Design of Parallel and High Performance Computing
HS 2017
Torsten Hoefler, Markus Püschel **Homework 5**
Department of Computer Science
ETH Zurich

A new lock?

Does the following code ensure that at any time, at most one thread is in the critical region if we assume sequential consistency? Use sequential consistency to prove your answer.

The variables me and $other$ are thread-local, the variables $want[0]$, $want[1]$ and $turn$ are shared. The variables $want$ and $turn$ are initially zero, me contains the thread id($\in 0, 1$) for each thread, while $other$ contains the value $1 - me$.

```
want[me] = 1;
while (turn != me) {
    while (want[other]) {}
    turn = me;
}
// CR
want[me] = 0;
```

Peterson Lock on x86

Prove that the following implementation of the Peterson lock is correct in the x86 memory model. The variables me and $other$ are thread-local, the variables $want[0]$, $want[1]$ and $victim$ are shared. The variables $want$ and $victim$ are initially zero, me contains the thread id($\in 0, 1$) for each thread, while $other$ contains the value $1 - me$.

Use the following properties of the x86 memory model (cf. Sec. 8.2.2. of the Intel Intel 64 and IA-32 Architectures Software Developers Manual, Vol 3A):

- Writes to memory are not reordered with other writes.
- Reads are not reordered with other reads.
- Reads cannot pass earlier LFENCE and MFENCE instructions.
- Writes cannot pass earlier LFENCE, SFENCE, and MFENCE instructions.
- MFENCE instructions cannot pass earlier reads or writes.
- Stores are not reordered with older loads.

```
want[me] = 1;
victim = me;
asm("mfence");
while (want[other] && (victim == me)) {};
// CR
asm("mfence");
want[me] = 0;
```

Can you explain the reason for the mfence instruction in the unlock phase?

Solution: A new lock?

Assume both threads get to the critical region at the same time. They could execute the following read/write statements:

$T_A : W(want[A] = 1) \rightarrow R(turn == A) \rightarrow CR$
$T_B : W(want[B] = 1) \rightarrow R(turn == A) \rightarrow R(want[A] == 0) \rightarrow W(turn = B) \rightarrow R(turn == B) \rightarrow CR$

A sequentially consistent interleaving of those read/write operations would be:

$W_B(want[B] = 1)) \rightarrow R_B(turn == A) \rightarrow R_B(want[A] == 0) \rightarrow W_A(want[A] = 1) \rightarrow R_A(turn == A) \rightarrow CR_A \rightarrow W_B(turn = B) \rightarrow R(turn == B) \rightarrow CR_B$

This counterexample shows that the algorithm does not grantee mutual exclusion. ■

This solution can be achieved by looking at the problem long enough. How could we arrive there using a "standard" technique?

To check if a lock guarantees sequential consistency one common technique is to assume it does not, trying to reconstruct a sequentialy consistent order of the locks instruction where two threads reach the critical region simultaneously.

If this leads to a contradiction the lock does provide mutual exclusion, otherwise we found a counterexample.

Without loss of generality assume thread B was the last one to write to turn:

$W_B(turn = B) \rightarrow\rightarrow R_B(turn == B) \rightarrow CR_B$

if that is the case then B had to read $want[A]$ to be 0, otherwise it would spin, instead of entering the critical section

$R_B(want[A] == 0) \rightarrow W_B(turn = B) \rightarrow R_B(turn == B) \rightarrow CR_B$

and before that B has to read $turn$ as A, otherwise it would not have entered the while loop in the first place.

$W_B(want[B] = 1) \rightarrow R_B(turn == A) \rightarrow R_B(want[A] == 0) \rightarrow W_B(turn = B) \rightarrow R(turn == B) \rightarrow CR_B$

For thread A there are now two paths to reach the critical section:

- Reading $turn$ as A
- Reading $turn$ as B and $want[B]$ as 0

For the first option we get the sequence $W(want[A] = 1) \rightarrow R(turn == A) \rightarrow CR_A$ which is exactly the counterexample shown before.

For the second path we get (by the same method as described above):

$W_A(want[A] = 1) \rightarrow R_A(turn == B) \rightarrow R_A(want[B] == 0) \rightarrow W_A(turn = A) \rightarrow R(turn == A) \rightarrow CR_A$

However, there is no sequentially consistent interleving of those instructions with those from process B, because $R_A(want[B] == 0)$ has to happen before $W_B(want[B] = 1)$, while $R_B(want[A] == 0)$ has to happen before $W_A(want[A] = 1)$.

Solution: Peterson Lock on x86

The memory model on x86 guarantees (among other things):

1. Stores are not reordered with other stores
2. Loads are not reordered with other loads
3. Stores to the same location have a total order
4. Neither loads nor stores are reordered with mfence instructions

5. Transitive visibility

For both threads to enter the CR the following must happen:

$T_A : W(want[A] = 1) \xrightarrow{1} W(victim = A) \xrightarrow{4} R(want[B]) \xrightarrow{2} R(victim) \rightarrow CR_0$ (1)
$T_B : W(want[B] = 1) \xrightarrow{1} W(victim = B) \xrightarrow{4} R(want[A]) \xrightarrow{2} R(victim) \rightarrow CR_1$ (2)

Assume that T_A was the last thread to write into victim:

$W_B(victim = B) \xrightarrow{3} W_A(victim = A)$ (3)

This implies that T_A reads $victim$ as 0 (because there are no other writes to victim). Therefore, to enter the critical section, it must have read $want[B]$ as 0.

$W_A(victim = A) \xrightarrow{4} R_A(want[B] == 0)$ (4)

If we combine the equations 2-4 we get

$W_B(want[B] = 1) \rightarrow W_B(victim = B) \rightarrow W_A(victim = A) \rightarrow R_A(want[1] == 0)$

This is a contradiction, as $want[1]$ is set to 1 and later observed to be 0, with no other writes in between. ■