Синхронизација на повеќе процеси: Race Conditions & Deadlock

Оперативни системи 2014

Аудиториски вежби

Identify Race Condition in Java

- Every object that is **accessed from multiple threads** should be protected from a race condition.
  - **`public static`** fields should always be protected;
  - **`private`** fields of a thread are used only by that thread and shouldn't be protected;
  - Method local variables are visible only to the thread that executes the method, and shouldn't be protected;

Scope and Race Condition

```java
public class ExampleThread extends Thread {
    // would get a reference to some object, which becomes shared
    private IntegerWrapper wrapper;
    // visible by this thread only and is not shared.
    // No need for protection
    private int threadLocalField = 0;
    // can be accessed from other threads and
    // should be protected when used
    public int threadPublishedField = 0;

    public ExampleThread(int init, IntegerWrapper iw) {
        // init is primitive variable and thus is not shared
        threadLocalField = init;
        // this object can be shared, since iw is reference
        this.wrapper = iw;
    }
```

Scope and Race Condition

```
    private void privateFieldIncrement() {...}

    public void publicFieldIncrement() {...}

    public void wrapperIncrement() {...}

    @Override
    public void run() {
        privateFieldIncrement();
        publicFieldIncrement();
        wrapperIncrement();
    }
}
```

Testing Scenario

```
public static void main(String[] args) {
    HashSet<ExampleThread> threads = new HashSet<ExampleThread>();
    IntegerWrapper sharedWrapper = new IntegerWrapper();
    // shuffle the threads using HashSet
    for (int i = 0; i < 100; i++) {
        ExampleThread t = new ExampleThread(0, sharedWrapper);
        threads.add(t);
    }
    for (Thread t : threads) {
        t.start();      // execute in background
    }
    for (ExampleThread t : threads) {// modify thread variables
        /* The private fields are not accessible, and
         thus protected by design :) */
        t.publicFieldIncrement();
        t.wrapperIncrement();
    }
}
```

Reference vs. Primitive Values

- When invoking methods:
  - values of the arguments from primitive types are copied into local variables;
  - arguments that are not primitive are passed as references, and thus the actual object is shared among the threads;

```
public ExampleThread(int init, IntegerWrapper iw) {
    // init is primitive variable and thus is not shared
    threadLocalField = init;
    // this object can be shared, since iw is reference
    this.wrapper = iw;
}
```

Private Fields are Safe

```java
private void privateFieldIncrement() {
    // only this thread can access this field
    threadLocalField++;
    // this variable is visible only in this method (not shared)
    int localVar = threadLocalField;
    try {
        // added to force thread switching
        Thread.sleep(30);
    } catch (InterruptedException ex) {/** DO NOTHING */}
    // check for race condition! Will it ever occur?
    if (localVar != threadLocalField) {
        System.err.println("private-mismatch-%d" + getId());
    } else {
        System.out.println(String.format("[private-%d] %d", getId(),
                threadLocalField));
    }
}
```

Are Public Fields Safe?

```
private void forceSwitch(int sleepTime) {
    try {// added to force thread switching
        Thread.sleep(sleepTime);
    } catch (InterruptedException ex) {/** DO NOTHING */}
}

public void publicFieldIncrement() {
    // increment the public field, and store it to local var
    int localVar = ++threadPublishedField;
    forceSwitch(10);
    // check for race condition! Will it ever occur?
    if (localVar != threadPublishedField) {
        System.err.println("public-mismatch-" + getId());
    } else {
        System.out.println(String.format("[public-%d] %d", getId(),
                threadPublishedField));
    }
}
```

Shared Objects are not Safe!

```
public void wrapperIncrement() {
    // increment the shared variable
    wrapper.increment();
    int localVar = wrapper.getVal();
    forceSwitch(3);
    // check for race condition! It will be common :(
    if (localVar != wrapper.getVal()) {
        System.err.println("wrapper-mismatch-" + getId());
    } else {
        System.out.println(String.format("[wrapper-%d] %d", getId(),
                wrapper.getVal()));
    }
}
```

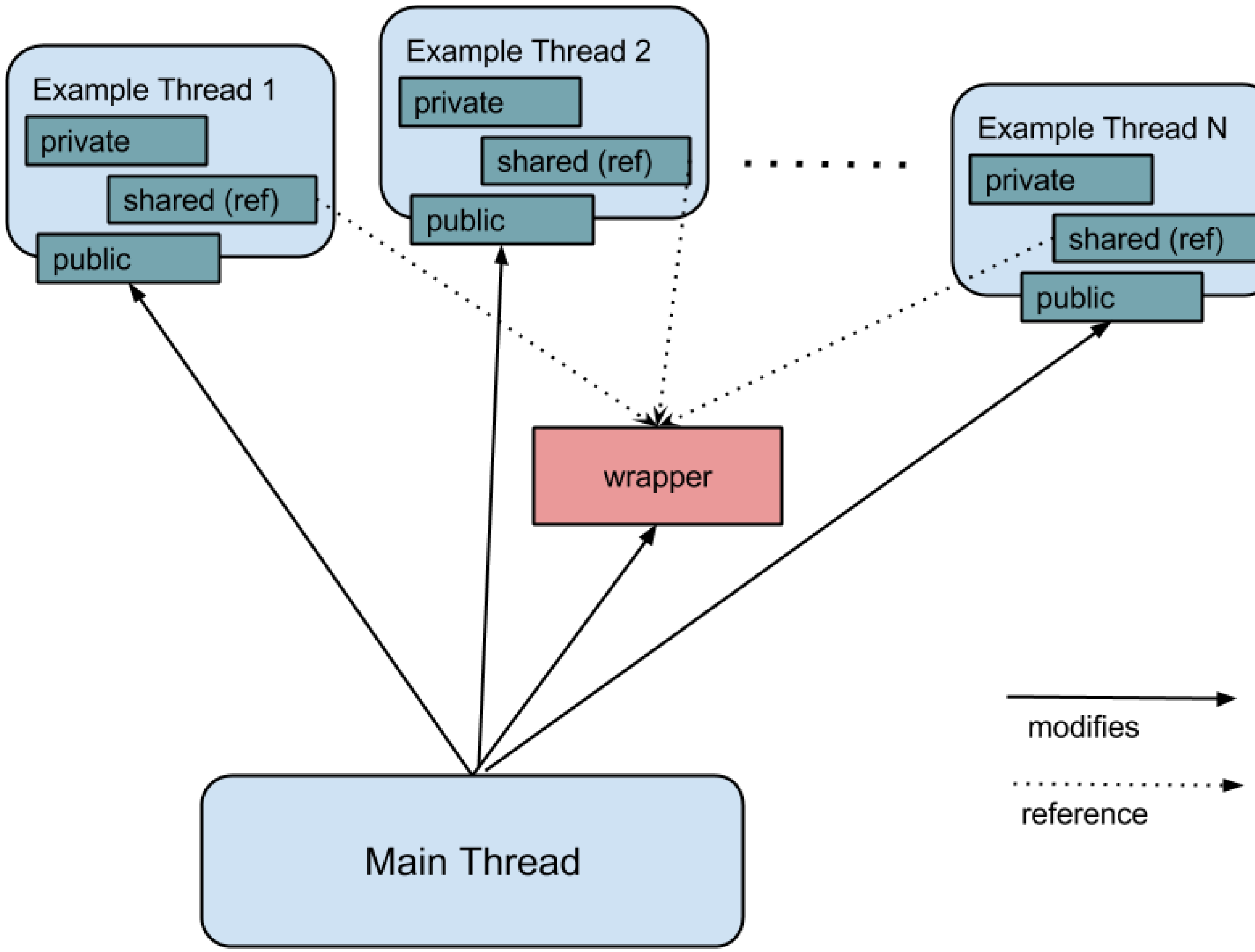
```
public class IntegerWrapper {
    private int val = 0;
    public void increment() { val++; }
    public int getVal() { return val; }
}
```

Test Output

- Total 103 mismatches
  - 2 public mismatches [`forceSwitch(10)`]
    - The public field is modified only by the containing ExampleThread and the main thread;
  - 101 shared object mismatches [`forceSwitch(3)`]
    - 100 ExampleThreads and the main thread are modifying the wrapper;
- Part of the output:

```
[public-50] 1
[private-35] 1
[public-69] 1
public-mismatch-30
wrapper-mismatch-82
[wrapper-31] 103
[public-69] 2
[wrapper-69] 104
[public-20] 1
wrapper-mismatch-24
wrapper-mismatch-66
wrapper-mismatch-17
```

Behind the Scene



Protecting Class Fields

```
public Lock lock = new ReentrantLock();
public Semaphore binarySemaphore = new Semaphore(1);

public void publicFieldSafeIncrement() {
    synchronized (this) {
        publicFieldIncrement();
    }
    // or
    lock.lock();
    publicFieldIncrement();
    lock.unlock();
    // or
    try {
        binarySemaphore.acquire();
        publicFieldIncrement();
    } catch (InterruptedException e) {
    } finally {
        binarySemaphore.release();
    }
}
```

Protecting Shared Objects (is there a difference?)

```java
public static Lock lock = new ReentrantLock();
public static Semaphore binarySemaphore = new Semaphore(1);

public void safeSharedObjectIncrement() {
    synchronized (wrapper) {
        wrapperIncrement();
    }
    // or
    lock.lock();
    wrapperIncrement();
    lock.unlock();
    // or
    try {
        binarySemaphore.acquire();
        wrapperIncrement();
    } catch (InterruptedException e) {
    } finally {
        binarySemaphore.release();
    }
}
```

Conditional Locking: Race Condition

```
// why is this incorrect?
if (wrapper.getVal() <= 5) {
    // forceSwitch(100);
    binarySemaphore.acquire();
    wrapper.increment();
    binarySemaphore.release();
}
```

Conditional Locking: DEADLOCK

```java
// what happens now? - DEADLOCK
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
    binarySemaphore.release();
}
```

Conditional Locking: As it Should Be

```
// finally correct
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
    binarySemaphore.release();
} else {
    binarySemaphore.release();
}
// or
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
}
binarySemaphore.release();
```

Conditional Deadlock

```
// yet another DEADLOCK
Semaphore x = new Semaphore(0);
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
    x.acquire();
} else {
    x.release();
}
binarySemaphore.release();
```

Conditional Deadlock: Fixed

```
// yet another FIXED DEADLOCK
Semaphore x = new Semaphore(0);
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
    // RELEASE CRITICAL REGION BEFORE BLOCKING
    binarySemaphore.release();
    x.acquire();
} else {
    x.release();
    binarySemaphore.release();
}
```

Circular Deadlock (Simplified Example)

```java
// Deadlock scenario
public Semaphore resA=new Semaphore(0);
public Semaphore resB=new Semaphore(0);
public void metodA() throws InterruptedException {
    resA.acquire(); // wait for resource A
    resB.release(); // signal that B is free
}
public void methodB() throws InterruptedException {
    resB.acquire(); // wait for resource B
    resA.release(); // signal that A is free
}
```

Deadlock Solution

- Check the semaphore initialization
  - If the initial conditions are not fixed

```
public Semaphore resA=new Semaphore(1);
public Semaphore resB=new Semaphore(0);
```

- Check the scenario
  - Reorder the locks
  - Check the conditions when lock occurs

Scheduler Dependent Deadlock

- Think twice before blocking inside a synchronized block.
  - Make sure that the unlock call (`resA.release()`) call is not inside a synchronized block with the same monitor;

```
final Object monitor = new Object();
public Semaphore resA = new Semaphore(0);

public void schedulerDependantDeadlock_A() throws InterruptedExceptio
    synchronized (monitor) {
        wrapper.increment();//read or modify shared object
        resA.acquire();
    }
}

public void schedulerDependantDeadlock_B() throws InterruptedExceptio
    synchronized (monitor) {
        resA.release();
        System.out.println(wrapper.getVal()); //shared object access
    }
}
```

Scheduler Dependent Deadlock: Fix

```
public void schedulerDependantDeadlock_A() throws Interrupte
    synchronized (monitor) {
        wrapper.increment();// read or modify shared object
    }
    // block outside of critical region
    resA.acquire();
}
```

Producer – Consumer

- Да се имплементира синхронизација на проблемот со произведувач и потрошувач. Притоа, имаме еден произведувач кој поставува ставки во бафер и произволен број на потрошувачи кои паралелно ги земаат поставените ставки.
- Иницијално баферот е празен.

Producer – Consumer (Scenario)

- Произведувачот врши полнење на баферот со користење на функцијата `state.fillBuffer();`
- Потрошувачот ја зема ставката наменета за него со методот `state.getItem(int id);`
  - Потрошувачот ја зема само ставката наменета за него, по што чека ново полнење на баферот.
- По земањето на ставката од баферот, потрошувачот треба повика `state.decrementNumberOfItemsLeft()` за да каже дека ја земал ставката.
- Потрошувачот кој ќе ја земе последната ставка (го оставил баферот празен) му сигнализира на произведувачот за да го наполни баферот.
  - За проверка дали баферот е празен да се користи `state.isBufferEmpty();`

Producer – Consumer (Constraints)

- Треба да се овозможи повеќе потрошувачи паралелно да може да си ја земат својата ставка од баферот.
  - Паралелно повикување на `state.getItem(int id);`
- Не смее да се повика `state.getItem(int id)` доколку соодветната ставка претходно е земена и не е поставена.
- Не смее да се повика `state.fillBuffer()` доколку има ставки во баферот.
- Повиците `state.isBufferEmpty()` и `state.decrementNumberOfItemsLeft()` го модифицираат тековниот број на ставки во баферот.

Producer – Consumer

- Да се имплементираат методите `init()`, `Producer.execute()` и `Consumer.execute()`, при што ќе се изведе синхронизација за да се извршуваат според дефинираните услови.
- При извршувањето има една инстанца од Producer и повеќе инстанци од Consumer класата кои се извршуваат паралелно.
- Претпоставете дека методот `execute()` и кај двете класи се повикува во бесконечна while јамка.
- Решение:
  - http://code.finki.ukim.mk/course/26/problem/1238/