Parallel and Distributed Computing DD2443 - Pardis24 Exercises for Lecture 8

Name: Casper Kristiansson

September 19, 2024

Exercise 1

Question

Assume an initial memory in which X, Y and A are all set to 0. Consider the following threads:

T0: X = 1

`if (Y = 0) { A = 2 }`

T1: Y = 1

`if (X = 0) { A = 3 }`

For each of the following schedules, mark if they are allowed under Sequential Consistency (SC), Total Store Ordering (TSO), neither, or both. Indicate which reordering constraints they violate, if any.

1. T0:Read(Y,0), T1:Write(Y,1), T0:Write(X,1), T1:Read(X,0), T0:Write(A,2), T1:Write(A,3)
2. T0:Write(X,1), T1:Write(Y,1), T0:Read(Y,1), T1:Read(X,1)
3. T0:Write(A,2), T0:Write(X,1), T0:Read(Y,0), T1:Write(Y,1), T1:Read(X,1)
4. T0:Read(Y,0), T1:Read(X,0), T0:Write(X,1), T1:Write(Y,1), T0:Write(A,2), T1:Write(A,3)
5. T0:Write(X,1), T0:Read(Y,0), T1:Write(Y,1), T1:Read(X,1), T0:Write(A,2)

Answer

1. - **SC:** This type of order is not allowed due to when Y=1 is written by T1, the action of reading Y=0 by T0 is not possible with SC.
   - **TSO:** Not allowed because it allows reordering of writes before reads but not writes after reads. And because Y happens before T1 writes the value where the writing to A happens afterward, it will lead to not following the constraint of TSO.
2. - **SC:** Allowed because due to T0 and T1 perform there operations in same order as the program
   - **TSO:** Allowed because the reads of Y and X are consistent and follow the guideline for TSO for the reads afterward.
3. - **SC:** Allowed
   - **TSO:** Allowed
4. - **SC:** Not allowed because with SC when T1 writes Y=1 the operation that T0 to read Y=0 should not be possible.
   - **TSO:** Now allowed because with TSO the read operation of Y=0 happens before T1 writes Y=1 which makes it not uphold the constraints of TSO.
5. - **SC:** Allowed
   - **TSO:** Allowed

Exercise 2

Question

Machines with relaxed memory consistency typically provide programmers with fence instructions to tighten the ordering of memory instructions. Insert MFENCE (memory fence) instructions in Peterson's algorithm to ensure its correct behavior on a multi-processor system that implements a write buffer with store bypass. Assume that the fence atomically flushes the write buffer. Use as few fence instructions as necessary.

Answer

To utilize MFENCE Peterson's algorithm for ensuring its correct behavior on a multi-processor system. I will base it on this implementation of Peterson's algorithm:

Original Peterson's Algorithm

```
// Shared variables
int turn = 0;
bool flag[2] = {false, false};

// Process 0
while (true) {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1) {
        // busy wait
    }
    // critical section
    flag[0] = false;
    // remainder section
}

// Process 1
while (true) {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0) {
        // busy wait
    }
    // critical section
    flag[1] = false;
    // remainder section
}
```

Modified Peterson's Algorithm with MFENCE

```
// Shared variables
int turn = 0;
bool flag[2] = {false, false};

// Process 0
while (true) {
    flag[0] = true;
    MFENCE;
    turn = 1;
    MFENCE;
    while (flag[1] && turn == 1) {
        // busy wait
    }
    // critical section
    flag[0] = false;
    MFENCE;
    // remainder section
}

// Process 1
while (true) {
    flag[1] = true;
    MFENCE;
    turn = 0;
    MFENCE;
    while (flag[0] && turn == 0) {
        // busy wait
    }
    // critical section
    flag[1] = false;
    MFENCE;
    // remainder section
}
```

This implementation of Peterson's algorithm ensures proper synchronization between two processes by using MFENCE instructions. It prevents memory reordering in systems while ensuring relaxed memory consistency. The MFENCE instructions make sure that the correct visibility of updates to shared variables (flag and turn). This helps and ensures that each process sees the latest changes before entering the critical section.

Exercise 3

Question

A programmer who writes properly synchronized code relative to the high-level language's consistency model (e.g., Java) does not need to consider the architecture's memory consistency model. True or false?

Answer

The above statement is true. In a high-level language like Java, perfectly synchronized code follows the language's memory coherence model, such as

Java's precedence rules, to ensure that operations are performed smoothly. They will be visible to all threads in the correct order. The language runtime handles the complexity of the underlying hardware memory model. Therefore, as long as the programmer uses a synchronization mechanism such as synchronized blocks or volatile variables, They don't need to worry about the architecture's memory consistency model.

Exercise 4

Question

Consider the following variant of an ARM spinlock:

```
MOV R1, #0x1 ; load the 'lock taken' value

try: LDAXR R0, [L] ; load the lock value

CMP R0, #0 ; is the lock free?

STREXEQ R1, R0, [L] ; try and claim the lock

CMPEQ R0, #0 ; did this succeed?

BNE try ; no - try again. . .

; critical section

SYNC ; fence instruction

STR ZR, [L]
```

As in the slides, the lock is taken if variable L is set to 1. Note that the final store is a plain store with no 'release' semantics. Is the fence (SYNC) before the unlock needed? What would happen if it was removed?

Answer

The SYNC instruction on an ARM processor ensures memory ordering. This means that all memory operations within a critical section of an instruction are completed before any operations leading up after the operation. This means that if the SYNC is removed the operation in the critical section not being synced. This means that an operation might not be visible to other threads before the lock is released. In short, because a write-to memory could be delayed without SYNC could lead to the changes not being visible to other operations in a multithreading system.