

Seminář Java

VII (Paralelismus)

Radek Kočí

Fakulta informačních technologií VUT

21. března 2007

- Vlákna
- Multithreading
- Sdílení prostředků
- Synchronizace

Proces

- spuštěný program s vlastním adresovým prostorem

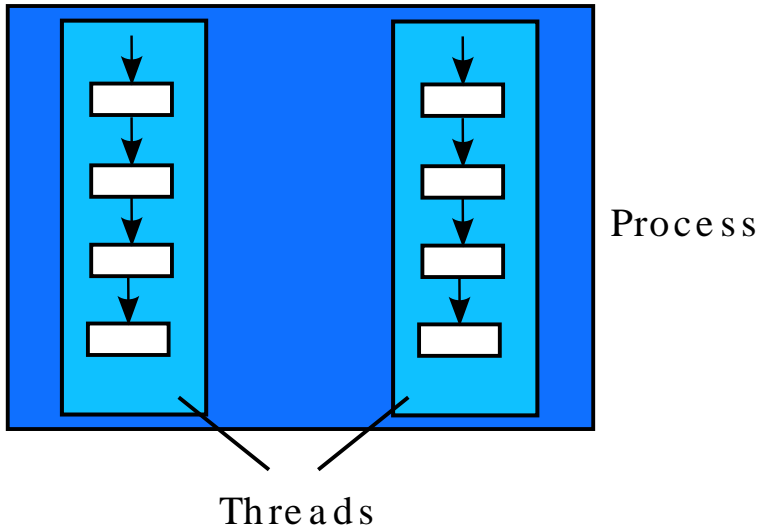
Vlákno (podproces)

- nezávislá vedlejší úloha
- spuštěná v kontextu procesu
- sdílí adresový prostor procesu

Multithreading

- technika rozdělení běhu programu na více podprocesů
- typicky pro oddělení částí programu, které jsou vázané na prostředky

Proces a vlákna



- vlákna jsou reprezentována objekty
- každé vlákno má svůj jedinečný identifikátor (přidělen při vytvoření)
- vlákno může mít svůj název

Balík `java.lang`

- `Thread`
- `Runnable`

Reprezentace vlákna

- instance třídy `Thread` (ev. odvozených tříd)

Vytvoření objektu, který reprezentuje *běh* vlákna

- rozšířením (dědičnost) třídy `Thread`
- implementací rozhraní `Runnable`

Metody

- **start()**
 - inicializace vlákna
 - volá metodu **run()**
 - odvozená třída *nepřekrývá* tuto metodu!
- **run()**
 - kód vlákna
 - odvozená třída *překrývá* tuto metodu
- **sleep(long millis)**
 - uspání vlákna na daný počet milisekund
 - výjimka *InterruptedException*

Ukázka SimpleThread

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```


Ukázka SimpleThread

```
public class TwoThreadsDemo {  
    public static void main (String[] args) {  
        new SimpleThread("Jamaica").start();  
        new SimpleThread("Fiji").start();  
    }  
}
```

Metody

- **run()**
 - kód vlákna
 - implementující třída musí implementovat tuto metodu

Implementující třída

- není vlákno
- informace, že instance třídy definuje chování vlákna
- pro spuštění vlákna potřebuje třídu **Thread**

Ukázka Clock

```
public class Clock extends Applet
    implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void paint(Graphics g) { ... }
        //get the time and convert it to a date
        ...
        g.drawString(dateFormatter.format(date), 5, 10)
    }
```

Ukázka Clock

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try { Thread.sleep(1000); }
        catch (InterruptedException e) {
            //the VM doesn't want us to sleep anymore,
            //so get back to work
        }
    }
}

//overrides Applet's stop method, not Thread's
public void stop() {
    clockThread = null;
}
}
```

Třída Thread nebo rozhraní Runnable

- hodně tříd *potřebuje* dědit (rozšiřovat) jinou třídu

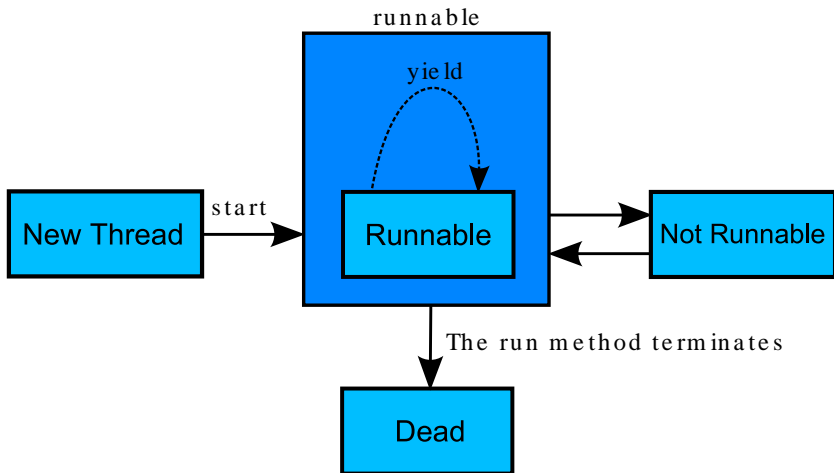
```
public class Clock extends Applet
    implements Runnable {
    ...
}
```

- Java neumožňuje vícenásobnou dědičnost

- ⇒ rozhraní **Runnable**

```
clockThread = new Thread(this, "Clock");
clockThread.start();
```

Životní cyklus vlákna



Vytvoření

- `new Thread(this, "Clock")`

Spuštění

- `start()`
- kód vlákna v metodě `run()` (`Thread` nebo `Runnable`)

Ukončení

- přirozeným doběhnutím metody `run()`
- *existují i jiné metody, ty se ale nedoporučují (deprecated)*
 - `stop()`
 - `destroy()`
 - `suspend()`, `resume()`

Test stavu vlákna

- `isAlive()`
- \Rightarrow `true` pokud bylo vlákno spuštěno a nebylo ukončeno (není dead)
- \Rightarrow `false` pokud vlákno nebylo spuštěno, nebo bylo ukončeno (je dead)

Pozastavení (stav *not runnable*)

- `sleep(...)`
- `wait()`
- při i/o operaci

Uvolnění (stav *runnable*)

- uplynutí doby čekání (viz `sleep(...)`)
- `notify()`, `notifyAll()`
- dokončení při i/o operaci

Chování vláken závisí na jejich interní reprezentaci.

Native threads

- vlákna operačního systému
- více procesorů
- preemtivní plánování

Green threads

- vlákna na úrovni JVM
- jeden procesor
- vlákno se musí přepnout samo
- již nejsou moc běžná

Jedna CPU

- provádění vláken se musí plánovat
- plánovač v Javě je *fixed-priority scheduling*
- plánuje vlákna na základě jejich priority relativně k ostatním vláknům

Priorita

- vlákno *dědí* prioritu vlákna, ve kterém bylo vytvořeno
- čtení/změna priority: `getPriority()`, `setPriority()`
- rozsah: `Thread.MIN_PRIORITY` – `Thread.MAX_PRIORITY`

Plánovač

- vybere vlákno (*runnable*) s nejvyšší prioritou
- pokud je jich více se stejnou prioritou, vybere náhodně/spravedlivě

Vlákno běží dokud se nestane:

- na systému s *time-slicing* uběhne přidělené časové kvantum
- jiné vlákno s vyšší prioritou přejde do stavu *runnable*
- skončí metoda `run()`
- vlákno se vzdá procesoru
- vlákno se dobrovolně vzdá procesoru – zpráva `yield()`
⇒ šance pro ostatní vlákna na *stejně* prioritě

`Thread` `Thread.currentThread()`

- vrátí právě běžící vlákno (objekt)

`setName` / `getName`

- každé vlákno může mít své jméno

`long getId()`

- každé vlákno má svůj jedinečný identifikátor

`Thread.State getState()`

- vrací stav vlákna
- `enum Thread.State`

`enum Thread.State`

- NEW – vlákno pouze vytvořeno
- RUNNABLE – běžící vlákno
- BLOCKED – vlákno čeká na monitoru
- WAITING – vlákno čeká na jiné vlákno / operaci (časově neomezeno)
- TIMED_WAITING – vlákno čeká na jiné vlákno (časově omezeno)
- TERMINATED – vlákno bylo ukončeno

WAITING

- `Object.wait()`
- `Thread.join()`
- `LockSupport.park()`
- I/O operace

TIMED_WAITING

- `Object.wait(long)`
- `Thread.join(long)`
- `Thread.sleep(long)`
- `LockSupport.parkNanos(long)`
- `LockSupport.parkUntil(long)`

`java.util.concurrent.locks.LockSupport`

`java.lang.ThreadGroup`

- vlákno patří vždy k nějaké skupině
- existuje implicitní systémová skupina
- skupiny tvoří stromovou hierarchii
- příslušnost ke skupině je neměnná
- vlákno implicitně "dědí" skupinu vytvářejícího vlákna
- **`Thread.currentThread().getThreadGroup()`**

Problém producent–konzument

- jedno vlákno (producent) zapisuje na sdílené místo data
- druhé vlákno (konzument) tato data čte
- operace zápis/čtení se musí střídat!

Ukázkový příklad

- třída **Producer** – producent
- třída **Consumer** – konzument
- třída **CubbyHole** – sdílený prostor (metody get a put)

Producent–konzument: Producent

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;

    public Producer(CubbyHole c) {
        cubbyhole = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Producent-konzument: Konzument

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;

    public Consumer(CubbyHole c) {
        cubbyhole = c;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
        }
    }
}
```

Producent–konzument: CubbyHole

```
public class CubbyHole {  
    private int contents;  
  
    public int get() {  
        return contents;  
    }  
  
    public int put(int value) {  
        contents = value;  
    }  
}
```

Producent–konzument: možné problémy

Producent je rychlejší

- konzument může "propásnout" čísla

Konzumer je rychlejší

- konzument čte stejné číslo vícekrát

⇒ race condition

- dvě (příp. více) vlákna čtou/zapisují sdílená data; výsledek závisí na časování (jak jsou vlákna plánována)

⇒ race condition

- dvě (příp. více) vlákna čtou/zapisují sdílená data; výsledek závisí na časování (jak jsou vlákna plánována)

Nutná synchronizace:

- vlákna nesmí současně přistoupit ke sdílenému objektu
- producent musí indikovat, že hodnota je připravena; nezapisuje dokud si hodnotu nepřečte konzument
- konzument musí indikovat, že přečetl hodnotu; nečte dokud producent nezapíše novou hodnotu

Monitor

- kritické sekce
- uzamčení objektu při přístupu ke kritické sekci
- pokud je objekt uzamčen, nikdo jiný nemůže přistupovat je kritickým sekcím objektu
- odemknutí objektu při výstupu z kritické sekce

Monitor v Javě (intrinsic lock)

- součástí každého objektu (třída **Object**)
- klíčové slovo **synchronized**

Kritická sekce v Javě

- metoda
- blok

Producent-konzument: CubbyHole

```
public class CubbyHole {
    private int contents;

    private boolean available = false;
    public synchronized int get() {
        //CubbyHole locked by the Producer
        ...
        // CubbyHole unlocked by the Producer
    }

    public synchronized int put(int value) {
        // CubbyHole locked by the Consumer
        ...
        // CubbyHole unlocked by the Consumer
    }
}
```


Zámek: reentrantní

```
public class Reentrant {  
    public synchronized void a() {  
        b();  
        System.out.println("here I am, in a()");  
    }  
  
    public synchronized void b() {  
        System.out.println("here I am, in b()");  
    }  
}
```

Producent–konzument: CubbyHole

```
public class CubbyHole {
    private int contents;
    private boolean available = false;
    public synchronized int get() {
        if (available == true) {
            available = false;
            return contents;
        }
    }
    public synchronized int put(int value) {
        if (available == false) {
            available = true;
            contents = value;
        }
    }
}
```

Producent-konzument: CubbyHole

```
public synchronized int get() {
    while (available == false) {
        try {
            // wait for Producer to put value
            wait();
        } catch (InterruptedException e) { }
    }
    available = false;

    // notify Producer that value has been
    // retrieved
    notifyAll();

    return contents;
}
```

Producent-konzument: CubbyHole

```
public synchronized void put(int value) {
    while (available == true) {
        try {
            // wait for Consumer to get value
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;

    // notify Consumer that value has been set
    notifyAll();
}
```

`wait()`

- aktuální vlákno bude čekat, dokud se nezavolá `notify()` (`notifyAll()`) nad objektem

`wait(long timeout)`

- ...nebo neuplyne timeout

`notify()`

- vzbudí jedno vlákno čekající na monitoru objektu

`notifyAll()`

- vzbudí všechna vlákna čekající na monitoru objektu

`wait(), ...`

- před suspendováním vlákna se odemkne monitor
- pokud vlákno vlastní více monitorů, odemkne se pouze monitor daného objektu

`notify(), ...`

- řízení není okamžitě předáno vzbuzenému vláknu

Tyto metody může volat pouze to vlákno, které je vlastníkem monitoru

- vstoupení do kritické sekce (`synchronized`) – metoda, blok

Při uzamčení objektu se zvýší náklady na režii

- uvážit změnu návrhu
- použít zámek (synchronized) na konkrétní objekt
- neodbyť souběžný přístup synchronizací všech metod

Blok jako kritická sekce

- stejný princip synchronizace
- metody se nemusí deklarovat jako synchronizované
- deklaruje se objekt, jehož monitor se použije při obsluze kritické sekce

```
synchronized(cybbyhole) {  
    cybbyhole.put(i);  
}  
synchronized(cybbyhole) {  
    value = cybbyhole.get();  
}
```


Operace `interrupt()` (Thread)

- je nastaven příznak
- přerušitelné operace (`sleep()`, ...) testují tento příznak

Vlákno je běžící

- pouze se nastaví příznak
- lze otestovat (`isInterrupted()`)

Vlákno je čekající

- je nastaven příznak
- vlákno se rozběhne a vygeneruje se výjimka (`InterruptedException`)

Viditelnost sdílené proměnné

- změna hodnoty se nemusí projevit okamžitě (optimalizace)
- ⇒ klíčové slovo `volatile`

Proč nepoužívat `stop()` a `suspend()`?

`stop()`

- uvolní všechny monitory blokované vláknem
- nebezpečí přístupu k objektům v nekonzistentním stavu
- ⇒ přirozené ukončení metody `run()`

`suspend()`, `resume()`

- suspenduje/uvolní vlákno
- suspendované vlákno drží monitor (viz kritická sekce); vlákno, které ho má uvolnit (viz `resume()`), musí vstoupit do této sekce ⇒ dead-lock
- ⇒ `wait()`, `notify()`

více na

<http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>

`java.util.concurrent.locks`

- rozhraní `Lock`, `Condition`, `ReadWriteLock`
- třídy `ReentrantLock`, `ReentrantReadWriteLock`
- třída `ReentrantReadWriteLock.ReadLock`
- třída `ReentrantReadWriteLock.WriteLock`

```
public interface Lock {  
    void lock();  
    void lockInterruptibly()  
        throws InterruptedException;  
    boolean tryLock();  
    boolean tryLock(long timeout, TimeUnit unit)  
        throws InterruptedException;  
    void unlock();  
    Condition newCondition();  
}
```

ReentrantLock implements Lock

```
Lock lock = new ReentrantLock();

lock.lock();
try {
    ...
} finally {
    lock.unlock();
}
```

Lock a Condition

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    long awaitNanos(long nanosTimeout)
        throws InterruptedException;
    void awaitUninterruptibly()
        throws InterruptedException;
    long awaitUntil(Date deadline)
        throws InterruptedException;
    void signal();
    void signalAll();
}
```

Intrinsic locks

- méně flexibilní
- JVM ví o vztahu vlákna a zámku

Explicit locks

- flexibilní
- JVM neví o vztahu vlákna a zámku

`java.util.concurrent`

- `BlockingQueue`
- `SynchronousQueue`
- `Semaphore`
- ...

```
private BlockingQueue cubbyhole;  
cubbyhole.put(i);
```

...

Statické metody třídy `Collections`

- `XXX synchronizedXXX(XXX c);`
- `Collection synchronizedSet(Collection c);`
- `Map synchronizedMap(Map c);`
- ...

Synchronizované kolekce

```
public class SynchronDemo {
    public static void main(String[] args) {
        List seznam = new ArrayList();
        seznam.add(new Integer(20));
        ...
        Iterator i = seznam.iterator();

        // Simulace souběžné akce jiného vlákna!!
        seznam.remove(new Integer(20));
        // ...

        Integer c = (Integer) i.next();
    }
}
```

Vyjimka **ConcurrentModificationException**

Synchronization wrapper

```
public class SynchronDemo2 {
    public static void main(String[] args) {
        List seznam = Collections.
            synchronizedList(new ArrayList());

        seznam.add(new Integer(20));

        synchronized(seznam) {
            Iterator i = seznam.iterator();
            Integer c = (Integer) i.next();
        }
    }
}
```

SynchronizedCollection

```
static class SynchronizedCollection<E> ... {  
    Collection<E> c;    // Backing Collection  
    Object      mutex; // Object on which to  
                        // synchronize  
  
    SynchronizedCollection(Collection<E> c) {  
        this.c = c;  
        mutex = this;  
    }  
    SynchronizedCollection(Collection<E> c,  
                            Object mutex)  
    {  
        this.c = c;  
        this.mutex = mutex;  
    }  
}
```

SynchronizedCollection

```
public boolean add(E o) {  
    synchronized(mutex) {return c.add(o);}  
}
```

```
public Iterator<E> iterator() {  
    return c.iterator();  
    // Must be manually synched by user!  
}
```

`http://java.sun.com/docs/books/tutorial/essential/
http://www.programming-x.com/programming/
brian-goetz.html
http://www.javaworld.com`