Синхронизација на повеќе процеси

Оперативни системи

Аудиториска вежба 4

Содржина

Аналогија

Компјутерски системи

- Програмски код
- Реурси
  - Објекти (String, Integer, …)
  - Референци (мемориски локации)
- Процеси / нитки
- Процесот (нитката) го извршува програмскиот код, кој ги корсити ресурсите

Ресторан

- Рецепт
- Фрижидер
  - Намирници (млеко, јајца, …)
  - Позиција (прва полица, втора полица, …)
- Готвач
- Готвачот ги следи инструкциите од рецептот, кои ги користат намирниците од фрижидерот

Сценарио со повеќе нитки и споделени ресурси

Повеќе нитки можат да го извршуваат истиот код, кој ги користи споделените ресурси

Повеќе готвачи може да го следат истиот рецепт и да ги користат истите намирници од фрижидерот

Идентификување на услови на трка во Java

Било кој објект **којшто го пристапуваат повеќе нитки** треба да биде заштитен од услови на трка.

Локалните променливи на секој метод се видливи само на нитката која го извршува методот (не се споделени), и не треба да бидат заштитени:

```
public class Test {
   private static String staticField; // can be shared
   private String classField; // can be shared
   public String example() {
      String localVariable="something"; // never shared
   }
   public String example(String mayBeShared) {
      String shouldBeProtected = mayBeShared;
   }
}
```

Идентификување на услови на трка во Java

```
class Incrementer {
        private int x;
        public Incrementer(int i) { x=i; }
        public void increment() { x++; }
}

private static Incrementer shared=new Incrementer(1);

Thread t=new Thread() {
    private int threadLocal=0;
     public void run() {
         threadLocal++; // should not be synchronized
         shared.increment(); // executed by every thread
         // every thread is changing the value of x,
         // even though it is private
     }
}
```

Идентификување на услови на трка во Java

```
class Composite {
    private Composite c; int x;
    public Composite(int b) { x=b; }
    public Composite(Composite a, int b) { c=a; x=b; }
    public void move() { if(c!=null) c.move(); x++; }
  }

private static Composite shared=new Composite(1);

Thread t=new Thread() {
    public void run() {
      Composite local = new Composite(shared, 1);
     // should be synchronized, because of 'shared'
      local.move();
    }
}
```

Scope на променливите и услови на трка во Java

```
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 access from other threads and should be protected when used
   public int threadPublishedField = 0;

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

Scope на променливите и услови на трка во Java

```
private void privateFieldIncrement() {
        …
}
public void publicFieldIncrement() {
        …
}
public void wrapperIncrement() {
        …
}
public void run() {
    privateFieldIncrement();
    publicFieldIncrement();
    wrapperIncrement();
}
```

Тест сценарио

```
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();
        }
        for(ExampleThread t : threads) { // modify thread variables
            /* The private fields are not accessible, and thus protected by design */
            t.publicFieldIncrement();
            t.wrapperIncrement();
        }
}
```

Референци наспроти примитивни променливи

Кога повикуваме одреден метод:

- Вредностите на аргументите од примитивен тип се копираат во локалните променливи
- Аргументите кои не се примитивни **се праќаат по референца**, односно вистинските објекти се споделуваат меѓу нитките

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

Приватните променливи се безбедни, ОСВЕН ако не се споделени

```
private void privateFieldIncrement() {
        // only this thread can access this field
        threadLocalField++;
        // this variable is only visible 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));
        }
    }
```

Дали public променливите се безбедни?

```
private void forceSwitch(int sleepTime) {
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException ex) { /* DO NOTHING */}
    }
    public void publicFieldIncrement() {
        // increment the public field and store it to localVar
        int localVar = ++threadPublishedField;
        forceSwitch(10);
        // check for race condition, will it ever occur?
        if (localVar != threadPublishedField) {
            System.err.println("public-mismatch-%d" + getId());
        } else {
            System.out.println(String.format("[public-%d] %d",
                getId(), threadPublishedField));
        }
    }
```

Споделените обjекти не се безбедни

```
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-%d" + getId());
        } else {
            System.out.println(String.format("[wrapper-%d] %d",
                        getId(), wrapper.getVal()));
        }
}
class IntegerWrapper {
   private int value = 0;
   public void increment() { this.value ++;}
   public int getVal() { return value;}
}
```

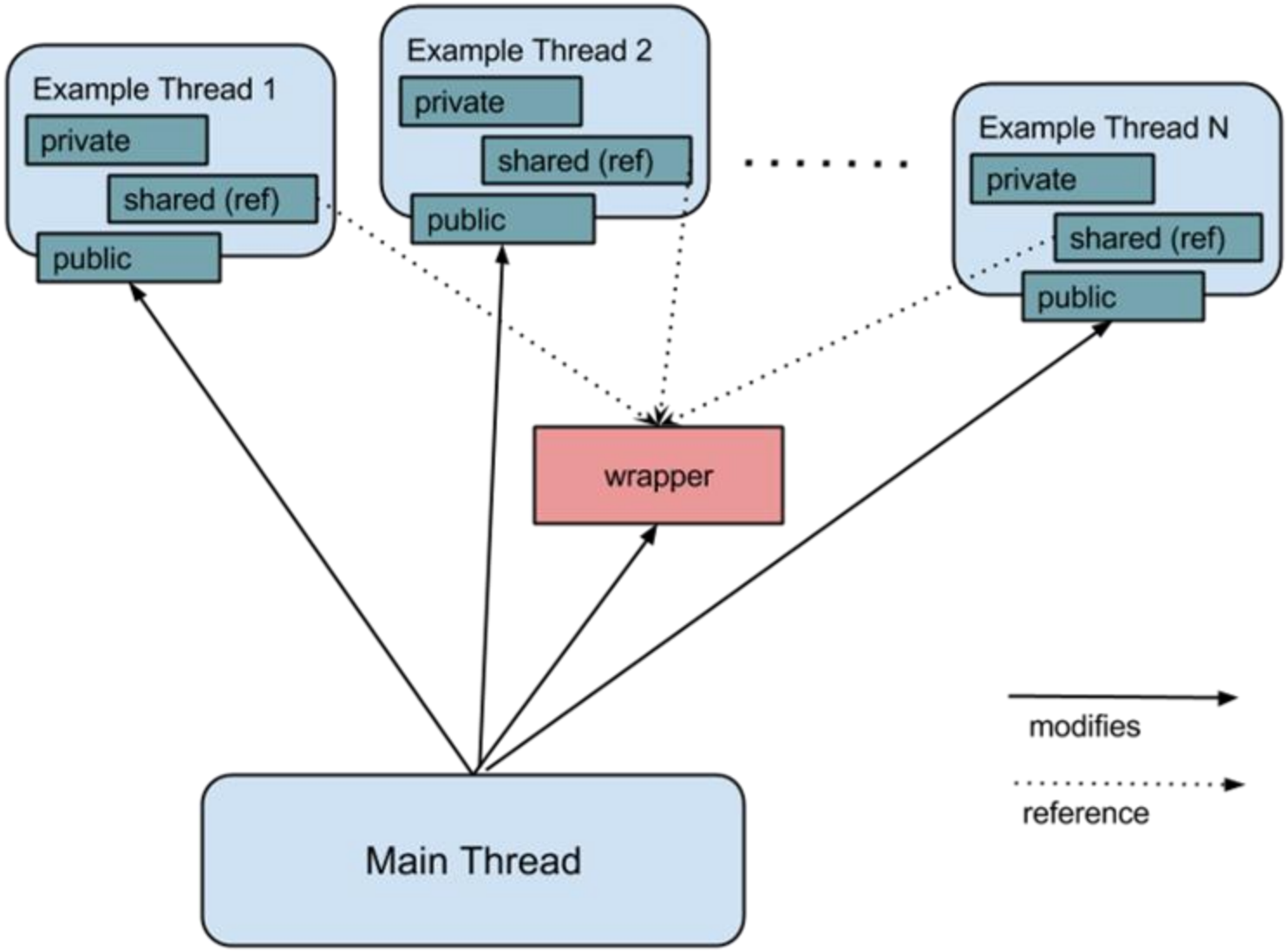
Тестирање на програмата

- Вкупно 103 несовпаѓања
- 2 public несовпаѓања [forceSwitch(10)]
- public променливата е модифицирана само од ExampleThread нитката која што ја содржи и од main нитката
- 101 несовпаѓање на споделената променлива [forceSwitch(3)]
- 100 ExampleThreads нитки и main нитката ја модифицираат споделената wrapper променлива

Пример од излезот од програмата:

```
[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
```

Што се случува во позадина



Заштитување на променливите

```
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 ex) {} finally {
            binarySemaphore.release();
        }
}
```

Заштитување на променливите (разлика ?)

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

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

Условно заклучување: Услови на трка

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

Нитка 1:

If(5<=5)

// pause

// pause

val=6+1

Нитка 2:

// pause

If(5<=5)

val=5+1

// done

Условно заклучување: Deadlock

```
binarySemaphore.acquire();
if (wrapper.getVal() <= 5) {
    wrapper.increment();
    binarySemaphore.release();
}
```

Нитка 1:
// lock
If(5<=5)
val=5+1
// unlock
// done

Нитка 2:
// try lock
// pause
// pause
// pause
// pause
// pause
// pause
// pause
……

Нитка 3:
// try lock
// pause
// pause
// lock
If(6<=5)
// done

Условно заклучување: Како што треба

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

Условен Deadlock

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

Нитка 1:

// lock

If(5<=5)

val=5+1

// wait x

Нитка 2..N:

// try lock

// pause

// pause

// pause

…

Условен Deadlock - Решение

```
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();
}
```

Циркуларен Deadlock

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

public void methodA() 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
}
```

Решение за Deadlock

- Проверка на иницијализацијата на семафорот
  - Исправка на иницијалните услови

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

- Да се провери сценариото
  - Да се промени редоследот
  - Да се проверат условите кога се заклучува регионот

Deadlock кој зависи од распоредувачот

- Размислете два пати пред да блокирате во synchronized блок.
- Осигурајте се дека повикот за ослободување не ресурс (resA.**release()**) не се наоѓа во synchronized блок со истиот монитор

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

public void schedulerDependentDeadlockA() throws InterruptedException {
        synchronized(monitor) {
            wrapper.increment(); // read or modify shared object
            resA.acquire();
        }
    }
    public void schedulerDependentDeadlockB() throws InterruptedException {
        synchronized(monitor) {
            resA.release();
            System.out.println(wrapper.getVal()); // shared object access
        }
    }
```

Deadlock кој зависи од распоредувачот - Решение

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

Задача 1: Producer - Controller

- Проблем: **Producer** – **Controller**, со ограничен број проверки.
- Потребно е да направите оптимизација на додавањата и проверките на податоците од одреден бафер според следните услови:
- Кога се додава податок во баферот, во истиот момент:
  - Не може да има додавање на други податоци
  - Не може да се прави проверка на податоци
- Кога се прави проверка на податоци, во истиот момент:
  - Може да има максимум 10 активни проверки
  - Не може да има додавање пред да завршат сите започнати проверки
- Иницијално во баферот има податоци.

Задача 1: Producer - Controller

- Баферот е претставен со инстанцата buffer од класата Buffer. Притоа може да ги користите следните методи:

  - state.produce()
    - Додава елемент во баферот.
    - Фрла RuntimeException со соодветна порака доколку во истиот момент се врши додавање или проверка на друг податок.
  - state.check()
    - Врши проверка на податок од баферот.
    - Проверува дали во истиот момент се врши додавање или паралелна проверка на повеќе од 10 податоци.

Задача 1: Producer - Controller

- Имплементирајте ги методите execute() од класата Producer и Controller, кои ќе функционираат според претходните правила.
- Тие треба да ги користат методите state.produce() и state.check() за додавање и проверка на податоци од баферот, соодветно.
- Сите семафори и глобални променливи треба да ги дефинирате самите, а нивната иницијализација да ја направите во методот init().
- При имплементацијата на методите, не смеете да додадете try-catch блокови во методите. (Важно при тестирањето)
- При извршувањето има повеќе инстанци од класите Producer и Controller, кои вршат повеќе од едно додавање и проверка, соодветно.
- Додавањата и проверките се стартуваат (скоро) истовремено и паралелно се извршуваат.

Решение 1: Иницијализација

```
static Semaphore accessBuffer;
static Semaphore lock;
static Semaphore canCheck;
static int numChecks = 0;

public static void init() {
    accessBuffer = new Semaphore(1);
    lock = new Semaphore(1);
    canCheck = new Semaphore(10);
}
```

Решение 1: execute() методите

Producer

```
accessBuffer.acquire();
this.buffer.produce();
accessBuffer.release();
```

Controller

```
lock.acquire();
if (numChecks == 0) {
    accessBuffer.acquire();
}
numChecks++;
lock.release();
canCheck.acquire();
buffer.check();
lock.acquire();
numChecks--;
canCheck.release();
if (numChecks==0) {
    accessBuffer.release();
}
lock.release();
```

Решение 2: Иницијализација

```
static Semaphore producer;
static Semaphore controller;

public static void init() {
    producer = new Semaphore(1);
    controller = new Semaphore(10);
}
```

Решение 2: execute() методите

Producer

```
producer.acquire();
// Acquires the given number of permits
// from this semaphore, blocking
// until all are available
controller.acquire(10);
buffer.produce();
controller.release(10);
producer.release();
```

Controller

```
controller.acquire();
buffer.check();
controller.release();
```

Задача 2: Произведувач - Потрошувач

- 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.**

Задача 2: Произведувач - Потрошувач

- 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();

Задача 2: Произведувач - Потрошувач (Ограничувања)

- 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.

Задача 2: Произведувач - Потрошувач

- 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:
  - Кодот е поставен на курсот

Задача 3: Паралелно пребарување и броење

Да се имплементира паралелно пребарување на броеви низ голема низа. Се бара колку пати се појавува даден број s во целата низа. Со цел побрзо пребарување, низата се дели на t број еднакви делови, каде t е бројот на стартувани нитки.
Откако сите нитки ќе завршат со броење, главната нитка печети колку пати е најден бараниот број во низата. На крај нитката која ги нашла најголемиот број на појавувања во својот дел го печати тоа на излез.

Решението е прикачено на курсот

ПРАШАЊА