

Lesson Preview

Synchronization

- more **synchronization constructs**
- hardware supported **synchronization**

1. 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 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 available in the instructors notes.

Visual Metaphor

"Synchronization is like ... waiting for a coworker to finish so you can continue working"

- May repeatedly check to continue
 - sync using spinlocks

- May wait for a signal to continue
 - sync using mutexes and condition variables

- Waiting hurts performance
 - CPUs waste cycles for checking; cache effects

- May repeatedly check to continue
 - Are you done? Still working?

- May wait for a signal to continue
 - Hey, I'm done!

- Waiting hurts performance
 - Workers waste productive time while waiting



1:47 / 2:04



YouTube



2. Let's illustrate synchronization and synchronization mechanisms using our toy shop example, synchronization in operating systems is like waiting for a co-worker 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 co-worker to signal them once they are 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 a 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 come and do your work now is maybe a little bit more laid back but this other guy may have gone out for lunch in the meantime so it may take it longer to actually come back and precede 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 okay 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 are signalling another process that was periodically blocked to come back and start executing again.

More About Synchronization?



Mutexes



Condition Variables

Why more?



3. We spend already quite a bit of time talking about dealing with concurrency in multi-threaded 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 all safe and well underway 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.

More About Synchronization?

Limitation of mutexes and condition variables

- ⊖ error prone / correctness / ease-of-use
 - unlock wrong mutex, signal wrong condition variable
- ⊖ lack of expressive power
 - helper variables for access or priority control

Low level support

- hardware atomic instructions

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 signalling 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 of 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 a reader's writer 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 a 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. So 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 views the underlying atomic instructions, to achieve efficient implementations of certain synchronization constructs.



Spinlocks (basic sync construct)

Spinlock is like a mutex

- mutual exclusion
- lock and unlock (free)

BUT

- lock == busy

=> SPINNING!!!

```
spinlock_lock(s);  
    // critical section  
spinlock_unlock(s);
```

4. One of the most basic synchronization constructs that's commonly supported in an operating system is **spinlock**. In some ways, spinlocks are similar to a mutex. The lock is used to protect the 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(放棄) 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 its time slice expired, or potentially a higher priority thread became runnable. Because of their simplicities, 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.



Semaphores

- common sync construct in OS kernels
- like a traffic light : **STOP** and **GO**
- similar to a mutex ... but more general

5. The next construct we will talk about is semaphores, and these are a common synchronization constructs that have been part of operating system kernels for a while. As a first approximation, a semaphore acts like 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 ways similar to what we saw in, with the mutex. It either allows a thread to obtain the 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 with a mutex, and we will take a look at how, what exactly a semaphore is a bit more formally next.



Semaphores designed by E.W. Dijkstra

semaphore == integer value \Rightarrow count-based sync

on init

- assigned a max value (positive int) \Rightarrow maximum count

on try (wait) P (proberen)

- if non-zero \Rightarrow decrement and proceed \Rightarrow counting semaphore

if initialized with 1

- semaphore == mutex (binary semaphore)

on exit (post) V (verhogen)

- increment

Semaphores are represented via an integer number, some positive integer value. When they're initialized, they're assigned some maximum value, some positive integer (此數即可以同時執行 critical section 的 threads 個數). Threads arriving at the semaphore will try it out. If the semaphore's value is non-zero, they will decrement it (即表明還可以執行 critical section 的 threads 少了一個) and proceed (即執行 critical

section). **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'll allow us to express count-related synchronization requirements. **For instance, five producers may be able to produce an item at the same time.** The semaphore will be initialized with the number five, and then it will count and make sure that exactly five producers are allowed to proceed. **If a semaphore is initialized with one, 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 or signal the semaphore. What that will do, it will increment the semaphore's counter (即表明可以执行的 critical section 的 threads 又多了一個).** For a binary semaphore, this is equivalent to unlocking a mutex. Finally, some historic tidbits(一口;小欄報導). Semaphores were originally designed by Dijkstra. He was a Dutch computer scientist and also a Turing Award winner. And the wait and post operations that are used with 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.



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);
```

6. Here's some of the operations that are part of the Posix Semaphore API. Semaphore.h defines the sem_t semaphore type. Initializing a semaphore is done with the sem_init code. This takes as a parameter of semaphore type variable and also it takes the initialization count and a flag (即 pshared). 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.

7. As a quick quiz, complete the arguments in the initialization routine for a semaphore. So that the use of the semaphore is identical in its behavior to that of a mutex that is used by threads within a single process. A reference link to the semaphore API has been included in the instructor notes.

[Instructor notes](#) 中的那個網頁中有用的內容:

Name

`sem_init` - initialize an unnamed semaphore

Synopsis

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Link with `-pthread`.

Description

`sem_init()` initializes the unnamed semaphore at the address pointed to by `sem`. The `value` argument specifies the initial value for the semaphore.

The `pshared` argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

If `pshared` has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

If `pshared` is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory (see `shm_open(3)`, `mmap(2)`, and `shmget(2)`). (Since a child created by `fork(2)` inherits its parent's memory mappings, it can also access the semaphore.) Any process that can access the shared memory region can operate on the semaphore using `sem_post(3)`, `sem_wait(3)`, etc.

Initializing a semaphore that has already been initialized results in undefined behavior.



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);
```

8. To use a semaphore in this manner, you should initialize the semaphore so that it is a non-process shared semaphore. So, this argument to the initialization call should be 0. And so that its initial count is 1. So this argument will be 1. Then, when the semaphore wait operation is called. It will decrement this counter and it will allow exactly one thread at a time to enter the critical section and posting to a semaphore will increment the counter and it will be identical to a mutex being freed. [Note that most operating systems textbooks will include some examples on how to implement one synchronization construct with another including mutexes or condition variables with semaphores.](#) And there are many other examples. So you're welcome to experiment on your own with these kinds of examples.



Reader/Writer Locks

Sync'ing different types of accesses
read (never modify)
shared access



write (always modify)
exclusive access

RWlocks

- specify the type of access
→ then lock behaves accordingly

9. 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, versus those accesses that do modify a shared resource. 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 run times as well supports 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 always specify the type of access, read versus write that you want to perform. And then underneath the lock behaves accordingly.



Using RW Locks



```
#include <linux/spinlock.h>
rwlock_t m;
read_lock(m);
    // critical section
read_unlock(m);

write_lock(m);
    // critical section
write_unlock(m);
```

10. 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 in 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.



Using RW Locks



- rlock support in Windows (.NET), Java, POSIX ...
- read/write == shared/exclusive

Semantic differences

- recursive read.lock ... → what happens on read.unlock?
- upgrade / downgrade **priority**?
- interaction with scheduling policy
 - e.g., block if higher **priority** writer waiting

Reader writer locks are supported in many operating systems and language run times. 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 a recursive read_lock operations to be invoked (意思可能是上一層 read_lock invoke 下一層 read_lock, think of function call) . But then, it differs across implementations, and exactly what happens on read_unlock. In some cases, a single 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, so an owner of a shared lock, may be given a 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 a read lock will first release it and then try to re-acquire 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 interaction is there between the state of the lock, the priority of the threads, and the scheduling policy in the system overall. For instance (如果有 higher priority writer waiting), 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 mechanisms, 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.

Monitors

Monitors specify...

- shared resource
- entry procedure
- possible condition variables

on entry

- lock, check ...

on exit

- unlock, check, signal...



11. 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 peer-wise 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 the necessary locking and checking will take place when the thread is entering the monitor. Similarly, when the 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.

Monitors

Monitors == high-level synchronization construct

- MESA by XEROX PARC
- Java
 - synchronized methods generate monitor code
 - notify () explicitly

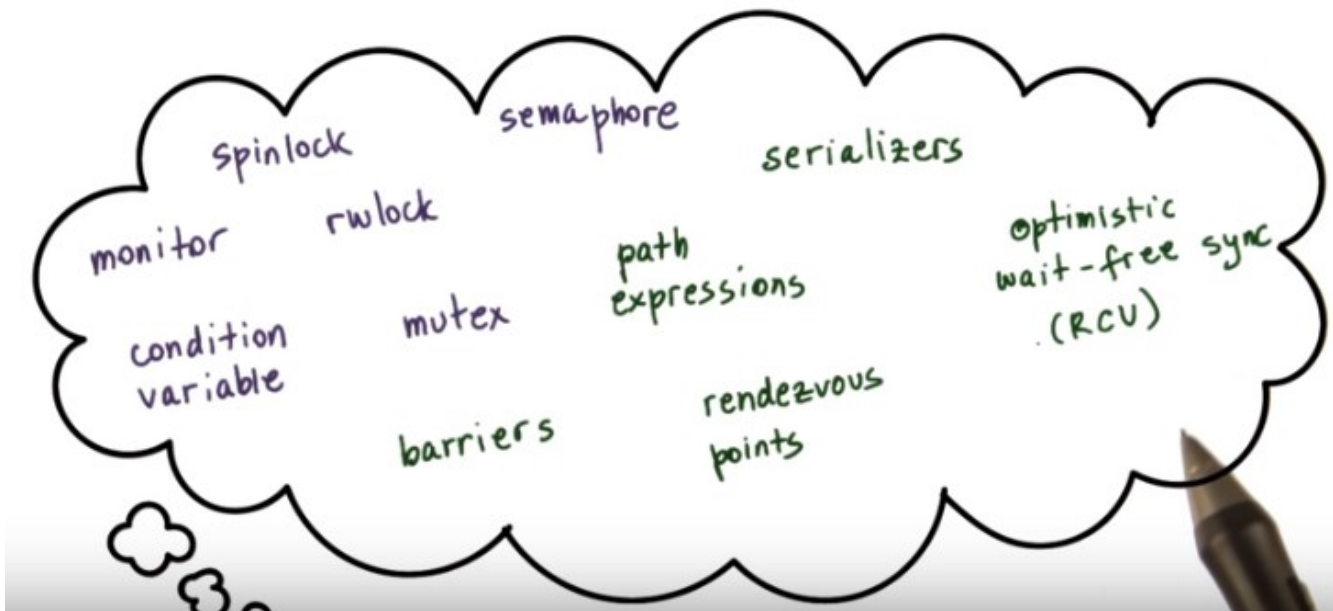


Monitors == programming style

- enter_ /exit_ Critical Section
- in Threads and Concurrency lesson

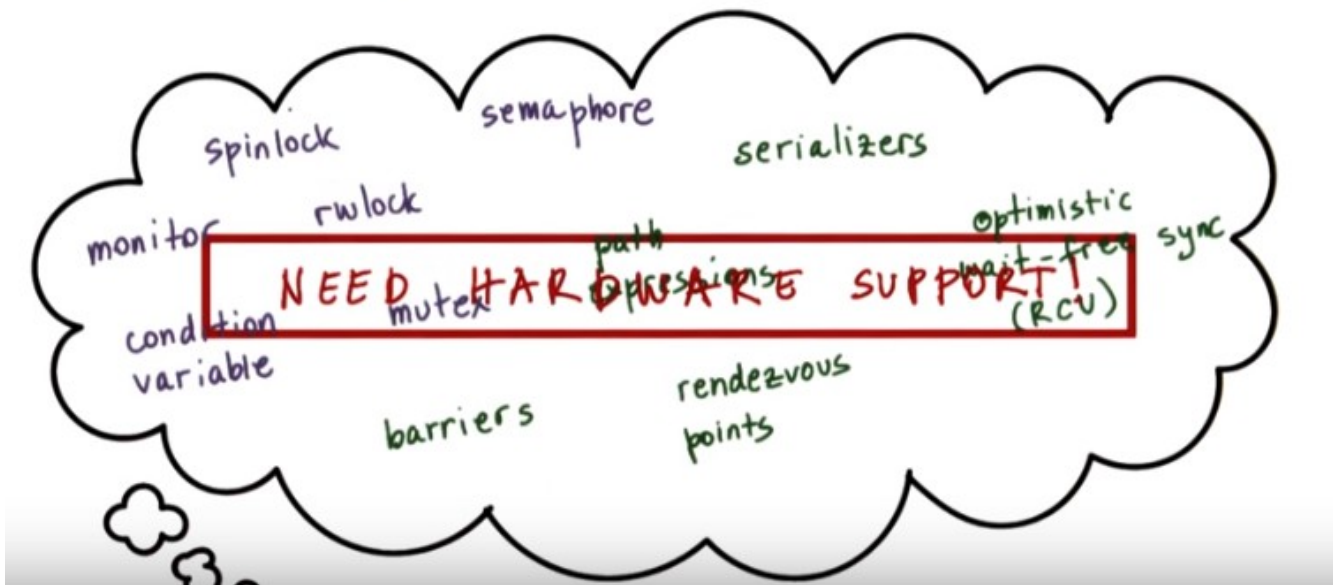
Because of all of this, monitors are referred to as a high-level synchronization construct. Historically, monitors were included in the MESA language runtime developed by Xerox PARC. Today, Java supports monitors too. Every single object in Java has an internal lock, and methods that are declared to be synchronized are entry points into this monitor. When complied, 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 is what we described in the threads and concurrency lesson with the enter critical section and exit critical section section sub code(應該就是分成幾大塊那個).

More Synchronization Constructs



12. 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 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. Another useful construct are [barriers](#), and these are almost like a reverse from a semaphore in that if a semaphore will allow n 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 in 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 this so-called read-copy update log, RCU log, that's part of the Linux kernel. 這一種的意思應該是專門把程序寫成不會出現 concurrency 的樣子。

More Synchronization Constructs



'後面的兩個 spinlock quiz 及其後的一段' 說明直接從軟件上不能實現 正確 且 有效率 的 spinlock, 所以我們需要硬件支持. 且說明下面的 atomically 是對的, 不是 automatically.

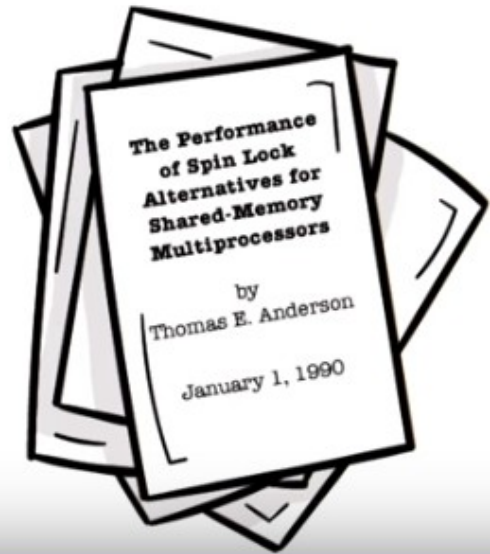
One thing that all of these methods have in common is that at the lowest level, they all require some support from the underlying hardware to atomically make updates to a memory location. This is the only way they can actually guarantee that the lock is properly required, and that the state change is performed in a way that is safe. And that it 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 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.

Spinlocks Revisited

spinlock => basic sync construct

"The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors" by Anderson

- alternative implementations of spinlocks
- generalize techniques to other constructs



13. We said that spinlocks are one of the most basic synchronization primitives. And that they're 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 on Shared Memory Multiprocessors by Tom Anderson. The paper discusses different implementations of spinlocks and this is relevant also for other synchronization constructs that use internally spinlocks. 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. As a quiz, let's look at a possible spinlock implementation. Here is a possible pseudo code. To be clear, the lock needs to be initialized as free and that will be 0. And 1 will indicate that the lock is busy. To lock the spinlock, we first need to check to make sure that the lock is free. And if so, then we can change its state and block it. So change its state to busy. Otherwise, if the lock isn't free, we have to keep spinning. So we have to keep repeating this check and this operation. Finally, releasing the lock means that we set the value of the lock to free. These steps are listed in the Instructor Notes. The questions in this quiz are does the implementation of the lock correctly guarantee mutual exclusion? Also, is this implementation efficient? Mark your answers here.

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

'以下的兩個 spinlock quiz 及其後的一段' 說明直接從軟件上不能實現 正確 且 有效率 的 spinlock, 所以我們需要硬件支持:

Spinlock Quiz 1

Does this spinlock
implementation
correctly guarantee
mutual exclusion?
Is it efficient?

mutual exclusion: ☐ CORRECT
☒ INCORRECT

efficiency: ☐ EFFICIENT
☒ INEFFICIENT

```
spinlock_init(lock):  
    lock = free; // 0 = free; 1 = busy  
  
spinlock_lock(lock):  
    spin:  
        if(lock == free) { lock = busy; }  
        else { goto spin; }  
  
spinlock_unlock(lock):  
    lock = free;
```

lock 為 free 的意思就是 還沒有誰 acquire 了這個 lock 的, lock 為 busy 的意思就是 已經有別的 thread acquire 了這個 lock 了.

15. 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 lock is free. And they will move on to perform this lock equal busy operation at the same time. Only one thread will actually set the lock value to busy correctly. The other one will simply override it and will then proceed. It will think that it has correctly acquired the lock. So as a result, both processes or both threads can end up in the critical section and that clearly is incorrect.

16. Here's another slightly different version of the same implementation that avoids the go to 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 a spinlock correct in terms of its ability to guarantee mutual exclusion, and also, is this an efficient implementation?



Spinlock Quiz 2

Does this spinlock
implementation
correctly guarantee
mutual exclusion?
Is it efficient?

mutual exclusion: ☐ CORRECT
☒ INCORRECT

efficiency: ☐ EFFICIENT
☒ INEFFICIENT

```
spinlock_init(lock):  
    lock = free; // 0 = free; 1 = busy  
  
spinlock_lock(lock):  
    while (lock == busy); // spin  
    lock = busy;  
  
spinlock_unlock(lock):  
    lock = free;
```

17. 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's absolutely no way purely in software to guarantee that there won't be some interleaving of exactly how these threads perform this check in 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 spinlock. 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.

Need for Hardware Support

```
spinlock_lock(lock):  
    while (lock == busy);  
    // spin  
    lock = busy;
```

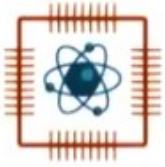
PROBLEM:

- concurrent check/update
on different CPUs can
overlap

=> hardware-supported atomic instructions

本段講得好

18. So we need to get some help from the hardware, looking at this spinlock example from the previous video. We somehow needed the checking of the lock value and the setting of the lock value to happen indivisibly, atomically, so that we can guarantee that only one thread at a time can successfully obtain the lock. The problem with this implementation is that it takes multiple cycles to perform the check in this 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



Atomic Instructions

=> atomic instructions == critical section
with hardware-supported synchronization
specialize/optimize to available atomics

Hardware-specific

- test-and-set
- read-and-increment
- compare-and-swap

Guarantees

- atomicity
- mutual exclusion
- queue all concurrent instructions but one

```
spinlock_lock(lock): // spin until free
while(test_and_set(lock) == busy);
```

test_and_set(lock)

- atomically returns (tests) original value and sets new value = 1 (busy)
- first thread: test_and_set(lock) => 0: free
- next ones: test_and_set(lock) => 1: busy
- reset lock to 1 (busy), but that's ok

上圖框中的例子: 函數 test_and_set(lock) 的作用為: 將 lock 的值設為 1, 並返回 lock 的舊值, 即:
若 lock = 0, 則 test_and_set(lock) = 0, 且此 test_and_set 執行後, lock = 1;
若 lock = 1, 則 test_and_set(lock) = 1, 且此 test_and_set 執行後, lock = 1;
且注意 test_and_set 函數是 mutual exclusion 的, 即只能有一個 thread 在執行此函數, 這是硬件實現的。

上例中, 0 即 free, 1 即 busy.

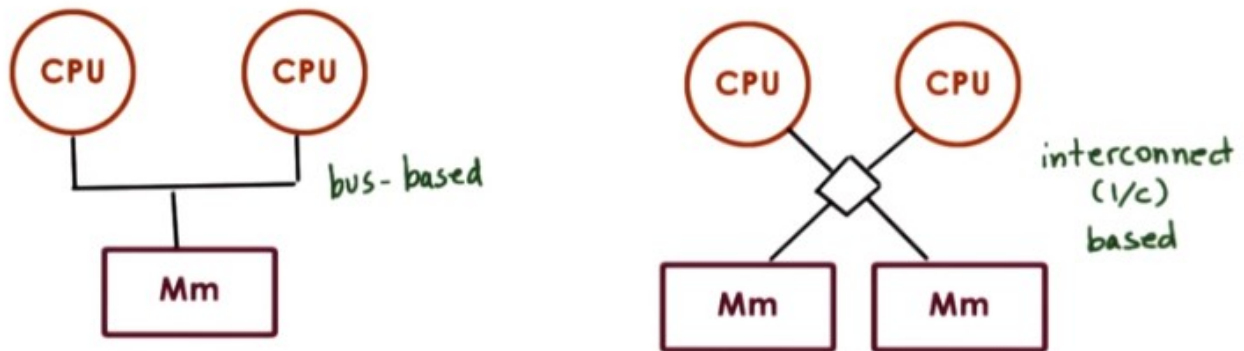
若最初 lock = 1, 則 test_and_set(lock) = 1, 故繼續循環, 且此時 lock 一直為 1, 執行不了後面的 critical section.

若最初 lock = 0, 則 test_and_set(lock) = 0, 故跳出循環, 執行後面的 critical section, 且跳出循環後 lock = 1, 即其它 thread 不能執行 critical section, 相當於此 thread acquire 了 lock.

19. 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 multi-step, multi-cycle operation. But, because they're atomic instructions, the hardware makes guarantees that the set of operations that are included in these instructions will happen atomically, so not just halfway, the entire operation or none of it (這就是 atomically 的意思). That it will happen in mutual exclusion, meaning that only one 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 you look at our spin lock example, using the first atomic test and set up operation, the spin lock implementation can look as follows. Here test is set to automatically returns or tests the original value of the memory locations that's past this parameter, lock in this case. And sets the new value of this memory location to be one. This happens automatically. Remember, one, to us, indicates that the lock is

busy. When we have multiple threads that are contending for this lock, when they are trying to execute this spinlock operation, only one 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 zero because the original value of the lock was zero. Originally, the lock was free. That thread will therefore exit the Y loop because test_and_set will return zero, read, and that is different than busy. Why? 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, or then test_and_set will return one, because this first thread already set the value of lock to be one. Therefore, those remaining threads will just continue spinning into this wild loop. Notice that in the process, these other threads, they're repeatedly resetting the value of the lock to one. 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 lock field to be one again and to be busy. However, that's okay. The very first thread, when it executed test_and_set, they 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, others like read and increment may not be available on all platforms. [And in fact we may have versions of this](#), where in some cases there is availability of an atomic operation that atomically increments something that does not necessarily return the old value. In other cases, there may be atomics that support read_and_decrement as opposed to read_and_increment. [In addition, there may be differences in efficiencies with which different atomic operations execute on different architectures.](#) For this reason, software such as synchronization constructs that are built using certain atomic constructions has to be [ported](#). We have to make sure that the implementation actually uses one of the atomic constructions that's available on the target platform. Also we have to make sure that the implementation of software of these synchronizations constructs is [optimized](#) so that use the most efficient atomicity on our target platform. And so that it uses them in an efficient way to begin with. Anderson's paper presents several alternatives in how spinlocks can be implemented using available hardware atomics, and we will discuss these in the remainder of this lecture.

Shared Memory Multiprocessors



I/C 就是 interconnect 的簡寫

SMPs 即 shared memory multiprocessors, 性質是 the memory is accessible to all CPUs

20. Before moving on with the discussion of the spin lock alternatives that are presented in Anderson's paper, let's do a refresh on multiprocessor systems and our cache coherence mechanisms. This is necessary in order to understand the design trade-offs and the performance trends that will be discussed in this paper. A multiprocessor system consists of more than one CPU. In some memory that is accessible to all of these CPUs. The shared memory may be a single memory component that's equidistant(等距的) from all of the CPUs. Or there will be multiple memory components. Regardless of the number of memory components, they are somehow interconnected 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 base configuration, I can have multiple memory **references** in flight. Where one memory reference(視頻裡指到的右邊的 CPU) is applied to this memory module(右邊的 Mm) and another one(左邊的 CPU) to the other memory module(左邊的 Mm). 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 are spread out across multiple memory modules. So the bus is basically shared across all the modules. Because of this property that the memory's 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.

Shared Memory Multiprocessors and Caches

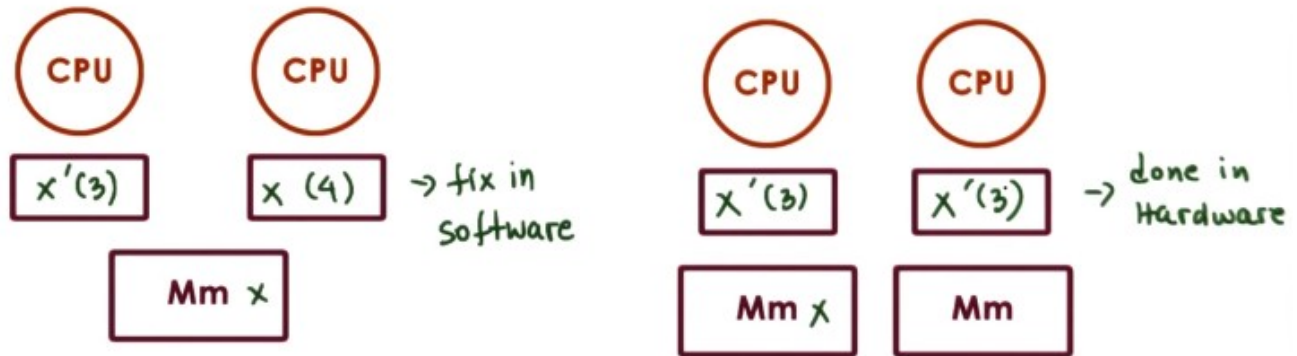


Caches

- hide memory latency ; memory "further away" due to contention
- no write , write-through , write-back

In addition, each of the CPUs in these kinds of systems can have caches. Access to the cached 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(爭用) 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 CPUs because of this contention effect. So when data is present in the cache, the CPUs will read data from the cache instead of memory, and that will have an impact on performance. Now when CPUs perform a write, several things can happen. First (no-write), you 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. 故 no-write 是指 not write cache, 而不是 not write memory. Second (write-through), 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 (write back), 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(驅逐). We call this the write-back.

Cache Coherence



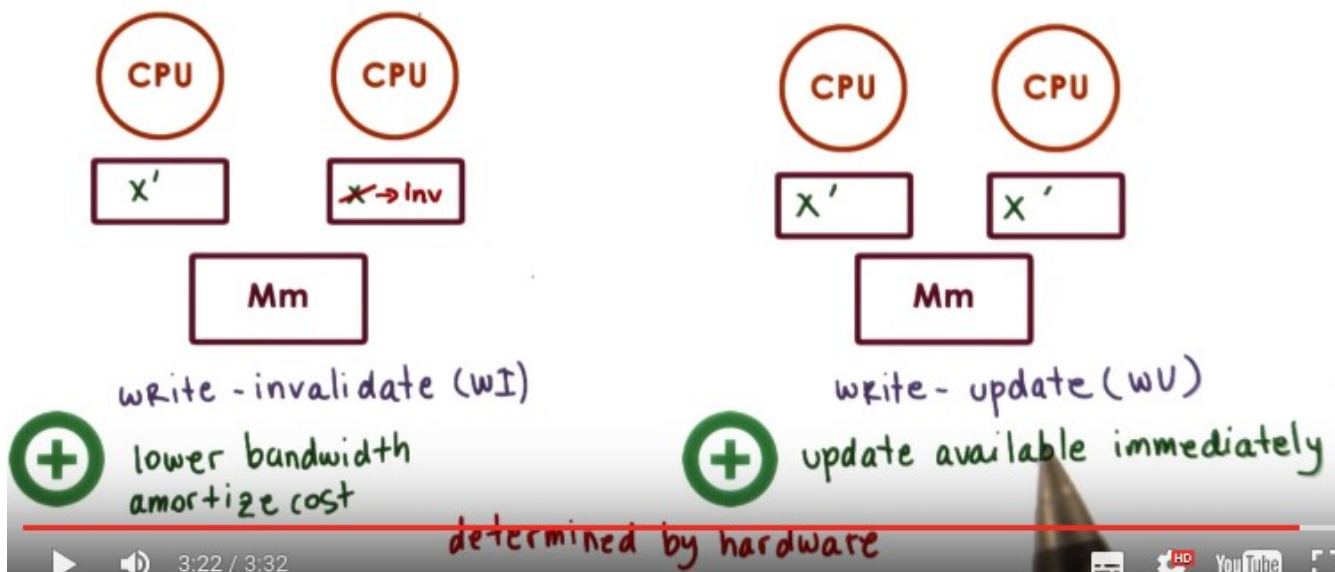
Cache - Coherence

- non-cache-coherent (NCC) vs. cache-coherent (CC)

後面會用到 NCC 和 CC 這兩個簡寫

21. One challenge is, what happens when multiple CPUs reference the same data, x in this case? The data could appear in multiple caches. Or in this case when we have multiple memory modules(右邊的情況), x is actually present in one of the memory modules, but both of the CPUs 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 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 platforms. 下面的 write-invalidate (WI)和 write-update (WU)就是講 hardware 是如何實現 cache-coherent 的.

Cache Coherence

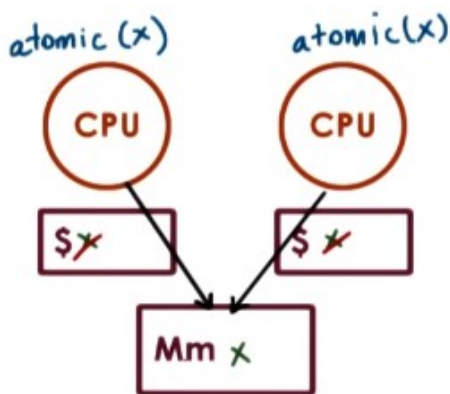


後面會多次用到 write-invalidate (WI)和 write-update (WU)

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 trade off is that with write-invalidate, we actually post lower bandwidth requirements on the shared interconnecting 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 in other caches. Plus once the 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 CPUs anytime soon, it is possible to amortize(分期償還, 緩沖) the cost of the coherence traffic over multiple changes. So 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 CPUs 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 has been updated on another CPU

will benefit from support for a 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.

Cache Coherence and Atomics



Atomics always issued to the memory controller



can be ordered & synchronized



take MUCH LONGER!!!
generates coherence traffic regardless of change

Atomics & SMP

- expensive b/c of bus or I/C contention
- expensive b/c of cache bypass & coherence traffic

22. 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 arbiter interleaving of threads that are sharing the CPU, as well as threads that are concurrently executing across multiple CPUs. Let's consider the following situation. We have two CPUs. 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 the 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 constructions to read and update the cash value of the memory reference, that's the the target of the atomic construction. 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 versus write invalidate protocols, we have latency on the chip. Given all 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 cash value in 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 create a central entry point where all of these references can be ordered and synchronized in a unique matter. So none of the rates conditions that could have occurred if we allowed atomics to access the cache, that 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. We will always have to access memory and they(不同 threads) will also contend on memory. In addition in order to guarantee atomic behavior, we have to generate the coherence traffic and 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 enforcing 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 (SMPs 即 shared memory multiprocessors, 性質是 the memory is accessible to all CPUs), 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 the coherence traffic, regardless to what happens with the memory allocation that's the target of the atomic instruction

Spinlock Performance Metrics

1. Reduce latency
 - "time to acquire a free lock"
 - ideally immediately execute atomic
2. Reduce waiting time (delay)
 - "time to stop spinning and acquire a lock that has been freed"
 - ideally immediately
3. Reduce contention
 - "bus/network I/C traffic"
 - ideally zero



23. With this background on SMPs, cash 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 spinlock to have low latency. By latency, we are 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 whenever it is spinning and a lock becomes free, is we want to reduce that time that 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 the 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 non 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 as 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 a critical section and then release the spinlock. So if we have a contention situation, we may potentially even be delaying the unlock operation for this 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 based on these criteria.

24. As a quiz, let me ask a question regarding the spinlock performance objectives and accompanying performance metrics, which is discussed. Among the performance metrics that we just discussed, are there any conflicting goals? In other words, does one of these goals counteract any of the other ones? I'm giving you three possible answers, that one conflicts with two, that one conflicts with three, and that two conflicts with three. You should check all the ones that apply.



Conflicting metrics Quiz

1. Reduce latency

- "time to acquire free lock"
- ideally, immediately execute atomic

2. Reduce waiting time

- "time to acquire freed lock after waiting"
- ideally, immediately acquire

3. Reduce contention


- "reduce bus/network/I-C traffic (requests)"
- ideally, contention is zero

Among the described metrics are there any **conflicting** goals? check all that apply.

- ☐ 1 conflicts w/ 2
- ☒ 1 conflicts w/ 3
- ☒ 2 conflicts w/ 3

25. The goal number one 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 one conflicts with three is one of the correct answers. Similarly with two, if we want to reduce the waiting time to delay, then we have to make sure we're continuously spinning on the lock as long as it is 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 and so two is conflicting with three. 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 answer as a conflicting one.

Test-and-Set-Spinlock

 Latency
minimal (just atomic)

```
spinlock_init(lock):  
    lock = free; // 0 = free; 1 = busy  
  
spinlock_lock(lock): // spin...  
    while(test_and_set(lock) == busy);  
  
spinlock_unlock(lock):  
    lock = free;
```

$L = \text{free}(\emptyset)$
 $\emptyset = \text{test_and_set}(L)$
 $\hookrightarrow L = \text{busy}(1)$

26. Let's look at this simple spinlock implementation we showed earlier in this lesson that uses the test-and-set instruction. In this one and in 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 0 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 construction that most hardware platforms support it. 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 the lock was originally free. Say it was 0. And then as soon as we execute this spinlock_lock operation, we make an attempt to execute test_and_set. The lock is free, so this will return 0. 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.

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;
```

⊕ Latency
minimal (just atomic)

⊕ Delay
potentially min
(spinning continuously
on the atomic)

L = busy (1)

⌈ test_and_set (1)

Regarding delay, this particular implementation potentially could perform well because we see that we're continuously just spinning on the atomic instruction. As long as the lock is busy, test_and_set will return a 1, will return busy, so we will remain in this while loop. However, whenever the lock does become freed, this test_and_set will, or at least one of them, will immediately detect that and will come out of the while loop. That single successful 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.

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;
```

↖ spinning on
atomic!



Latency
minimal (just atomic)



Delay
potentially min
(spinning continuously
on the atomic)



Contention
processors go to
memory on each
spin

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 they're 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. And 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 construction. 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.

Test_and_test_and_Set Spinlock

```
// test (cache), test_and_set (Mm)
// spins on cache (lock == busy)
// atomic if freed (test_and_set)
```

```
spinlock_lock(lock):
    while((lock == busy) OR
          (test_and_set(lock) == busy))
```

- everyone sees lock is free at the same time
- everyone tries to acquire the lock at the same time

spin on read
spin on cached value



Latency ... ok

Delay ... ok



Contention ... better, but...

- NCC - no difference

- CC-WU - ok

- CC-WI - horrible!

- Contention due to atomics
+ caches Invalidated ==

more contention

NCC: non-cache-coherent

CC-WU: cache-coherence with write-update

CC-WI: cache-coherence and write-invalidate

27. 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 that cached copy of the lock is indicating that the lock has changed its value, that it's cleaned, only then do we try to actually execute the atomic operation. So, here is what the resulting spin lock, lock operation will look like. The first part checks if the lock is busy. This checking is performed in the cache value, so this is the not involving any atomic operation, we're just checking whether a particular memory address is set to one or zero, 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, it 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, two 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, so the atomic operation, will not be a valued at all. We will not attempt to execute it. Now, only when the lock becomes free, so when this part of the predicate lock equals busy, when this is not true. Only then do we try to evaluate the second part of this predicate, at that point the test to 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 spin lock is referred to as the test_and_test_and_set spinlock. It is also reference as a spin and read spinlock, since we're spinning on the read value that's in cache, or spin on the cached values, so because this is the behavior of the lock. The Anderson paper uses the spin and reach term to refer to this lock. >From a latency and delay time point, this lock is okay. 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.

以下的原文表述得很混亂, 表述能力跟 Code Ganker 差不多, 浪費了老子接近兩個小時的時間, 日他大爺. 以下是我重新表述的. 先看我的, 再看原文中的.

NCC 時, 每一個 memory reference 都要 go to memory(因為此時只有 go to memory 才是可靠的), 這跟前面的 test_and_set 是一樣的, 故 no difference.

CC-WU 時, 當 lock 由 1 變為 0 時, 所有被 update 的 CPU 都知道 lock 變為 0 了, 它們都想去執行 while((lock == busy) OR (test_and_set(lock) == busy))中的第二個, 即 test_and_set(lock) == busy, 此時會有 contention. 但此時還是 not too bad.

CC-WI 時, 當 lock 由 1 變為 0 時, 同樣會有上面 CC-WU 時的 contention.

但考慮即這麼一種情況: 之前 lock 一直是 1, 然後某個 CPU 執行了一個 atomic instruction, 且此 atomic instruction 不成功, 即並沒改變 lock 的值(lock 仍為 1), 則會是甚麼後果? 我們知道 atomic instruction 會引起 cache coherence, 即會引起 CC-WU 或 CC-WI. 到底是引起 CC-WU 還是 CC-WI, 是由硬件決定.

若該 CPU 的 atomic instruction 引起的是 CC-WU, 則其它的 CPU 被 update, 且這些 CPU 看到的是 lock 的值由 1 update 為 1, 即沒變, 故這些 CPU 繼續 spin, no problem.

若該 CPU 的 atomic instruction 引起的是 CC-WI, 則其它的 CPU 不是被 update, 而是它們的 cache copy 被 invalidate. 此時這些 CPU 要去 memory 中找 lock 的值(因為它們的 cache 被 invalidate 了, 不可靠), 然後知道 lock 的值仍為 1, 就繼續 spin. 每次有 CPU 執行 atomic instruction 時, 其它 CPU 都會訪問 memory, 這就跟 contention 加在一起, 成為很不好的行為.

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 is not quite the case. First if we don't have a cached coherent architecture (NCC) then every single memory reference will go to memory (因為此時只有 go to memory 才是可靠的), just like with test_and_set. So there's no difference what so ever. If we have cache coherence with write-update (CC-WU) then it's not too bad. The one problem is that all of the processors with write-update will see that the lock becomes free. 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 (CC-WI). In this case, every single attempt to acquire the lock not only that it will generate contention for the memory module, but it will also create invalidation traffic. Now when we talked about the atomics, we said that a one outcome of executing an atomic instruction, is that we will trigger the cache coherence, so the write update or write invalidate traffic are regardless of what the situation is. If we have a write update (CC-WU) 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 right update event. Great. No change whatsoever. That particular CPU can continue spinning from the cached copy. However, with the write invalidate (CC-WI), we will simply invalidate the cached copy. So it is possible that the lock was busy before the cache coherence event, before somebody executed an atomic instruction. And if the atomic construction 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 cache 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 cache 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, everyone will also at the same time, try to acquire this lock.

28. Let's look some more into the implications of the test and test_and_set spinlock implementation. Here is the question. In an SMP system with N processors, what is the complexity of the memory contentions, relative to the number of processors, that will result from the test and test_and_set spinlock when the lock is freed? I would like for you to write the O of n complexity for a system that has cache coherence with a write-update and also cache coherence with write-invalidate



Test_and_test-and-set Spinlock Quiz

```
// test (cache), test_and_set (Mm)
// spins on cache (lock == busy)
// atomic if freed (test_and_set)

spinlock_lock(lock):
    while((lock == busy) OR
          (test_and_set(lock) == busy))
```

In an SMP system with N processors, what is the complexity of the memory contention (accesses), relative to N , that will result from releasing a test_and_test-and-set spinlock?

CC with write-update:

$O(N)$

CC with write-invalidate:

$O(N^2)$

本題要求的是 complexity of memory contention, 即 complexity of memory accesses, 即要 access memory 多少次.

29. If caches are write-updated, 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. 要执行 test_and_set(lock) N 次, 注意执行它時, lock 的值是從 memory 中讀的(skeptical), 而不是 cache 中讀的. so the complexity of the contention is going to be order of $O(N)$. If the caches are write-invalidated, then all of the processor's caches will be invalidated after that initial lock release. For some processors, by the time they reread the lock value from memory in order to execute this part of the predicate, the lock will already have 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 reread 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 (記這樣的 processor 為 B 型 processor, 共有 $O(N)$ 個). As a result, they will try to execute that 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 (所有的 N 個 processor 都被 invalidate 了, 其中每個 processor 都要被 $O(N)$ 個 B 型 processor invalidate, 故總共 invalidate 的次數為 $O(N^2)$). 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, the input that gets created, of the contention that gets created when the lock is freed using that test_and_set

spin lock is $O(N^2)$.

Spinlock "Delay" Alternatives

```
spinlock_lock(lock):  
    while((lock == busy) OR  
          (test_and_set(lock) == busy))  
    {  
        // failed to get lock  
        while(lock == busy);  
        delay();  
    }
```

Delay after lock release
- everyone sees lock is free
- not everyone attempts to acquire it

Contention ... improved



Latency ... ok



Delay ... much worse

上圖中, 若 lock 變為 free, 則所有 processor 都去執行 `test_and_set(lock) == busy`. 最快的那個精子 processor 在執行 `test_and_set(lock) == busy` 時看到的 lock 為 free, 然後將 lock 設為 busy (好擋別人) 後就跳出循環 (能跳出循環是因為 `test_and_set(lock)` 返回的還是 lock 的舊值即 free), 高高興興地進 critical section 去了 (即它 acquire the lock 並將 lock 設為 busy). 其餘 processor 在執行 `test_and_set(lock) == busy` 時, lock 已經為 busy 了, 故這它們 failed to get lock, 然後這群 loser 們進入 while 循環的 body 中, 即執行 `delay`. 後來發現 while 末尾有個分號, 故 while 是不管 `delay()` 的。

30. A simple way to deal with the problems of the test and test and set lock is to introduce a delay. Here's 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, we'll come out of this while loop. And then before going back to recheck what the value of the lock is and if it's indeed free to try 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 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 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 and set instructions. So, it'll 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 (latency 是 metrics 中的第一個, 即 time to acquire free lock), this spinlock is still okay. 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's no contention for the lock, then that delay is just waste of time.

Spinlock "Delay" Alternatives

```
spinlock_lock(lock):  
    while((lock == busy) OR  
          (test_and_set(lock) == busy))  
    {  
        delay();  
    }
```

Delay after each lock ref.

- doesn't spin constantly
- works on NCC archs
- but can hurt delay even more

Contention ... improved



Latency ... ok



Delay ... much worse

Another variant of the delay-based spinlocks is to introduce a delay after every single lock reference. So every time we go through this while loop, we include a delay. The main benefit of this is that it works in non-cache coherent architectures (意思是此方法可以解決 non-cache coherent architectures 的一些問題, 當然也能解決一些別的 architectures 的一些問題, 但對 non-cache 最管用). 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. 由 sample final 第 5 題知, 此情況的 Delay 更 worse, 是因為: 當 lock 變為 free 後, while 會只執行 or 的前一個, 然後 delay(), 注意此時沒有執行 or 的後一個, 即還沒有 acquire lock 的. 即每次 lock 變為 free 後, 都先 delay() 一下, 才有可能再 acquire lock.

Picking a Delay (for a "delay" spinlock)

Static Delay (based on fixed value, e.g., CPU ID)

- simple approach
- unnecessary delay under low contention

Dynamic Delay (backoff-based)

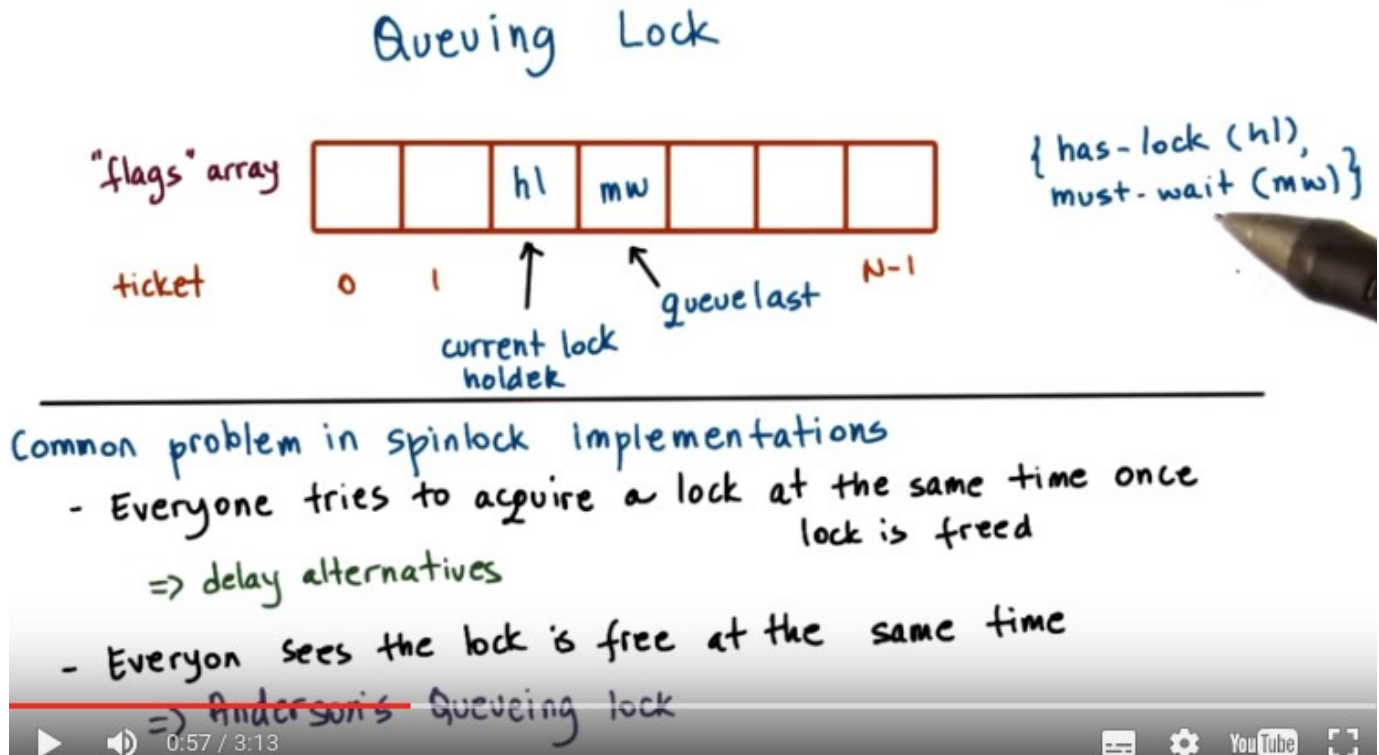
- random delay in a range that increases with "perceived" contention
- perceived == failed test-and-set()
- delay after each reference will keep growing based on contention or length of critical section



delay 之目的就是将不同的 processor 在時間上區分開來

31. One issue with the delay based spinlocks is how to pick a good delay value. Two basic strategies make sense. Static delay and a dynamic delay. For 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 (如 CPU ID) 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 one. The second process is running on a CPU with an ID 32, and so they're the only two that are contending for the spin lock, and yet the second process that's running on the CPU with ID 32 will be delayed substantially, regardless of the fact that there is no contention for the lock. So to avoid this issue, a more popular approach is use dynamic delay and that they named the delay 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 the system is operating in a mode of low contention then it will choose dynamic delay value that's within a smaller range. So it will back off just for a little bit, it will delay just for 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. They'll delay 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 behaviors. 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

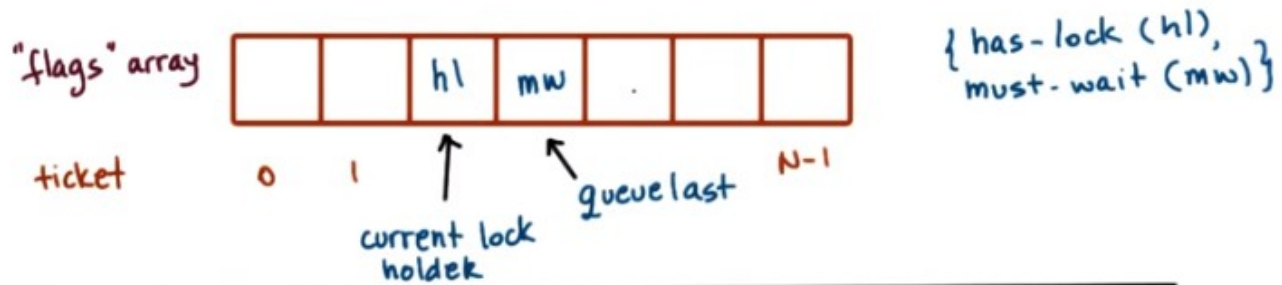
system is operating in low or high contention, so that it can pick an appropriate delay. So a good metric to estimate the contention, is to track the number of failed test_and_set operations. If a test and set operation fails, the more likely it is that there is a higher degree of contention. The problem with this, however, is if we're delaying after every single lock reference, then our delay will keep growing based on both whether there is indeed contention in the system (此時的 grow 是我們想要的), or if simply the owner of the critical section is delayed or if it's executing a long, critical section (此時的 grow 不是我們想要的. 此時為何要 grow? 因為一個人把 critical section 佔久了, 其它人就 fail, 即 fail 的多了). 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.



上圖先看下半部分, 再看上半部分.

32. 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 queuing 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 problem, if not everybody sees that the lock is free, 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 queuing lock looks as follows. It uses an array of flags, with up to N elements where N is the number of processors in the system. And every single one of the elements will have 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 clearly will have a value of has-lock. And they will also indicate the index into this array that has the last element on the queue. 注意此 array 不一定是滿的, queue last 即最後一個元素的 index.

Queuing Lock



- set unique ticket for arriving thread
- assigned $\text{queue}[\text{ticket}]$ is private lock
- enter CS when you have lock:
 - $\text{queue}[\text{ticket}] == \text{must-wait} \Rightarrow \text{spin}$
 - $\text{queue}[\text{ticket}] == \text{has-lock} \Rightarrow \text{enter CS}$
- signal/set next lock holder on exit

Downside:

⊖ assumes read-and-increment atomic

⊖ $O(N)$ size

CS 即 critical section.

$O(N)$ size 意思是我們需要 $O(N)$ 這麼多空間來放這個 queuing lock, 跟前面的 complexity of memory contention 不是一回事.

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 queue last 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 queue last pointer is incremented is done atomically. So basically this queuing lock depends on some support for an atomic read_and_increments. 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 queuing spinlock, the assigned element of this flags array, the $\text{queue}[\text{ticket}]$, that acts like a private lock (馬上講意思). What that means is that as long as the value of $\text{queue}[\text{ticket}]$ 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 flag in this queuing 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 sections 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 is much larger than the other spinlock implementations. All other locks we saw needed only one memory location to keep the spinlock information 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.

Queuing Lock Implementation

```
init:
    flags[0] = has-lock;
    flags[1..p-1] = must-wait;
    queuelast = 0; // global variable

lock:
    myplace = r&inc(queuelast); // get ticket
    // spin
    while(flags[myplace mod p] == must-wait)
    // now in C.S
    flags[myplace mod p] = must-wait;

unlock:
    flags[myplace+1 mod p] = has-lock;
```



Latency
more costly r&inc



Delay
directly signal
next CPU/thread
to run



Contention
better! but requires
CC and cacheline
aligned elements

only 1 CPU/thread sees
the lock is free and
tries to acquire lock!

p 為前面一段中的 N, 即 number of processors

CC 即 cache coherent

上圖中, //spin 是對 while 一句的注釋, 但//now in CS 不是對它下面那句 flags 的注釋, //now in CS 代表一個 block, 此 block 是 operate on CS.

33. Here is the implementation of the queuing 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 the global variable queuelast. To obtain a ticket into this queueing lock a process must first perform a read_and_increment atomic operation (即 r&inc). 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 of zero 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 will get tickets that will point to some elements of the flags array that are from zero to p minus one. 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, you must wait in order to get this build ready for the next threads and next processes that try to acquire this lock. Notice that we're using some modular math (mod p) in order to wrap around this index read in increment and just continue increasing the value queuelast where as our array's 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 its value from must-wait, it will become 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 spin lock implementation involves a read and increment on a variable, queuelast. All of this spinning condense~ 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 on the elements of the flags array. These are two different memory locations, two different variables. >From a latency (不是 delay) 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. 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 delays 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 up front 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 stability 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, this spinning has to happen on potentially remote memory refer. 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 spin lock 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 queue in lock solves that problem. By having a separate, essentially a private, lock in this array of locks, only one thread at a time sees that the lock is free, and only one thread at a time attempts to acquire the lock.

34. Assume we are using the Anderson's queueing spinlock implementation where each array element can have one of two values: has-lock, and let's say that's zero, and must-wait, let's say we use one for that. Now if a system has 32 CPUs, how large is the array data structure that's used in the queueing spinlock? Your choices are 32 bits, 32 bytes or neither. Choose the correct answer.



Queueing Lock Quiz

Assume we are using Anderson's queueing spinlock implementation where each array element of the queue can have one of two values: has-lock (0) and must-wait (1). If a system has 32 CPUs, then how large is the array data structure?



- ☐ 32 bits
- ☐ 32 bytes
- ☒ neither

35. The correct answer is neither. Remember that for the queueing implementation to work correctly each of the elements of the array 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 will be 32 by 64 byte. But in practice, there may be other cache line sizes that are used on the architectures that you use.

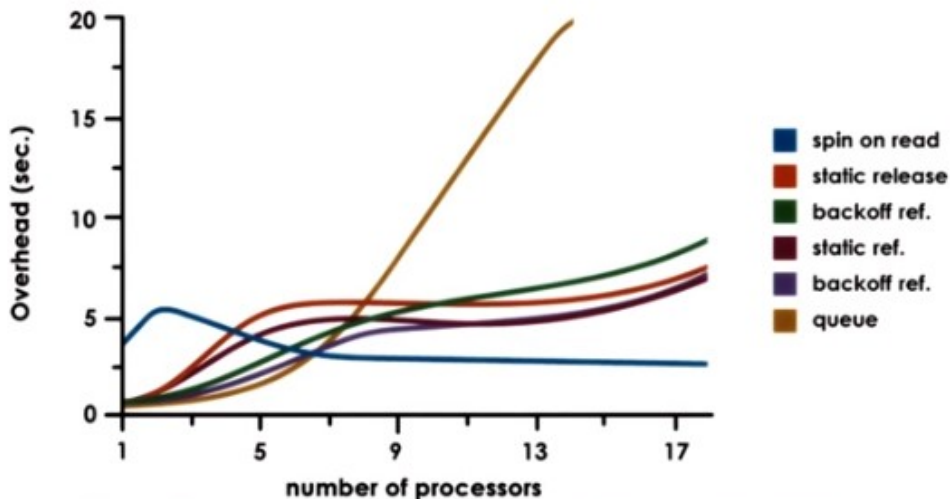


Fig. 3 - Principle performance comparison: spin-waiting overhead (seconds) in executing benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

Setup

- N processes running CS 1M times
- N varied based on system

Metrics

- overhead compared to ideal perf.
- theoretical limit based on # of CS to be run

Instructor Notes

ERRATA : Color legend error, please check figure in paper. e.g., queue should be blue, and spin-on-read should be gold. 即上圖中，從右端看，最下面那個藍線為 queue，最上面那個金線為 spin-on-read，即 test_and_test_and_set.

36. Then finally, let's take a look at one of the results from the performance measurements that are shown in the 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 one process per processor (processor 這個名稱原來就來自 process 啊). In the platform that was used, was a Sequence Symmetry that had 20 processors. So that's why the maximum number of processes was also kept to 20. Also this platform was cache coherent with write_and_invalidate. The metric that was computed based on the experiments was the overhead that was measured compared to in 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 effects due to the fact that each of these critical sections needs to be locked and unlocked. Then how long would it take to run these number of critical sections. The measured difference between that theoretical limit and whatever it actually took to perform this experiment was considered the overhead. And this is what these graphs represent. These experiments were performed for every one of the different spinlock implementations we discussed. >From the spin on read, the test_and_test_and_set, all the way to the queuing implementation. These results don't include that the basic test_and_set when we're spinning on the atomic construction. Basically that implementation would result in something that would just completely be off the charts. It would be the highest overhead the worst performance measurement. And then, notice how we have for the different variations of the delay both the static and the dynamic delay.

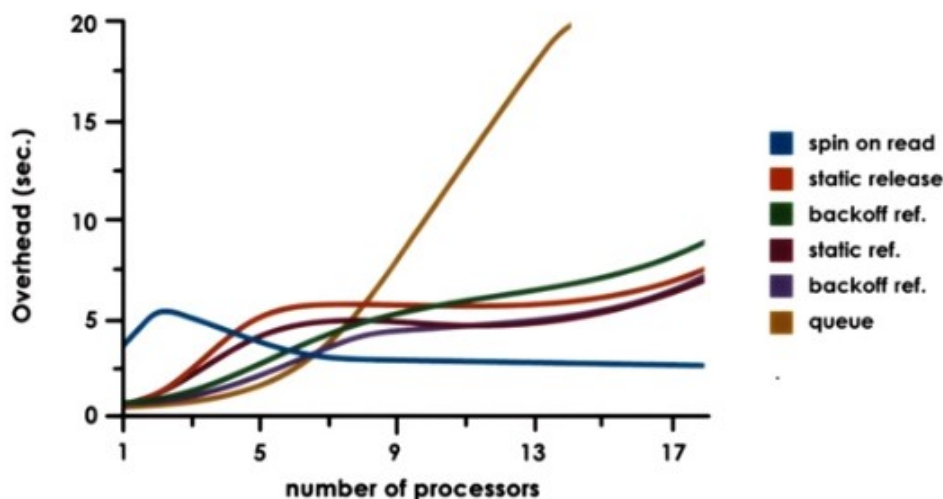
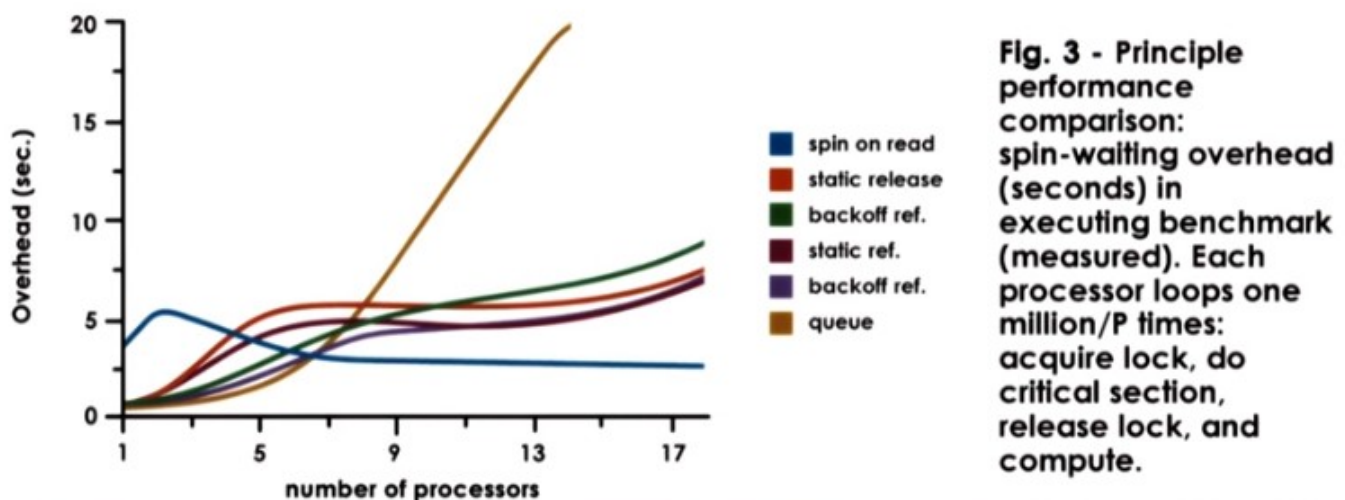


Fig. 3 - Principle performance comparison: spin-waiting overhead (seconds) in executing benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

Under high loads
- queue best (most scalable), test_and_test_and_set worst
static better than dynamic, ref. better than release (avoids extra invalidations)

上圖中，從右端看，最下面那個藍線為 queue，最上面那個金線為 spin-on-read，即 test_and_test_and_set. 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 log, that's this line here, that one will perform worst. That isn't 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 log, we have an order of $O(N^2)$ 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 constructions. Whereas with dynamic, we still end up with some randomness, with some number of collisions. Those are avoided in the static case. Also note that delaying after every single memory reference is slightly better than delaying after the log is freed only, it's only on release. Because when we avoid after every reference, we end up avoiding some additional invalidations that come from the fact that sequence is a write invalidate architecture.



Under light loads

- test-and-test-and-set good (low latency)
- dynamic better than static (lower delay)
- queueing lock worst (high latency due to r&inc.)

上圖中, 從右端看, 最下面那個藍線為 queue, 最上面那個金線為 and spin-on-read, 即 test_and_test_and_set.

Under light loads, when we have few processes then few processors as well, we can make couple of observations. First we see that test and test and set performs really pretty well in this case. And that's because this implementation of a spinlock has low latency. We were just performing a check, if lock equal busy and then we were moving on to the atomic test_and_set. We also see that in terms of the delay alternatives, the dynamic delay alternatives, the backup delay alternatives, perform better than the static ones. And that is because with the dynamic alternatives, we end up with lower delay. But 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 that are contending on the lock and so we have wasted cycles. We also see that under light loads, the queueing lock performs the worst. This is the performance of the queueing lock. And the reason for that is we explain that with the queueing clock, we have 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 is 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, number of processors, write invalidate, write update, etc. The paper includes additional results that point to some of these trade offs in more detail.[spi](#)



37. In this lesson, we talked about other synchronization constructs beyond just mutex 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.

38. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.