is free at the same time. If we solve this time, if not everybody sees that the lock is freed, then we’re essentially also solving the second problem because then not everybody will try to acquire the lock at the same time. So let’s take a look at what this lock looks like.

The queueing lock looks as follows. It uses an array of flags having one of two values. Either has_lock or must_wait. In addition we will have two pointers, they will indicate the current lock holder, so that one will clearly have a value of has_lock.



And it will also indicate the index into this array that has the last element on the queue. When a new thread arrives at the lock, it will get a ticket and that will be the current position of the thread in the lock. This will be done by adding it after the existing last element in the queue. So basically the *queuelast* value will be incremented and the new thread will be assigned the next position in the array. Since multiple threads may be arriving at the lock at the same time, we will clearly have to make sure that the way that this *queuelast* pointer is incremented is done atomically. So basically this queueing lock depends on some support for an atomic read_and_increment, it will rely on that kind of hardware atomic operation. This is not as common as test_and_set that we used in the previous spinlock implementations. For each of the threads that are arriving at this queueing spinlock, the assigned element of this flags array, a queue of ticket, that acts like a private lock. What that means is that as long as queue[ticket] is must_wait, then the thread will have to spin just like with a spinlock. When the value of this element becomes has_lock, that will be an indication that the lock is free and you can go ahead and proceed and enter the critical section. When a thread completes the critical section and needs to release the lock, it needs to signal the next thread in this queueing array that it currently has the lock. So these are the steps that control who gets to execute and what needs to happen when a critical section is complete and a lock needs to be released. Other than the fact that this lock requires some support for some read_and_increment to be performed atomically, clearly it's going to require a size that's much larger than the other spinlock implementations. All other locks we saw needed only 1 memory location to keep the spinlock location whether the spinlock is free or busy. And here we need N such locations to keep the values has_lock or must_wait for each of the elements in this array.

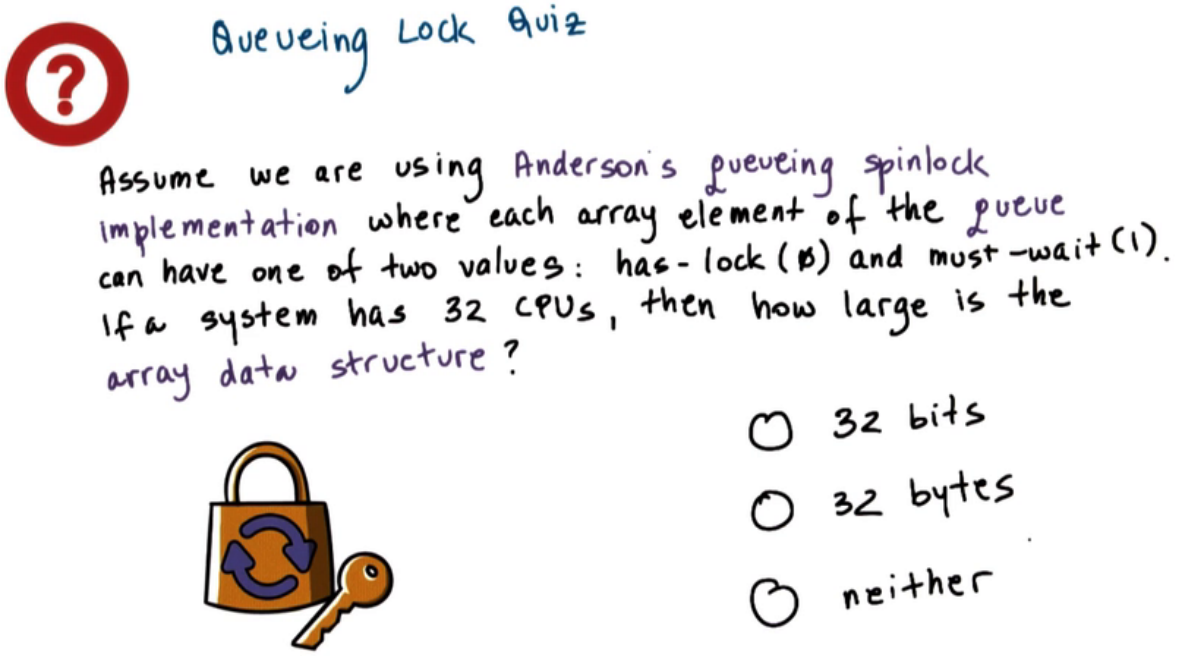
33 - Queueing Lock Implementation



Here is the implementation the queueing lock or the so called Anderson lock. The lock is an array and the array elements have values has_lock or must_wait. Initially, the first element of the array has the value has_lock and all the other elements have the value must_wait. Also, part of the lock structure is a global variable queuelast. To obtain a ticket into this queueing lock, a process must first perform a read_and_increment atomic operation. That will return the myplace, so the index into that particular process into the queue. The process will then continue to spin on the corresponding element of the array as long as its value is must_wait. For the very first thread that arrives at this lock, the value of flags(0) will be has_lock so that one will successfully acquire the lock and proceed in the critical section. Every subsequent thread, as long as the first thread is in the critical section, will come and get tickets that will point to some elements of the flags array that are from 0 to p-1 and they will not be able to proceed. Their value will be must_wait. When the owner of the lock is done with the critical section, it will reset its element in the array to must_wait in order to get this ready for the next threads that, next processes that tries to acquire this lock. Notice that we're using some modular math in order to wrap around this index and read_and_increment will just continue increasing the value of queuelast whereas our array is of limited size. Releasing the lock means that we need to change the value of the next element in the array. So, myplace + 1 we will change it's value from must_wait ot has_lock. That means that that thread that process will now come out of the spin loop that it's in. Notice that the atomic in this spinlock implementation involves a read_and_increment on a variable queuelast. All of the spinning in this implementation happens on completely different variables, so the elements of the flags array. For this reason, issuing the atomic operation read_and_increment any kind of invalidation traffic is not going to concern any of the spinning that's happening on the elements of the flags array. These are two different memory locations, two different variables. From a latency perspective, this lock is not

very efficient because it performs a more complex atomic operation, the read_and_increment. This read_and_increment operation takes more cycles than an atomic test_and_set. In addition it needs to perform this modular shift in order to find the right index into the array and all of that needs to happen before it can determine whether or not it has to spin or be in the critical section. So the latency is not good. The delay is really good however when a lock is freed the next processor to run is directly signaled by changing the value of its flag. Since we're spinning on different locations we can afford to spin all the time and therefore we can notice that the value has changed immediately. From a contention perspective, this lock is much better than any of the other alternatives that we discussed since the atomic is only executed once upfront and it's not part of the spinning code. Plus the atomic instructions and the spinning are done on different variables so the invalidations that are triggered by the atomic will not affect the processor's ability to spin on local caches. However in order for us to achieve this, we have to make sure that first we have a cache coherent architecture. If we don't have cache coherent architecture, the spinning has to happen on potentially remote memory reference. Second, we also have to make sure that every element is in a separate cache line. Otherwise, when we change the value of one element of the array we will invalidate the entire cache line so we will invalidate potentially the caches of the other elements in the array of the processors that are spinning on other elements. And that's clearly not something we want to achieve. To summarize, the benefits of this lock come from the fact that it addresses the key problem that we mentioned with the other spinlock implementations. In that everyone saw that the lock was free at the same time and everyone tried to acquire the lock at the same time. The queuing lock solves that problem. By having a separate, essentially a private lock in this array of locks, only 1 thread at at time sees that the lock is free and only 1 thread at a time attempts to acquire the lock.

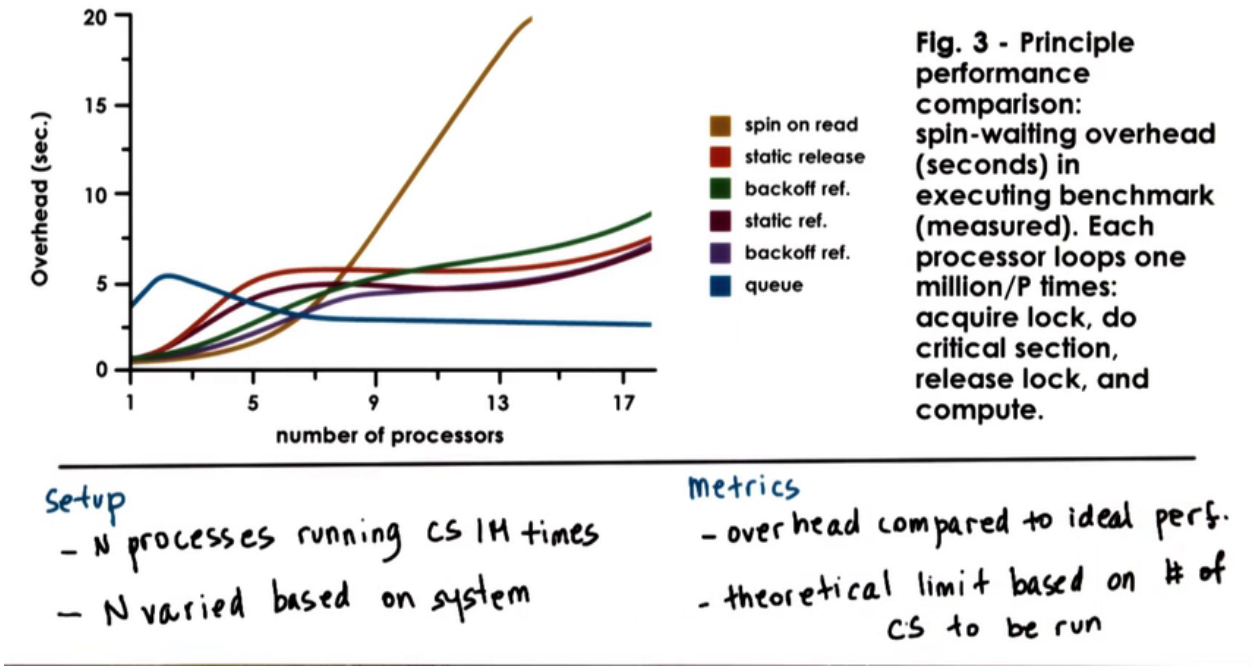
34 - Queueing Lock Array Quiz



35 - Queueing Lock Array Quiz

The correct answer is neither. Remember that for the queueing implementation to work correctly, each of the elements has to be in a different cache line. So the size of the data structure depends on the size of the cache line. For example, on my machine the cache line is 64 bytes, so the size of the data structure would be 32 x 64 bytes but in practice there may be other cache line sizes that are used on the architectures that you use.

36 - Spinlock Performance Comparisons



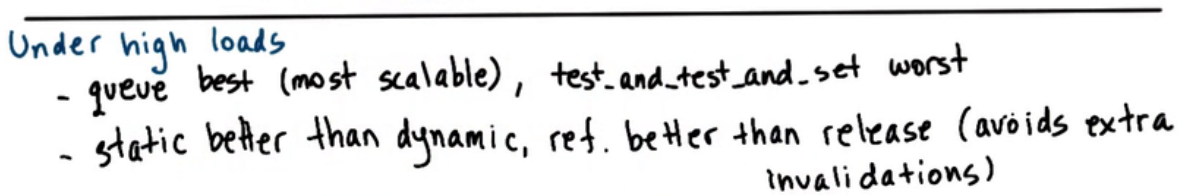
pan

Instructor Notes

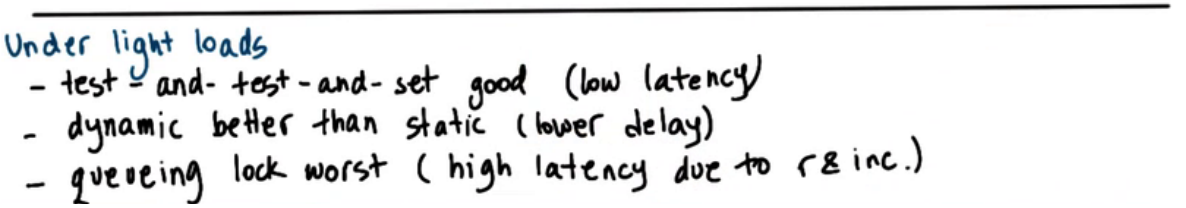
- Anderson, Thomas E.
- "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors".
- IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990.

Finally, let's take a look at one of the results from the performance measurements that are show in Anderson's paper. This is figure 3 from that paper if you're following along. This figure shows measurements that were gathered from executing a program with multiple processes and each process executed a critical section, the critical section was executed in a loop 1 million times. The number of processes in the system was varied such that there is only 1 process per processor and the platform that was used was a Sequent Symmetry Model B that had 20 processors. So that's why the maximum number of processes was also kept to 20. Also this process was cache coherent with write_invalidate. The metric that was computed based on the experiments was the overhead that was measured compared to a case of ideal performance. An ideal performance corresponded in these measurements to a theoretical limit how long it takes to execute that fixed number of critical sections. So basically there is no contention, no effect to the fact that each of these critical sections needs to be locked and unlocked and how long would it take to run these critical sections. The measured difference between that theoretical limit and whatever it actually took to perform this experiment was considered the overhead and this what these graphs represents. These experiments were performed for every one of the different spinlock implementations that we discussed. From the spin_on_read, the test_and_test_and_set all the way to the queueing implementation. These results don't include the basic test_and_set when we're spinning on the atomic instruction, basically that

implementation would result in something that would just completely be off the chart, it would be the highest overhead, the worst performance measurements. And then notice that we have for the different variations of the delay, both the static and the dynamic delay. So let's see what happens under high load.



This is where we have lots of processors and lots of processes running on those processors that are contending for the lock. We see from these measurements that the queueing lock, it's the best one. It is the most scalable and as we add more and more processors, more and more load, it performs best. The test_and_test_and_set lock that one will perform worst. That is in particular the case because here we have an architecture that's cache coherent with write_invalidate and we said that in that case on release of the test_and_test_and_set lock we have order of N squared memory references so a very high contention on the shared bus and that's going to really hurt performance. Of the delay based alternatives, we see that the static implementations are a little bit better than their dynamic counterparts since under high loads with static we end up nicely balancing out the atomic instructions whereas with dynamic we still end up with some randomness, with some number of collisions that are avoided in the static case. Also note that delaying after every single memory reference is slightly better than delaying after the lock is freed only, so only on release, because when we avoid after every reference we end up avoiding some additional invalidations that come from the fact that Sequent is a write_invalidate architecture.



Under light loads when we have few processes and few processors as well, we can make a couple of observations. First we see that test_and_set performs really pretty well in this case and that's because this implementation of spinlock has low latency. We are just performing a check, if lock == busy, and then we moving onto the atomic test_and_set. We also see that in terms of the delay alternatives, the dynamic delay alternatives, the back off delay alternatives perform better than the static ones. And that is because with the dynamic alternatives, we end up with lower delays. We said that static alternatives can lead to situations in which the two processors that have the extreme, the smallest and the largest delay, are the only ones contending on the lock and so we have wasted cycles. We also see that under light loads, the queueing locks performs the worst, this is the performance of the queueing lock, and the reason for that is we explained that with the queueing lock we had pretty high latency because we need to implement the read_and_increment, and the modular processing, etc. So this is what hurts the performance of the queueing lock under light loads. One final comment regarding the

performance results that we just discussed that this points to the fact that with system design there isn't really a single good answer that the design points should be driven based on the expected workload, light loads, high loads, architectural features, numbers of processors, write_invalidate, write_update, etc. The paper includes additional results that point to some of these trade offs in more detail.

37 - Lesson Summary

In this lesson we talked about other synchronization constructs beyond just mutexes and condition variables and described some of the synchronization problems that these constructs are well suited for. In addition, we talked about the spinlock alternatives that are described in Anderson's paper and learned how hardware support, specifically how atomic instructions are used when implementing constructs like spinlocks.