

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 06
Lecture – 27
How to implement Locking (Hardware Solutions and Usage)

Hello. In this particular video we will look at Hardware Solutions to Solve the Critical Section Problem.

In the previous video we had looked at Software Solutions to Solve the Critical Section Problem. However, in practise these solutions are not very efficient. So in practise we have both in the operating system as well as in applications, hardware techniques are used to efficiently solve the critical section problem. We will start this video with the small motivating example.

(Refer Slide Time: 00:52)

Analyze this

- Does this scheme provide mutual exclusion?

Process 1

```
while(1){
  while(lock != 0);
  lock = 1; // lock
  critical section
  lock = 0; // unlock
  other code
}
```

lock=0

Process 2

```
while(1){
  while(lock != 0);
  lock = 1; // lock
  critical section
  lock = 0; // unlock
  other code
}
```

No

lock = 0

P1: while(lock != 0);

P2: while(lock != 0);

P2: lock = 1;

P1: lock = 1;

.... Both processes in critical section

context switch

46

Let us start with this particular example. Let us say we have the two processes; process 1 and process 2 and both are having a similar critical section that accesses the same shared data. And also let us say that we have this shared variable call lock, so this variable is shared between process 1 and process 2. Now what each of these processes do is two things; first in order to lock the critical section, the process would first execute this while loop until lock becomes 0, when lock becomes 0 then the process would enter here and

set lock to 1 and then execute the critical section. At the end of the critical section the process would set lock to 0 in order to unlock the section. The question that I would post over here is; does this particular scheme achieve mutual exclusion? The answer is no.

And we have seen such things in the previous video as well. Due to context switching between the two processes we would reach a state some times when both processes execute in the critical section. Thus, we will not be able to achieve mutual exclusion. So, for instance if we actually look at this, we see that lock as in initial value of 0. Then suppose process P1 executes in the CPU, and it executes while lock not equal to 0. Since, we have lock as 0 over here, so this particular while loop will break, however we have a context switch that occurs before the process could set lock equal to 1.

Now, this context switch results in process P2 executing. Now P2 will also see the value of lock equal to 0, and therefore break from this while loop and then set lock to 1. Now again if there is a context switch occurring and process P1 will continue to execute from where it stopped, it will see the old value of lock which is 0 because, the context is returned impact to the old value of the process before the context switch occurred. Therefore, process P1 will see the value of lock as 0. And therefore, it would set lock equal to 1 and enter into the critical section. Thus we have reached the state where both processes are in the critical section.

(Refer Slide Time: 03:42)

If only...

- We could make this operation atomic

```
Process 1
while(1){
  while(lock != 0);
  lock= 1; // lock
  critical section
  lock = 0; // unlock
  other code
}
```

Make atomic

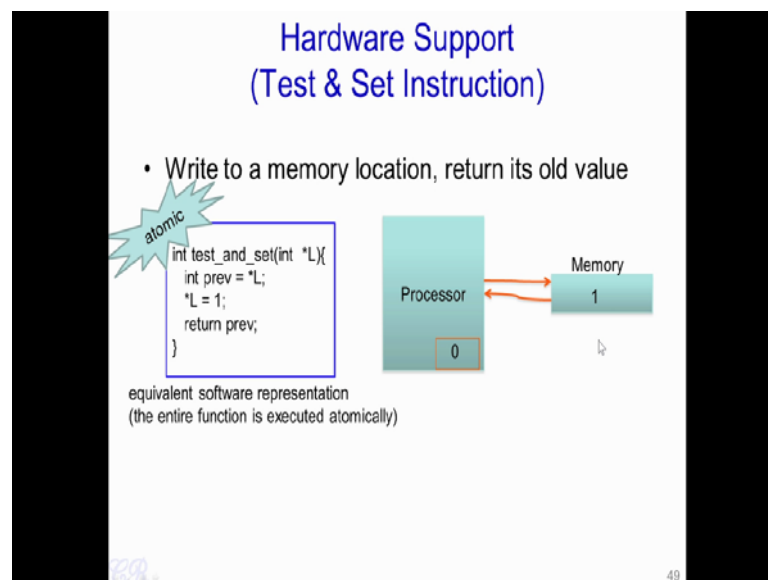
Hardware to the rescue....

47

Now, the main reason that this particular scheme failed was that as we have written this in software we were unable to make these two statements atomic. That is, we are unable to ensure that a context switch do not occur in between these two statements. Said another way, we are unable to make these two statements execute as a single unit. And thus, due to this reason we were unable to achieve mutual exclusion. Now most processors such as the x86 Intel processors have dedicated instructions that will ensure that these statements are executed automatically.

Let us take an example of such an atomic instructions through which we could implement this set of statements in an atomic manner. So we will take a very general instruction first and then analyse specifically for the Intel x86 type processors.

(Refer Slide Time: 04:51)



The first instruction that we would see today is the Text and Set Instruction. Essentially if we were to write this instruction in c it would look something like this, it is a function which takes a pointer to a memory location. In this function first the contents of that memory location is stored into this variable called previous, then that memory is set to 1 and the value returned is the previous value. In other words this function would return the previous contents of the memory and set that memory to a value of 1.

Now, when we look at this from hardware prospective and essentially from the processor prospective, this entire function is an atomic function. Essentially we would have one instruction that would do all of this thing in one shot that is we would have one

instruction that would perform all of these operations automatically. Let us look at this particular diagram to demonstrate how this thing actually works.

Let say that we have the processor and the memory location pointed to by L is over here. So, what would happen when this test and set gets executed? Is the following; first the previous contents of the memory get loaded into a register present in the processor and then the memory context is set to 1. Let us look at it another time; first the memory contents pointed to by L is loaded into a register present in the processor and then the context of that memory location is set to 1.

(Refer Slide Time: 06:46)

**Hardware Support
(Test & Set Instruction)**

- Write to a memory location, return its old value

```
atomic
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

Processor 0
Processor 0
Memory 1

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

50

Now, the atomicity of the test and set instruction can be defined or explained as this statement here which takes that, if two CPU's execute test and set at exactly the same time the hardware or that is a processor ensures that one test and set for one of these processors does both it is operations. That is, reading context of the memory to a register and setting that particular memory location, so both these steps are done before the other process starts executing.

In other words, it is not possible for two processors to execute the test and set instruction and both processors read the previous value of the memory location. So, essentially this thing is wrong, is not correct that is it is not possible for both processors to simultaneously read the contents of the memory location and set the value of memory.

(Refer Slide Time: 07:56)

Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

```
atomic
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

Processor 0
Processor 1
Memory 1

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

51

However, what actually is guaranteed by the processor hardware is that when processors run on each processor and both processors execute the test and set instruction at exactly the same time the hardware will ensure that one process completes its entire instruction before the next process could execute its instruction. Therefore, in such a case one process would read the value of 0 while the other process would read the value of 1. Now we will see how this particular instruction is used to solve the critical section problem.

(Refer Slide Time: 08:30)

Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

```
atomic
int test_and_set(int *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

```
while(1){
    while(test_and_set(&lock) == 1);
    critical section
    lock = 0; // unlock
    other code
}
```

equivalent software representation
(the entire function is executed atomically)

Usage for locking

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

53

This particular snippet over here shows how a process could use the test and set atomic instruction which hardware supports in order to solve the critical section problem. Essentially, in order to lock the critical section we would have a while loop present over here which would invoke the test and set with a lock that is with the memory location. Now, this while loop will execute continuously until the value returned by the test and set is 0; when the value returned by test and set instruction is 0 then critical section is entered and at the end of this the value of lock is set to 0.

When there are multiple processors having such a code snippet, the hardware guarantees that the test and set for exactly one process would return a value of 0. All other processes will loop continuously in this particular while loop. So when this process as completed executing the critical section the value of lock would be set to 0. So, setting the value of lock could to 0 would result in exactly one other process to obtain a value of 0 from the return of the test and set. Therefore, exactly one other process would then enter into the critical section.

The first innovation of the test and set will read a 0 and set the lock to 1 and return. The second test and set innovation will then see the lock as 1 and will loop continuously until the lock become 0. The value of the lock becomes 0 only when the first process unlocks set that is exclusively sets the value of lock to 0. In this way we see that by a little help from the hardware it is feasible to solve critical section problem very efficiently.

(Refer Slide Time: 10:41)

Intel Hardware Support (xchg Instruction)

- Write to a memory location, return its old value

atomic

```
int xchg(int *L, int v){
  int prev = *L;
  *L = v;
  return prev;
}
```

```
graph LR
    P1[Processor 20] <--> M[Memory 10]
    P2[Processor 30] <--> M
```

equivalent software representation
(the entire function is executed atomically)

Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

54

So, Intel systems do not support the test and set instruction. On the other hand it supports an instruction known as exchange; xchg. An exchange instruction is represent over it go here and this is also an atomic instruction and the exchange instruction takes two parameters, so it takes a memory address and an integer value. Now what is done is that the context of that memory location is stored in previous, then the value of v that is the value which is passed to exchange is then stored into that memory location and the previous value is returned.

So, essentially what this achieves is that we are able to exchange register variable with a memory location. To take an example, so we look at this when once an exchange instruction is executed by a process running in this processor it will exchange a register value with that of memory location. And this entire thing is done automatically.

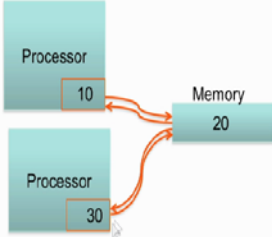
(Refer Slide Time: 11:49)

Intel Hardware Support (xchg Instruction)

- Write to a memory location, return its old value

atomic

```
int xchg(int *L, int v){
  int prev = *L;
  *L = v;
  return prev;
}
```



equivalent software representation
(the entire function is executed atomically)

Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

54

Even if another process executes the exchange at exactly the same time, the Intel hardware ensures that these two operations are done distinctively. So, first one process would execute and complete the exchange instruction and only then the second process would execute and exchange the data.

(Refer Slide Time: 12:14)

Intel Hardware Support (xchg Instruction)

- Write to a memory location, return its old value

atomic

```
int xchg(int *L, int v){
    int prev = *L;
    *L = v;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

54

(Refer Slide Time: 12:14)

Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}
```

```
void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}
```

```
void release(int *locked){
    *locked = 0;
}
```

55

This particular slide shows how an exchange instruction in Intel hardware is used to solve the critical section problem. We could have two primitive functions called Acquire and Release. So the acquire is used to lock critical section, while release is used to unlock a critical section. This acquires and release functions are passed pointer to a memory location known as Locked. In acquire in an infinite rather in acquire in a loop this function exchange locked is invoked. So, an exchange present over here and it is passed two parameters; the address of the memory location in this case the address of

locked and the value which you want to set. First the value in this case one is moved into the eax register in the Intel processor and then the exchange instruction is invoked.

As a result the memory or the data which was stored in memory gets loaded into the eax register. And the value in this case one which was loaded into the eax register gets stored into memory. Now this particular eax register is what is returned back to acquire. This loop would break if the value returned by exchange that is the context of the eax register is 0. In order to release the lock, we simply set the locked value to 0.

Let see how it works. First, the memory location over here would be 0 and when the process invokes exchange instruction that 0 value, is pushed into the eax register, and the value of one which was present in the eax register comes into the memory.

(Refer Slide Time: 14:29)

The diagram illustrates the hardware support for the xchg instruction. It shows two processors: one labeled 'Get Lock Processor' with a value of 0 in its register, and another labeled 'Processor' with a value of 1 in its register. A central 'Memory' block contains the value 1. Arrows indicate the flow of data: the value 1 from memory is moved to the 'Get Lock Processor' register, and the value 0 from its register is moved to the 'Memory' block. To the right, a code block shows the implementation of the xchg instruction and its use in an acquire-release lock algorithm. A note states that the value returned by xchg is stored in the %eax register.

Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :
xchg reg, mem

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}
```

```
void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}
```

```
void release(int *locked){
    *locked = 0;
}
```

56

As a result, the memory as a value of one while the eax register present in the processor as a value of 0 and this is what is used to break from this while loop and enter into the critical section. Now in order to release the lock, we have simply setting the value of locked to 0. Now when a second process invokes acquire it will continue to loop in this while loop until the value of this memory location is set to 0 by the first processor.

(Refer Slide Time: 15:08)

Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :
xchg reg, mem

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

void release(int *locked){
    *locked = 0;
}
```

57

You see over here, so in order to release lock we simply set the value in memory to 0 and then in another processor perhaps would be able to obtain the value of 0 and then acquire the critical section.

(Refer Slide Time: 15:23)

High Level Constructs

- Spinlock
- Mutex
- Semaphore

60

This instruction like we exchange is used to build higher level constructs which are used in various critical section problems. So, we will look at the Spinlock, and Mutex in this video and Semaphore in a later video.

(Refer Slide Time: 15:39)

Spinlocks Usage

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will **acquire** the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process **releases** it

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}

void acquire(int *locked){
    while(1){
        if(xchg(locked, 1) == 0)
            break;
    }
}

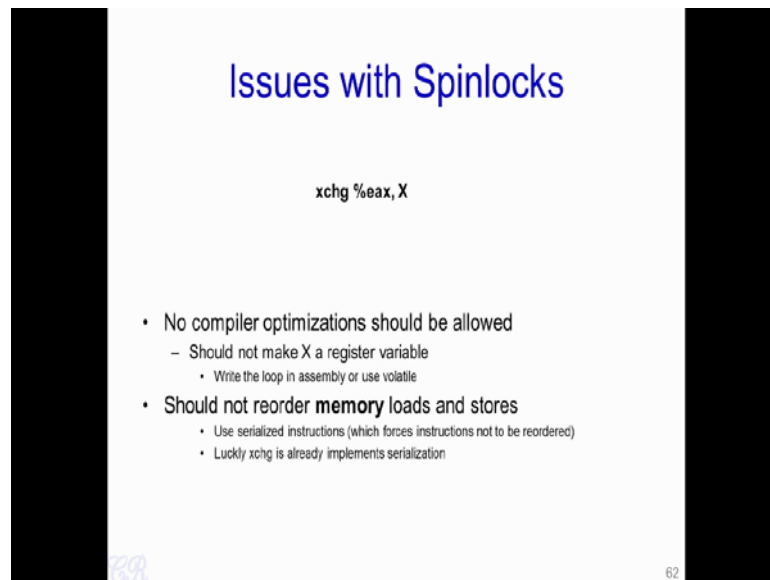
void release(int *locked){
    *locked = 0;
}
```

See spinlock.c and spinlock.h in xv6 [15] 61

Spinlock is what we have seen before. It has two functions acquire and release, while acquire is used to gain access to the critical section essentially it is used to lock the critical section, while the release function is used to unlock the critical section. In order to use this we could have two processor; process 1 and process 2. In order to enter into the critical section the process as to acquire the lock and this function acquire will continuously loop until the value return by exchange is 0. When the value of exchange is 0 then this loop breaks and the process 1 would enter into the critical section. Releasing the lock is just done by setting the value of locked to 0.

So, one process will acquire the lock at given time while the other process will wait in this particular loop and continuously invoke the exchange function until it obtains 0. So the first process in order to release the lock would have set the value to 0, this would cause the second process to break from this loop and enter into the critical section. We you could see more details about the way spinlocks are implemented in xv6 by looking into these two files that is spinlock dot c and spinlock dot h.

(Refer Slide Time: 17:13)



The slide is titled "Issues with Spinlocks" in blue text. Below the title, the assembly instruction `xchg %eax, X` is displayed. A list of bullet points follows, detailing compiler optimization issues. The first bullet point is "No compiler optimizations should be allowed", with a sub-bullet "Should not make X a register variable" which includes "Write the loop in assembly or use volatile". The second bullet point is "Should not reorder memory loads and stores", with sub-bullets "Use serialized instructions (which forces instructions not to be reordered)" and "Luckily xchg is already implements serialization". The slide number "62" is in the bottom right corner.

Issues with Spinlocks

```
xchg %eax, X
```

- No compiler optimizations should be allowed
 - Should not make X a register variable
 - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
 - Use serialized instructions (which forces instructions not to be reordered)
 - Luckily xchg is already implements serialization

62

We will now look at some of the issues that go about with the use of the exchange instruction. As we have seen the exchange instruction is the most crucial part of any of the constructs such as the spinlock that we have seen in the previous slide. So therefore, it is important to understand what are the issues related to the exchange instruction essentially this particular format of the exchange instruction exchanges data between register `eax` and a memory location `X`. Now it should be ensuring that there are no compiler optimizations that are done on this variable `X`.

So, some of the common optimization that is possible is to make the value of `x` stored in a register. In such a case, we are simply exchanging data between register `eax` and then other register which will not solve the purpose. Therefore, we should write this particular loop in assembly or use the keyword `volatile`.

Another requirement while implementing the exchange instruction is to ensure that memory operations are not reordered. The CPU should not reorder memory loads and stores. So in order to achieve this we use serialized instructions which force instruction not to be reordered. Luckily for us the exchange instruction by default already implements serialization. So there is nothing much we need to take care about this.

(Refer Slide Time: 18:46)

More issues with Spinlocks

`xchg %eax, X`

- No caching of (X) possible. All `xchg` operations are bus transactions.
 - CPU asserts the LOCK, to inform that there is a 'locked' memory access
- acquire function in spinlock invokes `xchg` in a loop...each operation is a bus transaction **huge performance hits**

63

However, what we need to look at more closely is the fact of cache memories. Now, recollect that exchange instruction exchanges data between a register and memory location. Now each CPU present in the system could have their own private cache, for instance CPU 0 as it is own private L 1 cache, similarly CPU 1 as it is own L 1 cache. It should be ensured that this value of x is not cached in each of these CPU's, essentially caching of CPUs is not possible. But rather, each and every execution of this exchange instruction should actually go to memory and load or store the value of X.

Secondly it should also ensure that when one CPU is reading and writing to this x, that is in other words when one CPU is invoking the exchanged instruction during this time no other CPU can modify the value of X, as this will break the atomic nature that is present. In order to do this the CPU asserts what is known as a lock line to inform all other CPU's in the system that there is locked memory access that is going to take place.

As a result of this lock instruction all exchanged instruction which read and write data would result in the reading and the writing data from a memory. So, this may be tremendous amounts of performance hits.


(Refer Slide Time: 20:30)

A better acquire

```
int xchg(addr, value){
    %eax = value
    xchg %eax, (addr)
}
```

```
void acquire(int *locked){
    reg = 1
    while(1)
        if(xchg(locked, reg) == 0)
            break;
}
```

```
void acquire(int *locked) {
    reg = 1;
    while (xchg(locked, reg) == 1)
        while (*locked == 1);
}
```



Original.
Loop with xchg.
Bus transactions.
Huge overheads

Better way
inner loop allows caching of locked. Access cache instead of memory.

64

Therefore, in order to make a better acquire, acquire which is more efficient we will look at a small tweak to this particular acquire function. So originally, we use to have a loop over here and in this loop we would continuously keep invoking the exchange function and checking whether it returns a value of 0.

Note that, each of these exchange instructions as a huge performance overhead because it requires that the locked keyword that is X value would go all the way to the memory and read and write data from the memory. Essentially, the caching is not possible and therefore huge overheads.

On the other hand if we make a minor change to this acquire function as shown over here it would improve performance quite significantly. So here we have two loops one is the regular exchange loop which as you know would result in a bus transaction and as huge overheads, while does an inner loop which simply loops checking the value of the memory location locked.

This particular internal thing can be cash able and therefore, will not incur much performance overhead, while, the external while loop will have significant overhead, but it is not invoking two of them. Most of the time this particular cached memory it would be read.

Now the cache coherency protocol will ensure that when another process changes the value of locked this process would see the value of locked changing from 1 to 0 and it will exit the while loop and go back to this particular outside while loop. The exchanged function would then exchange the locked value with the register value and set the value to 1, and therefore it break from the while loop.

Thank you.