CS-E4110 Concurrent Programming
Autumn 2016 - Tutorials
Weak Memory Models

2016-11-09
### Dekker’s Algorithm

boolean wantp ← false, boolean wantq ← false  
integer turn ← 1

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>2</td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>3</td>
<td>wantp ← true</td>
<td>wantq ← true</td>
</tr>
<tr>
<td>4</td>
<td>while wantq</td>
<td>while wantp</td>
</tr>
<tr>
<td>5</td>
<td>if turn = 2</td>
<td>if turn = 1</td>
</tr>
<tr>
<td>6</td>
<td>wantp ← false</td>
<td>wantq ← false</td>
</tr>
<tr>
<td>7</td>
<td>await turn = 1</td>
<td>await turn = 2</td>
</tr>
<tr>
<td>8</td>
<td>wantp ← true</td>
<td>wantq ← true</td>
</tr>
<tr>
<td>9</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>10</td>
<td>turn ← 2</td>
<td>turn ← 1</td>
</tr>
<tr>
<td>11</td>
<td>wantp ← false</td>
<td>wantq ← false</td>
</tr>
</tbody>
</table>

**Task 1**

Dekker’s algorithm is a solution to the critical section problem for two threads.

To familiarize yourself with the algorithm, write a Java implementation for it. Use the Java Memory Model to argue that your solution is correct.
**Task 2**

Use the x86-TSO abstract machine model to demonstrate why the following naive C/C++ implementation of Dekker’s algorithm is incorrect and the two threads P and Q may violate mutual exclusion for the critical section.

```c
//Shared variables
bool want_p = false;
bool want_q = false;
int turn = 0;

//Executed by Thread P
void p_body() {
    want_p = true;
    while(want_q) {
        if(turn == 2) {
            want_p = false;
            while(turn != 1) {
                //busy wait
            }
            want_p = true;
        }
    }
    //critical section
    turn = 2;
    want_p = false;
}

//Executed by Thread Q
void q_body() {
    want_q = true;
    while(want_p) {
        if(turn == 1) {
            want_q = false;
            while(turn != 2) {
                //busy wait
            }
            want_q = true;
        }
    }
    //critical section
    turn = 1;
    want_q = false;
}
```

**Task 3**

Introduce memory fences with the `MFENCE` instruction to fix the algorithm from Task 2 for the x86-TSO abstract machine model.
Task 4

This is a demonstration task. Using C++11 low level atomics is not a part of the course scope.

Use C++11 low level atomics to write a portable and correct implementation of Dekker’s algorithm.

Recommended reading:


Task 5

This is a demonstration task. Using C/C++11 low level atomics is not a part of the course scope.

The following C++11 code implements a trivial one-off mechanism for transferring integer values between threads. Explain why it is possible for certain architectures, that for a pair of put() and wait_and_get() calls, the assertion on line 6 does not fail, but the assertion on line 16 does fails. Assume that the compiler will not reorder any instructions.

Use low level C++11 atomics to write an improved version.

```cpp
bool isReady = false;
int value = 0;

//Called from Thread P
void put(int i) {
    assert(i > 0);
    value = i;
    isReady = true;
}

//Called from Thread Q
int wait_and_get() {
    while(!isReady) {
        //busy wait
    }
    assert(value > 0);
    return value;
}
```