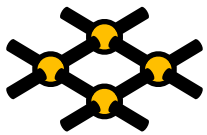
Vláknové programování
část V

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192

2014–03–25

Přehled přednášky

```
long promenna = 10000000L;
```

Paměťový model Javy

- *happens-before*
  - částečné uspořádání

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock.[3]

Volatile variable rule. A write to a volatile field *happens-before* every subsequent read of that same field.[4]

Thread start rule. A call to `Thread.start` on a thread *happens-before* every action in the started thread.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.

Interruption rule. A thread calling `interrupt` on another thread *happens-before* the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If *A happens-before B*, and *B happens-before C*, then *A happens-before C*.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

- Piggybacking
  - spojení happens-before pravidla s jiným pravidlem, obvykle monitorem nebo `volatile`
  - raději nepoužívat
  - příklad: `http://kickjava.com/src/java/util/concurrent/FutureTask.java.htm`
    - postaveno na `tryReleaseShared` happens-before `tryAcquireShared`
    - kombinace volatilní proměnné `runner`, do které `tryReleaseShared` zapisuje s program order

Paměťový model Javy

```
        void innerSet(V v) {
2           for (;;) {
                int s = getState();
4               if (ranOrCancelled(s))
                    return;
6               if (compareAndSetState(s, RAN))
                    break;
8           }
            result = v;
10          releaseShared(0);
            done();
12      }
```

```
        V innerGet() throws InterruptedException JavaDoc, ExecutionException Jav
2           acquireSharedInterruptibly(0);
            if (getState() == CANCELLED)
4               throw new CancellationException JavaDoc();
            if (exception != null)
6               throw new ExecutionException JavaDoc(exception);
            return result;
8       }
```

Paměťový model Javy

```
1         protected int tryAcquireShared(int ignore) {
              return innerIsDone()? 1 : -1;
3         }

5         /**
           * Implements AQS base release to always signal after setting
7          * final done status by nulling runner thread.
           */
9         protected boolean tryReleaseShared(int ignore) {
              runner = null;
11            return true;
          }
13
          boolean innerIsCancelled() {
15            return getState() == CANCELLED;
          }
17
          boolean innerIsDone() {
19            return ranOrCancelled(getState()) && runner == null;
          }
```

Paměťový model Javy

- líná inicializace

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }

    static class Resource {
    }
}
```

Paměťový model Javy

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }

    static class Resource {
    }
}
```

- líná inicializace thread-safe

Paměťový model Javy

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() {
        return resource;
    }

    static class Resource {
    }
}
```

- „dychtivá“ inicializace
- využívá skutečnosti, že statické inicializátory jsou vždy dokončeny před použitím třídy

Paměťový model Javy

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceFactory.ResourceHolder.resource;
    }

    static class Resource {
    }
}
```

- idiom líné inicializace s použitím holder class
- využívá líné inicializace tříd

Paměťový model Javy

```
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
```

Paměťový model Javy

```
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
```

- Double Checked Locking anti-pattern
- pomíjí možnost, že **`resource`** je v nedefinovaném stavu
- od Java 5.0 možno spravit použitím **`volatile`**
- nepoužívat
  - ani v C/C++!

Java SWING

- Multiplatformní GUI v Javě
- SWING single-thread rule
  - všechny prvky mohou být vytvářeny, měněny a dotazovány pouze z vlákna obsluhujícího události
  - `SwingUtilities.isEventDispatchThread` – kontrola, zda jsme ve vlákně obsluhující události
  - `SwingUtilities.invokeLater` – předávání `Runnable` do vlákna obsluhujícího události
  - `SwingUtilities.invokeAndWait` – předávání `Runnable` do vlákna obsluhujícího události a zablokuje se do dokončení akce
  - callbacky se řeší pomocí akcí action listener z vlákna obsluhujícího události
  - dlouho běžící callbacky je možno odštípnout do nového vlákna (přímo nebo přes Executory)

Java SWING

- Inicializace

```
public class JavaGUI {

    public static void main(String[] args) {
        CounterDialog dialog = new CounterDialog();
        dialog.setSize(400,300);
        dialog.setVisible(true);
    }

}
```

Java SWING

- Předávání vláknu událostí SWINGu

```
    @Override
2   public void run() {
        for (int i = 0; i <= 1000; i++) {
4           if (shouldShutdown.get()) {
                break;
6           }
            final String labelText = String.valueOf(i);
8           javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
10                  counterLabel.setText(labelText);
                }
12          });
            try {
14              Thread.sleep(1000);
            } catch (InterruptedException ignored) {
16          }
        }
18      runButton.setText("Run");
        runButton.setEnabled(true);
20  }
```

Java SWING

- Callbacky

```
1       buttonRun.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
3               onRun();
            }
5       });
```

```
    private void onRun() {
2       if (!isRunning.get()) {
            buttonRun.setText("Stop");
4           isRunning.set(true);
            if (counterThread != null) {
6               try {
                    counterThread.join();
8               } catch (InterruptedException ignored) {
                }
10          }
            counterThread = new CounterThread(counterLabel, buttonRun);
12          counterThread.start();
        } else {
14          buttonRun.setEnabled(false);
            counterThread.requestShutdown();
16          counterThread.interrupt();
            isRunning.set(false);
18      }
    }
```

Java Native Interfaces

- Volání nativních metod z Javy
- Struktura JNI volání

```
/* C */
JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj, jstring javaString)
{
    //ziskani nativniho retezce z javaString
    const char *nativeString = (*env)->GetStringUTFChars(env, javaString, 0);
    ...
    //nezapomenout uvolnit!
    (*env)->ReleaseStringUTFChars(env, javaString, nativeString);
}

// C++
JNIEXPORT void JNICALL Java_ClassName_MethodName
  (JNIEnv *env, jobject obj, jstring javaString)
{
    //ziskani nativniho retezce z javaString
    const char *nativeString = env->GetStringUTFChars(javaString, 0);
    ...
    //nezapomenout uvolnit!
    env->ReleaseStringUTFChars(javaString, nativeString);
}
```

Vlákna a JNI

- Ukazatel na `JNIEnv` je platný pouze z vlákna, jemuž je přiřazen
  - nelze předávat mezi vlákny
  - stejný při opakovaných voláních v témže vlákně
- Lokální reference nesmí opustit vlákno
  - lokální reference jsou platné pouze v rámci daného volání
    - nelze se je uschovávat ve `static` proměnných
  - převést na globální, pokud je třeba (`NewGlobalRef`)
  - globální reference vylučují objekt z garbage collection
  - existují slabé lokální reference (`NewWeakGlobalRef`)
    - umožňují garbage collection odkazovaného objektu
    - potřeba kvůli class unloading
    - musí se kontrolovat, zda odkazovaný objekt existuje (`IsSameObject` s NULL parametrem)

Vlákna a JNI

```
static jclass stringClass = NULL;
...
if (stringClass == NULL) {
    jclass localRefCls =
        (*env)->FindClass(env, "java/lang/String");
    if (localRefCls == NULL) {
        return NULL; /* exception thrown */
    }
    /* Create a global reference */
    stringClass = (*env)->NewGlobalRef(env, localRefCls);

    /* The local reference is no longer useful */
    (*env)->DeleteLocalRef(env, localRefCls);

    /* Is the global reference created successfully? */
    if (stringClass == NULL) {
        return NULL; /* out of memory exception thrown */
    }
}
...
// potreba explicitne mazat
if (terminate) {
   (*env)->DeleteGlobalRef(env, stringClass);
}
```

Zdroj: JNI Book

Vlákna a JNI

- Použití monitorů
  - vždy monitor uvolnit

```
synchronized (obj) {
    ...                     // synchronized block
}
```

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) ...;
...
if ((*env)->ExceptionOccurred(env)) {
    ... /* exception handling */
    /* remember to call MonitorExit here */
    if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;
}
... /* Normal execution path. */
if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;
```

Zdroj: JNI Book

Vlákna a JNI

- Wait/Notify
  - generické volání metod (`GetMethodID`, `CallVoidMethod`)
  - potřeba držet monitor

```
1  /* precomputed method IDs */
   static jmethodID MID_Object_wait;
3  static jmethodID MID_Object_notify;
   static jmethodID MID_Object_notifyAll;
5
   void
7  JNU_MonitorWait(JNIEnv *env, jobject object, jlong timeout) {
       (*env)->CallVoidMethod(env, object, MID_Object_wait, timeout);
9  }

11 void
   JNU_MonitorNotify(JNIEnv *env, jobject object) {
13     (*env)->CallVoidMethod(env, object, MID_Object_notify);
   }
15
   void
17 JNU_MonitorNotifyAll(JNIEnv *env, jobject object) {
       (*env)->CallVoidMethod(env, object, MID_Object_notifyAll);
19 }
```

Zdroj: JNI Book

Vlákna a JNI

- Explicitní získání `JNIenv` pro stávající vlákno
  - např. callback volaný OS
  - odkaz na JavaVM lze předávat mezi voláními a vlákny
  - získání odkazu např. `JNI_GetCreatedJavaVMs` nebo `GetJavaVM`

```
1  JavaVM *jvm; /* already set */

3  f() {
       JNIEnv *env;
5      (*jvm)->AttachCurrentThread(jvm, (void **)&env, NULL);
       ... /* use env */
7  }
```

Zdroj: JNI Book

Vlákna a JNI

- Mapování vláknového modelu OS a JVM
  - záleží, zda pro danou platformu JVM podporuje nativní vlákna
  - závisí od dané platformy i od daného JVM
- Můžeme implementovat v Javě afinitu k CPU
  - závisí od dané platformy i od daného JVM
  - `sched_setaffinity(gettid(), ...)`
  - v praxi může pro danou platformu dobře fungovat