Design of Parallel and High Performance Computing
HS 2018
Markus Püschel, Torsten Hoefler
Department of Computer Science
ETH Zurich

Homework 3

Sequential Consistency

Exercise 1

For the following executions, please indicate if they are sequentially consistent. All variables are initially set to 0.

1. P1: W(x,1);
   P2: R(x,0); R(x,1)
2. P1: W(x,1);
   P2: R(x,1); R(x,0);
3. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,1); R(x,2);
4. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,2); R(x,1);
5. P1: W(x,1);
   P2: W(x,2);
   P3: R(x,2); R(x,1);
   P4: R(x,1); R(x,2);
6. P1: W(x,1); R(x,1); R(y,0);
   P2: W(y,1); R(y,1); R(x,1);
   P3: R(x,1); R(y,0);
   P4: R(y,0); R(x,0);
7. P1: W(x,1); R(x,1); R(y,0);
   P2: W(y,1); R(y,1); R(x,1);
   P3: R(y,1); R(x,0);

Solution

1. P2: R(x,0); P1: W(x,1); P2: R(x,1)
2. not sequentially consistent: it is not possible to read 0 from x once the value 1 has been written to it
3. P1: W(x,1); P3: R(x,1); P2: W(x,2); P3: R(x,2);
4. P2: W(x,2); P3: R(x,2); P1: W(x,1); P3: R(x,1);
5. not sequentially consistent: P3 and P4 observe the writes performed by P1 and P2 in different order.
6. P4: R(y,0); R(x,0); P1: W(x,1); P1: R(x,1); P1: R(y,0); P3: R(x,1); P3: R(y,0); W(y,1); R(y,1); R(x,1)
7. not sequentially consistent because based on P1s observations W(x,1) happens before W(y,1), but based on P3s observations W(y,1) must happen before W(x,1)

The x86 Memory Model: TLO+CC

1. Describe the memory model of the x86 architecture. Where does it differ from sequential consistency.

2. Two threads execute the following code (given in AT&T assembly syntax) on a machine using TLO+CC. Is it possible that the registers eax $= 0 \wedge$ ebx $= 0$ after both threads have finished?. Would it be possible in sequential consistency?

Initial: All registers and memory locations are 0	
Thread A	Thread B
movl \$1, 0x42 movl 0x50, %eax	movl \$1, 0x50 movl 0x42, %ebx

Solution

The x86 memory model provides the following guarantees:

- Reads are not reordered with other reads. (R $\rightarrow$ R)
- Writes are not reordered with other writes. (W $\rightarrow$ W)
- Writes are not reordered with older reads. (R $\rightarrow$ W)
- Reads may be reordered with older writes to different locations but not with older writes to the same location (W $\not\rightarrow$ R) **This point is different from sequential consistency. To still allow synchronized programming, the following guarantees are given**
- Memory ordering obeys causality
- Writes to the same location have a total order.
- In a multiprocessor system, locked instructions have a total order.
- Reads and writes are not reordered with locked instructions.

In sequential consistency, it is impossible that both threads observe eax=0 $\wedge$ ebx=0, however, in TLO+CC this is possible.

First, we will prove that it is impossible in sequential consistency. For this we first translate the above fragments into read and write statements:

Initial: All registers and memory locations are 0	
Thread A	Thread B
W(a, 1) R(b, x) W(eax, x)	W(b,1) R(a,x) W(ebx, x)

Now we assume that it **is** possible for both threads to finish and have eax=ebx=0 and show that this leads to a contradiction. Without loss of generality, we also assume Thread B is the last one to finish.

The first assumption tells us that the processes must execute the following read/write sequences:

$W_A(a,1) \rightarrow R_A(b,0) \rightarrow W_A(eax,0)$
$W_B(b,1) \rightarrow R_B(a,0) \rightarrow W_B(ebx,0)$

the second assumption translates to

$W_A(eax,0) \rightarrow W_B(ebx,0)$

if we combine the three equations, we see that it is neccessary to have $W_A(a,1) \rightarrow R_B(a,0)$ at the same time $W_B(b,1) \rightarrow R_A(b,0)$ which means that our initial assumption was false, and it is eax=ebx=0 is impossible in sequential consistency.

In the x86 memory model however, it is legal to reorder the given sequences into

Initial: All registers and memory locations are 0	
Thread A	Thread B
R(b, x)	R(a,x)
W(a, 1)	W(b,1)
W(eax, x)	W(ebx, x)

which allows the following interleaving:

$R_A(b,0) \rightarrow R_B(a,0) \rightarrow W_A(a,1) \rightarrow W_B(b,1) \rightarrow W_A(eax,0) \rightarrow W_B(ebx,0)$.

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;
```

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 $W_A(want[A] = 1)$.