Parallel Programming

Recitation Session 6

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

April 15, 2010

Executive Summary

- Formalize understanding of mutual exclusion
- Closer look at `volatile`
- Proof mutual exclusion

Outline

Original Version

```
static int foo;

void bar () {
   foo = 0;
   while (foo != 255)
     ;
}
```

Optimized Version

Compiler will “optimize” the previous version:

```
static int foo;

void bar () {
    foo = 0;
    while (true)
      ;
}
```

⇒ Infinite loop

Volatile

With volatile the loop condition will not be optimized away:

```
volatile static int foo;

void bar () {
    foo = 0;
    while (foo != 255)
      ;
}
```

The variable is re-read from memory each time it is accessed

Java Language Specification

Volatile

A field may be declared volatile, in which case a thread must reconcile its working copy of the field with the master copy every time it accesses the variable. Moreover, operations on the master copies of one or more volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.

Source: http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#36930

Outline

Classroom Exercise 1

Program for thread A (`myid == 0`)

```
// Thread A
public void run() {
    while (true) {
A1:     non_critical section
A2:     while (!(signal.turn == 0)) {}
A3:     // critical_section
A4:     signal.turn = 1;
    }
}
```

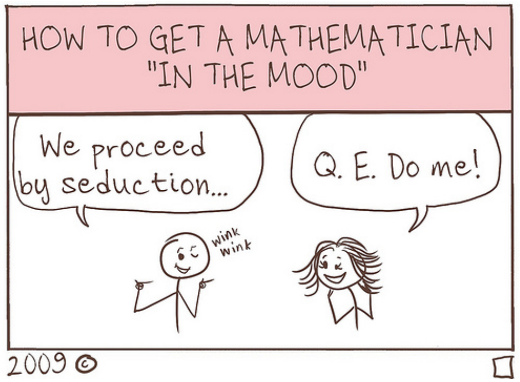

Classroom Exercise: PingPong

Program for thread B (`myid == 1`)

```
// Thread B
public void run() {
    while (true) {
B1:     non_critical section
B2:     while (!(signal.turn == 1)) {}
B3:     // critical_section
B4:     signal.turn = 0;
    }
}
```


Your Task (Now!)

- Show that these threads will never be both in their critical section at the same time.
- You should prove this property in a manner that's similar to the proof given in class.



Source: http://brownsharpie.courtneygibbons.org/?p=1146

Some thoughts on how to proceed

- We introduced already labels for statements and produced two distinct versions for thread A and thread B.
- Now you should formulate the invariant.

Invariants

1. `at(A3)` $\rightarrow$ `turn == 0`
2. `at(B3)` $\rightarrow$ `turn == 1`
3. `not [at(A3) AND at(B3)]`

We use the notation "`at(S)`" to indicate that execution is "at statement (location) `S`" $\Rightarrow$ all previous statements have executed while `S` has not yet started to execute

Proof strategy

- Proof by induction on the execution sequence.
- Base case: does (1) hold at the start of the execution of the program (threads at `A1` and `B1`)?
- Induction step: Assume that (1) holds. Will execution of an additional step invalidate (1)?

Proof (1)

- `at(A1)`: condition (1) is `false` $\Rightarrow$ do not care about signal
- `at(A2)`: condition (1) is `false` $\Rightarrow$ do not care about signal
- `at(A3)`: condition (1) is `true` $\Rightarrow$ `turn == 0`, follows from the fact that `turn` was `0` `at(A2)` and the transition from `A2` $\rightarrow$ `A3` did not change value of `turn`
- `at(A4)`: condition (1) is `false` $\Rightarrow$ do not care about `turn`

Now, we consider:

- `at(B1)`: no change to `turn`
- `at(B2)`: no change to `turn`
- `at(B3)`: no change to `turn`
- `at(B4)`: changes `turn` to `0`

$\Rightarrow$ Invariant (1) is `true`

Proof (2)

Same way. Please do it if you had trouble with proof of (1).

Proof (3): Proof by induction

Induction start trivial

Proof of induction step by contradiction:

- Assume thread A entered CS (A3) at time $t1$
- Assume thread B entered CS (B3) at time $t2$, where $t2 = t1 + \delta$

$\rightarrow$ Contradiction: since we are in A3 signal must be 0 (cannot be 0 and 1 at the same time)

- Assume thread B entered CS (B3) at time $t1$
- Assume thread A entered CS (A3) at time $t2$, where $t2 = t1 + \delta$

$\rightarrow$ Contradiction: since we are in B3 signal must be 1 (cannot be 0 and 1 at the same time)

Classroom Exercise 2: Based on 3rd Variation

```
class Turn {
    // 0 : wants to enter exclusive section
    // 1 : does not want to enter...
    private volatile int flag = 1;

    void request() {
        flag = 0;
    }

    void free() {
        flag = 1;
    }

    int read() {
        return flag;
    }
}
```


Worker

```
class Worker implements Runnable {
    private int myid;
    private Turn mysignal;
    private Turn othersignal;

    Worker(int id, Turn t0, Turn t1) {
        myid = id;
        mysignal = t0;
        othersignal = t1;
    }
    public void run() {
        while (true) {
            mysignal.request();
            while (true) {
                if (othersignal.read() == 1) break;
            }
            // critical section
            mysignal.free();
        }
    }
}
```


Master

```
class Master {
    public static void main(String[] args) {
        Turn gate0 = new Turn();
        Turn gate1 = new Turn();
        Thread t1 =
            new Thread(
                new Worker(0, gate0, gate1)
            );
        Thread t2 =
            new Thread(
                new Worker(1, gate1, gate0)
            );
        t1.start();
        t2.start();
    }
}
```

Worker 0

```java
public void run() {
    while (true) {
A1:
A2:     s0.request();
A3:     while (true) {
            if (s1.read() == 1)
                break;
        }
A4:     // critical section
A5:     s0.free();
    }
}
```


Worker 1

```java
public void run() {
    while (true) {
B1:
B2:     s1.request();
B3:     while (true) {
            if (s0.read() == 1)
                break;
        }
B4:     // critical section
B5:     s1.free();
    }
}
```


Mutual exclusion

Show that this solution provides mutual exclusion.

Invariants

1. `s0.flag == 0` equivalent to
   $(\texttt{at(A3)} \vee \texttt{at(A4)} \vee \texttt{at(A5)})$
2. `s1.flag == 0` equivalent to
   $(\texttt{at(B3)} \vee \texttt{at(B4)} \vee \texttt{at(B5)})$
3. $\texttt{not (at(A4)} \wedge \texttt{at(B4))}$

Induction

Show with induction that (1), (2), and (3) hold.

At the start, `s0.flag == 1` and `at(A1)` ⇒ OK

Induction step: assume (1) is true. Consider all possible moves

- A1 → A2
- A2 → A3
- A3 → A3
- A3 → A4
- A4 → A5
- A5 → A1

Let's look at them one by one:

Induction Step

- A1 → A2: no effect on (1) ⇒ OK
- A2 → A3: (1) holds (`s0.flag == 0` and `at(A3)`) ⇒ OK
- A3 → A3: (1) holds, no change to `s0.flag`, `at(A3)` ⇒ OK
- A3 → A4: (1) holds, no change to `s0.flag`, `at(A4)` ⇒ OK
- A4 → A5: (1) holds, no change to `s0.flag`, `at(A5)` ⇒ OK
- A5 → A1: (1) holds, `s0.flag == 1` and `at(A1)` ⇒ OK

Note that the "⇒ OK" is based on the observation that no action by Thread Worker 1 will have any effect on `s0.flag`

⇒ So (1) holds.

Your turn

Show that (2) holds as well.

Proving (3)

Proof by induction

At the start, `at(A1)` and `at(B1)`, so (3) holds.

Induction step: assume (3) holds and consider possible transitions.

Assume `at(A4)` and consider B3 → B4
(while `Worker0` remains at A4!)
⇒ no other transition is relevant or possible

But since `s0.flag==0` (because of (1)), a transition B3 → B4 is not possible, so (3) remains true.

Proving (3)

Same argument applies if we start with the assumption `at(B4)`.

So no transition will violate (3).

Of course this proof sketch depends on the fact that

- no action of `Worker0` will modify any of the states of `Worker1`, and
- no action of `Worker1` will modify any of the states of `Worker0`

Summary

- Mutual exclusion
- Volatile in Java
- Proofs for mutual exclusion
  - Try to solve assignment 6 – there will be at least one proof at the exam