

Vláknové programování

část V

Lukáš Hejtmánek, Petr Holub

`{xhejtman, hopet}@ics.muni.cz`



Laboratoř pokročilých síťových technologií

PV192
2012-03-20

Přehled přednášky

Úlohy a vlákna

Executors, Thread Pools a Futures

Ukončování a přerušování

ThreadPoolExecutors Revisited

Java NIO

Úlohy a vlákna

- Úloha vs. vlákno
 - úloha – co se vykonává (**Runnable**, **Callable**)
 - vlákno – kdo úlohu vykonává (Executor/Future/TPE/...)
- Oddělení úloh od vláken
 - úloha nesmí předpokládat nic o chování vlákna, které ji vykonává
 - Politika ukončení vs. politika přerušení

(příklady povětšinou převzaty z JCiP, Goetz)



Executors, Thread Pools

- Koncept vykonavatelů kódu: Executors
 - vykonávají se objekty implementující Runnable
 - různé typy Executors
- ExecutorService přidává
 - schopnost zastavit vykonávání
 - schopnost vykonávat Callable<V>, nikoli pouze Runnable()
 - vracet objekty representované jako Future
- ThreadPoolExecutor
 - všeobecně použitelný executor, jednoduché API
 - minimální i maximální počet vláken
 - recyklace vláken
 - likvidace nepoužívaných vláken

Runnable vs. Callable

- Interface Runnable

- implementuje úlohu
- lze použít s konstruktorem třídy Thread
 - ◆ konceptuálně čistější přístup: nerozšiřujeme třídu, kterou vlastně rozšiřovat nechceme
- použití i v hlavním vlákne

```
public class PrikladRunnable {
2   static class RunnableVlakno implements Runnable {
        public void run() {
4           System.out.println("Tu je vlakno.");
        }
6     }

8     public static void main(String[] args) {
        System.out.print("Startuji vlakno: ");
10        new Thread(new RunnableVlakno()).start();
        System.out.println("hotovo.");
12        System.out.println("Spoustim primo v hlavnim vlakne: ");
        new RunnableVlakno().run();
14    }
}
```

Runnable vs. Callable

- Interface Callable<V>

- na rozdíl od Runnable může vracet výsledek (typu V) a vyhodit výjimku

```
1 import java.util.concurrent.Callable;
3 public class PrikladCallable {
4     static class CallableVlakno implements Callable<String> {
5         public String call() throws Exception {
6             return "Retezec z Callable";
7         }
8     }
9
10    public static void main(String[] args) {
11        try {
12            String s = new CallableVlakno().call();
13            System.out.println(s);
14        } catch (Exception e) {
15            System.out.println("Chytil jsem vyjimku");
16        }
17    }
18 }
```

Executors

- Typy Executorů
 - SingleThreadExecutor
 - ◆ sekvenční vykonávání úloh
 - ◆ pokud vlákno selže, pokračuje se vykonáváním následujícího
 - ScheduledThreadPool
 - ◆ zpožděné či opakované vykonávání vláken
 - FixedThreadPool
 - ◆ používá pevný počet vláken
 - CachedThreadPool
 - ◆ vytváří nová vlákna dle potřeby
 - ◆ opakovaně používá existující uvolněná vlákna
 - ScheduledExecutorService
 - ◆ implementace spouštění s definovaným zpožděním a opakovaného spouštění
 - ◆ <http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html>
 - Executors factory
 - ◆ implementace vlastních typů Executorů
 - ◆ <http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html>

Executors

```

import java.util.concurrent.*;
2 import java.util.Random;

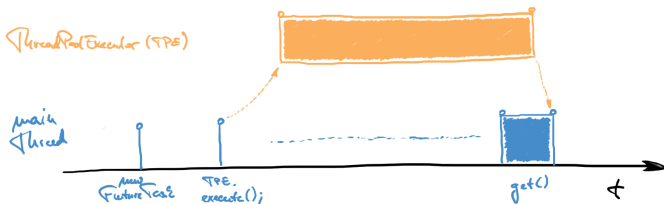
4 public class TPE {
    public static void main(String[] args) {
6         final Random random = new Random();
            // forkbomba: ;- )
8         // ExecutorService executor = Executors.newCachedThreadPool();
            ExecutorService executor = Executors.newFixedThreadPool(
10             Runtime.getRuntime().availableProcessors()-1);
            for (int i = 0; i < 100; i++) {
12                 executor.execute(new Runnable() {
                    public void run() {
14                         int max = random.nextInt();
                            for(int j = 0; j < max; j++) { j += 2; j--; }
16                         System.out.println("Dobehlo vlakno s max = " + max);
                                }
18                 });
            }
20         try {
            Thread.sleep(10000);
22             executor.shutdown();
            executor.awaitTermination(1000, TimeUnit.SECONDS);
24         } catch (InterruptedException e) {
            }
26     }
}

```


Futures

- Princip:

- někdy v budoucnu bude volající potřebovat výsledek výpočtu X
 - v době, kdy si volající řekne o výsledek výpočtu X : (a) výsledek je okamžitě vrácen, pokud je již k dispozici, nebo (b) volající se zablokuje, výsledek se dopočítá a vrátí, volající se odblokuje
- H. Baker, C. Hewitt, "The Incremental Garbage Collection of Processes". *Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12*. August 1977. podobný koncept
- D. Friedman. "CONS should not evaluate its arguments". S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257-284. Edinburgh University Press, Edinburgh. Also available as *Indiana University Department of Computer Science Technical Report TR44*. 1976



Futures a ThreadPoolExecutor

```
1 import java.util.concurrent.*;
3 public class Futures {
4     public static class StringCallable implements Callable {
5         public String call() throws Exception {
6             System.out.println("FT: Pocitam.");
7             Thread.sleep(5000);
8             System.out.println("FT: Vypocet hotov.");
9             return "12345";
10        }
11    }
12    public static void main(String[] args) {
13        ThreadPoolExecutor tpe = new ThreadPoolExecutor(2, 8, 60L,
14            TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
15        FutureTask ft = new FutureTask(new StringCallable());
16        System.out.println("main: Poustim vypocet.");
17        tpe.execute(ft);
18        // alternativa: Future ft = tpe.submit(new StringCallable());
19        try {
20            System.out.println("main: Chci vysledek.");
21            String s = (String) ft.get();
22            System.out.println("main: Mam vysledek: " + s);
23            tpe.shutdown();
24            tpe.awaitTermination(1, TimeUnit.MINUTES);
25        } catch (InterruptedException e) {}
26        catch (ExecutionException e) {}
27    }
28 }
```



Futures vs. CompletionService

- Problém: máme řadu odložených úloh (Future) a potřebujeme je v pořadí dokončení, nikoli zaslání
 1. opakované procházení seznamu a používání `get(0, TimeUnit.SECONDS);`
 2. použijeme `CompletionService`
- `CompletionService`
 - kombinuje `Executor` a `BlockingQueue`
 - `submit()` – vkládáme úlohy pomocí
 - `take()` a `poll()` – vybíráme dokončené úlohy
 - při prázdné frontě dokončných úloh se `take()` blokuje, `poll()` vrací `null`

Futures vs. CompletionService

```
1  ArrayList<FileData> stahniSoubory(ArrayList<String> list) {
2      ArrayList<FileData> ald = new ArrayList<FileData>();
3      CompletionService<FileData> completionService =
4          new ExecutorCompletionService<FileData>(
5              new ThreadPoolExecutor(1, 10, 60, TimeUnit.SECONDS,
6                  new LinkedBlockingQueue<Runnable>()));
7      for (final String s : list) {
8          completionService.submit(new Callable<FileData>() {
9              public FileData call() throws Exception {
10                 FileData fd = new FileData();
11                 fd.s = s; fd.data = getFile(s);
12                 return fd;
13             }
14         });
15     }
16     try {
17         for (int i = 0, size = list.size(); i < size; i++) {
18             Future<FileData> f = completionService.take();
19             ald.add(f.get());
20         }
21     } catch (InterruptedException e) {
22         Thread.currentThread().interrupt();
23     } catch (ExecutionException e) { launderThrowable(e.getCause()); }
24     return ald;
}
```



Futures vs. CompletionService

```
public static RuntimeException launderThrowable(Throwable t) {  
2     if (t instanceof RuntimeException)  
        return (RuntimeException) t;  
4     else if (t instanceof Error)  
        throw (Error) t;  
6     else  
        throw new IllegalStateException("Not unchecked", t);  
8 }  
}
```



Ukončování a přerušování pro pokročilé

- Kooperativní ukončování úloh a přerušování vláken
 - příznakem proměnné
 - přerušením – interrupt
 - `Thread.stop` – deprecated
- Důvody ukončení úloh
 - uživatelem vyvolané ukončení úlohy (GUI, JMX)
 - časově omezené úlohy
 - události uvnitř – několik úloh hledá řešení paralelně, jedna ho najde
 - externí chyby
 - ukončení aplikace

Ukončování a přerušování pro pokročilé

- Politika ukončování (cancellation policy)
 - vývojářem specifikováno pro každou **úlohu** (JavaDoc)
 - jak? – jak se vyvolává ukončení?
 - kdy? – kdy je možné vlákno ukončit?
 - co? – co bude třeba udělat před ukončením?
- Ukončování příznakem a/nebo přerušením?


Přerušení – interrupt

- Mechanismus zasílání zprávy mezi vlákny
 - sémanticky definováno jen jako signalizace mezi vlákny
 - nastavení příznaku

```

1  public class Thread {
      public void interrupt() {...}
3     public boolean isInterrupted() {...}
      public static boolean interrupted() {...}
5  }

```

- Pozor na metodu interrupted()
 - vrátí a *vymaže* stav příznaku
- Zpracování přerušení
 - vyhození výjimky `InterruptedException`
 - předání příznaku dále
 - polknutí příznaku 
- Typické metody na `InterruptedException`
 - `wait`, `sleep`, `join`
 - blokující operace na omezených frontách (`BlockingQueue x.put`)

Přerušení – interrupt

- Politiky přerušení

- specifikováno vývojářem pro každé vlákno
- standardní chování: uklid', dej vědět vlastníkovi (TPE) a zmiz
- nestandardní chování: není vhodné pro normální úlohy
- vlákno může potřebovat předat stav `interrupted` svému TPE
- úloha by neměla předpokládat nic o politice vlákna, v němž běží
 - ◆ předat stav dál
 - ◆ buď `throw new InterruptedException();`
 - ◆ nebo `Thread.currentThread().interrupt();`
např. pokud je úloha `Runnable`
- vlákno/TPE může následně `interrupted` příznak potřebovat
- specifikace: kdy?, jak?, další předání?

Přerušení – interrupt

- Kombinace blokujících operací s politikou přerušování a úlohy s ukončením až na konci

```
1 public Task getNextTask(BlockingQueue<Task> queue) {  
2     boolean interrupted = false;  
3     try {  
4         while (true) {  
5             try {  
6                 return queue.take();  
7             } catch (InterruptedException e) {  
8                 interrupted = true;  
9             }  
10        }  
11    } finally {  
12        if (interrupted) Thread.currentThread().interrupt();  
13    }  
14 }
```

- nesmíme příznak `interrupted` nastavit před voláním `take()`, protože by volání hned skončilo

Omezený běh – Futures

- **Future** má metodu `cancel` (`boolean mayInterruptIfRunnig`)
 - `mayInterruptIfRunnig = true` znamená, že se má běžící úloha přerušit
 - `mayInterruptIfRunnig = false` znamená, že se pouze nemá spustit, pokud ještě neběží
 - vrací, zda se ukončení povedlo
- Kdy můžeme použít `mayInterruptIfRunnig = true`?
 - pokud známe politiku přerušování vlákn
 - pro standardní implementace `Executor` to je známé a bezpečné

Omezený běh – Futures

```
public class FutureCancel {
2   ThreadPoolExecutor taskExec = new ThreadPoolExecutor(1,10,60,
      TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
4   public void timedRun (Runnable r, long timeout, TimeUnit unit)
      throws InterruptedException {
6       Future<?> task = taskExec.submit(r);
      try {
8           task.get(120, TimeUnit.SECONDS);
        } catch (ExecutionException e) {
10            throw new RuntimeException(e.getMessage());
        } catch (TimeoutException e) {
12            // uloha bude ukoncena nize
        }
14     finally {
16         // neskodne, pokud ukloha skončila,
            // jinak interrupt
            task.cancel(true);
18     }
}
```

Nepřerušitelná blokování

- Existují blokování, která nereagují na `interrupt`
- Příklady:
 - synchronní soketové I/O v `java.io`
 - ◆ *problém*: metody `read` a `write` na `InputStream` a `OutputStream` nereagují na `interrupt`
 - ◆ *řešení*: zavřít socket, visící čtení/zápis vyhodí `SocketException`
 - čekání na získání monitoru (intrinsic lock)
 - ◆ *problém*: vlákno čekající na monitor (`synchronized`) nereaguje na `interrupt`
 - ◆ *řešení*: neexistuje „násilné“ řešení pro monitory, musí se dočkat
 - ◆ *obejítí*: explicitní zámky `Lock` podporují metodu `lockInterruptibly`



Nepřerušitelná blokování

- Další vychytávky:
 - synchronní I/O v java.nio
 - ♦ přerušování vyhází u všech zablokovaných vláken `ClosedByInterruptException`, pokud je kanál typu `InterruptibleChannel`
 - ♦ zavření vyhází u všech zablokovaných vláken `AsynchronousCloseException`, pokud je kanál typu `InterruptibleChannel`
 - asynchronní I/O při použití Selector
 - ♦ `Selector.select` vyhodí výjimku `ClosedSelectorException`, pokud obdrží `interrupt`

Nepřerušitelná blokování

- Využití `ThreadPoolExecutor.newTaskFor(callable)`
 - dostupné od Java 6
 - vrací `RunnableFuture` pro danou úlohu
 - přepsání `newTaskFor` umožňuje vlastní tvorbu `RunnableFuture` a tudíž přepsat metodu `cancel()`
 - ◆ uzavření synchronních socketů pro java.io
 - ◆ statistiky, debugování, atd.
 - lze napsat tak, že si `Callable/Runnable` dodá vlastní implementaci `cancel()`
- <http://www.javaconcurrencyinpractice.com/listings/SocketUsingTask.java>



Zastavování vláknových služeb

- Problém dlouho běžících vláken
 - vlákna v exekutorech často běží déle, než tvůrce executorů
- Vlákno by měl zastavovat jeho „vlastník“
 - vlastník vláken není definován formálně
 - bere se ten, kdo ho vytvořil
 - vlastnictví není transitivní (jako u objektů – princip zapouzdření)
 - vlastník by měl poskytovat metody na řízení životního cyklu
 - požadavek na ukončení by měl být signalizován vlastníkovi

Zastavování vláknových služeb

```
public class LogWriter {
2   private final BlockingQueue<String> queue;
   private final LoggerThread logger;
4   private volatile boolean shutdownRequested = false;

6   public LogWriter() throws FileNotFoundException {
       this.queue = new LinkedBlockingQueue<String>();
       this.logger = new LoggerThread(new PrintWriter("mujSoubor"));
       logger.start();
10  }

12  private class LoggerThread extends Thread {
       private final PrintWriter writer;

14      private LoggerThread(PrintWriter writer) {
16          super("Logger Thread");
           this.writer = writer;
18      }

20      public void run() {
           try {
22          while (true)
               writer.println(queue.take());
           } catch (InterruptedException ignored) {
24          } finally {
26          writer.close();
           }
28      }
}
```

Zastavování vláknových služeb

```
1 public void stop() {  
2     shutdownRequested = true;  
3     logger.interrupt();  
4 }  
5  
6 public void log (String msg) throws InterruptedException {  
7     queue.put (msg);  
8 }
```



- Potřeba ukončovat konzumenty i producenty
 - konzument: `run()`
 - producent: `log(String msg)`


Zastavování vláknových služeb

```
2 public void logLepe (String msg) throws InterruptedException {  
3     if (!shutdownRequested)  
4         queue.put (msg);  
5     else  
6         throw new IllegalStateException("logger se ukoncuje");  
7 }
```

- Ukončení producenta
 - jakpak zjistíme jeho vlákno?
 - nijak ;-)
 - už je to správně?

Zastavování vláknových služeb

```
1 public void logLepe (String msg) throws InterruptedException {
2     if (!shutdownRequested)
3         queue.put (msg);
4     else
5         throw new IllegalStateException("logger se ukončuje");
6 }
```



- ... není!
- Race condition
 - složené testování podmínky a volání metody!
- Složené zamykání
 - testování a rezervace v jednom `synchronized` bloku
 - konzument testuje, že zpracoval všechny rezervace

Zastavování vláknových služeb

```
1 public class SafeLogWriter {  
2     private final BlockingQueue<String> queue;  
3     private final LoggerThread logger;  
4     @GuardedBy("this") private volatile boolean shutdownRequested  
5         = false;  
6     @GuardedBy("this") private int reservations;
```

...

```
1     public void run() {  
2         try {  
3             while (true) {  
4                 synchronized (this) {  
5                     if (shutdownRequested && reservations == 0)  
6                         break;  
7                 }  
8                 String msg = queue.take();  
9                 synchronized (this) {--reservations;};  
10                writer.println(msg);  
11            }  
12        } catch (InterruptedException ignored) {  
13        } finally {  
14            writer.close();  
15        }  
16    }
```



Zastavování vláknových služeb

```
2 public void log (String msg) throws InterruptedException {  
3     synchronized (this) {  
4         if (shutdownRequested)  
5             throw new IllegalStateException("logger se ukoncuje");  
6         ++reservations;  
7     }  
8     queue.put (msg);  
9 }
```

Zastavování vláknových služeb

- **ExecutorService**

- proč nepoužít, co je hotovo?
- **shutdown ()**
 - ◆ pohodové ukončení
 - ◆ dokončí se zařazené úlohy
- **shutdownNow ()**
 - ◆ vrací seznam úloh, které ještě nenastartovaly
 - ◆ problém, jak se dostat k seznamu úloh, které nastartovaly, ale byly ukončeny
- nemá metodu, která by umožnila dokončit běžící úlohy a nové už nestartovala
- zapouzdření do vlastního ukončování:
`exec.shutdown ();`
`exec.awaitTermination (timeout, unit);`
- využití i pro jednoduchá vlákna: `newSingleThreadExecutor ()`

Zastavování vláknových služeb

```

2 public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService exec;
4     private final Set<Runnable> tasksCancelledAtShutdown =
        Collections.synchronizedSet(new HashSet<Runnable>());

```

...

```

2     public List<Runnable> getCancelledTasks() {
        if (!exec.isTerminated())
            throw new IllegalStateException("/*...*/");
4         return new ArrayList<Runnable>(tasksCancelledAtShutdown);
    }
6
8     public void execute(final Runnable runnable) {
        exec.execute(new Runnable() {
            public void run() {
10                try {
                    runnable.run();
12                } finally {
                    if (isShutdown()
14                        && Thread.currentThread().isInterrupted())
                        tasksCancelledAtShutdown.add(runnable);
16                }
            }
18        });
    }

```


Zastavování vláknových služeb

- Vzor – jedovaté sousto
 - ukončování systému producent – konzument
 - jedovaté sousto – jeden konkrétní typ zprávy
 - funguje pro známý počet producentů
 - ◆ konzument umře po požití N_{prod} otrávených soust
 - lze rozšířit i na více konzumentů
 - ◆ každý producent musí do fronty zapsat N_{konz} otrávených soust
 - ◆ problém s počtem zpráv $N_{prod} \cdot N_{konz}$

Ošetření abnormálního ukončení vlákna

- Zachytávání `RuntimeException`

- normálně se nedělá, měla by vyústit v stacktrace
- potřeba zpracovat, pokud vlákno vykonává úplně cizí kód
- strategie:
 - ◆ zachytit, uložit, pokračovat
`try {...} catch (...) {...}`
v případě, že se vlákno o sebe musí postarat samo
 - ◆ ukončit a dát vědět vlastníkovi
`try {...} finally {...}`
možnost předat `Throwable`

```
Throwable thrown = null;  
2 try {runTask(getTaskFromQueue());}  
  catch (Throwable e) {thrown = e;}  
4 finally { threadExited (this, thrown);}
```

Ošetření abnormálního ukončení vlákna

- **UncaughtExceptionHandler**

- aplikace si může nastavit vlastní zpracování nezachycených výjimek
- pokud není nastaven, vypisuje se stacktrace na `System.err`

1. **Thread.setUncaughtExceptionHandler**

- ◆ Java \geq 5.0
- ◆ per vlákno

2. **ThreadGroup**

- ◆ Java < 5.0

- zavolá se pouze první
- pro TPE se nastavuje pomocí vlastní **ThreadFactory** přes konstruktor TPE
 - ◆ standardní TPE nechá po nezachycené výjimce ukončit dané vlákno
 - ◆ bez **UncaughtExceptionHandler** mohou vlákna tiše mizet
 - ◆ možnost task obalit do dalšího Runnable/Callable
 - ◆ vlastní TPE s alternativním **afterExecute**

- Propagace nezachycených výjimek

- do **UncaughtExceptionHandler** se dostanou pouze úlohy zaslané přes `execute()`
- `submit()` vrací výjimku jakou součást návratové hodnoty/stavu – `Future.get()`

Ukončování JVM

- Normální ukončení (orderly termination)
 - ukončení posledního nedémonického vlákna
 - volání `System.exit()` ;
 - platformově závislé ukončení (SIGINT, Ctrl-C)
- Abnormální ukončení (abrupt termination)
 - volání `Runtime.halt()` ;
 - platformově závislé ukončení (SIGKILL)
- Háčky při ukončení (shutdown hooks)
 - `Runtime.addShutdownHook`
 - předává se implementace vlákna
 - JVM negarantuje pořadí
 - pokud v době ukončování běží jiná vlákna, poběží paralelně s háčky
 - háčky musí být thread-safe: synchronizace
 - např. signalizace ukončení jiným vláknům, mazání dočasných souborů,
...
 - pokud nějaké vlákno počítá se signalizací ukončení při ukončování JVM, může si samo zaregistrovat háček (ale ne z konstruktoru!)
 - použití jednoho velkého háčku: odpadá problém se synchronizací, možnost zajištění definovaného pořadí ukončování komponent

Ukončování JVM

- Démonická vlákna

- metoda `setDaemon()`
- démonický stav se dědí
- ukončování JVM: pokud běží jen démonická vlákna, JVM se normálně ukončí
 - ◆ neprovedou se bloky `finally`
 - ◆ neprovede se vyčištění zásobníku
- příklad: garbage collection, čištění dočasné paměťové cache
- **nepoužívat z lenosti!**

- Finalizers

- týká se objektů s netriviální metodou `finalize()`
 - ◆ obtížné napsat správně
 - ◆ musí být synchronizovány
 - ◆ není garantováno pořadí
 - ◆ výkonnostní penalta
 - ◆ obvykle jde nahradit pomocí bloku `finally` a explicitního uvolnění zdrojů
- po doběhnutí háčku se spustí finalizers pokud `runFinalizersOnExit == true`
- **vyhýbat se jim!**

Typy úloh pro TPE

- Nezávislé úlohy – ideální
- Problémy
 - závislost/komunikace úloh zaslanych do jednoho TPE
 - ◆ ohraničená velikost TPE
 - jednovláknový executor → TPE
 - úlohy citlivé na latenci odpovědi
 - ◆ ohraničená velikost TPE
 - ◆ dlouho běžící úlohy
 - problém s úlohami využívajícími `ThreadLocal`
 - ◆ recyklace vláken
 - nestejně velké úlohy v jednom TPE

Typy úloh pro TPE

Je tohle správně?

```
1  static ExecutorService exec = Executors.newSingleThreadExecutor();
2
3  public static class RenderPageTask implements Callable<String> {
4      public String call() throws Exception {
5          Future<String> header, footer;
6          header = exec.submit(new LoadFileTask("header.html"));
7          footer = exec.submit(new LoadFileTask("footer.html"));
8          String page = renderBody();
9          return header.get() + page + footer.get();
10     }
11
12     private String renderBody() {
13         return " body ";
14     }
15 }
```

Typy úloh pro TPE

ANO

```
1      ExecutorService mainExec = Executors.newSingleThreadExecutor();
2      Future<String> task = mainExec.submit(new RenderPageTask());
3      try {
4          System.out.println("Vysledek: " + task.get());
5      } catch (InterruptedException e) {
6          e.printStackTrace();
7      } catch (ExecutionException e) {
8          e.printStackTrace();
9      }
10     exec.shutdown();
11     mainExec.shutdown();
```


Typy úloh pro TPE

NE

```
1      Future<String> task = exec.submit(new RenderPageTask());
2      try {
3          System.out.println("Vysledek: " + task.get());
4      } catch (InterruptedException e) {
5          e.printStackTrace();
6      } catch (ExecutionException e) {
7          e.printStackTrace();
8      }
9      exec.shutdown();
```

Typy úloh pro TPE

- Nezávislé úlohy – ideální
- Problémy
 - závislost/komunikace úloh zaslanych do jednoho TPE
 - ◆ ohraničená velikost TPE
 - jednovláknový executor → TPE
 - úlohy citlivé na latenci odpovědi
 - ◆ ohraničená velikost TPE
 - ◆ dlouho běžící úlohy
 - problém s úlohami využívajícími `ThreadLocal`
 - ◆ recyklace vláken
 - nestejně velké úlohy v jednom TPE

Velikost TPE

- Doporučení Javy: $N_{CPU} + 1$ pro výpočetní úlohy
- Obecněji

$$N_{vlaken} = N_{CPU} \cdot U_{CPU} \cdot \left(1 + \frac{W}{C}\right)$$

kde U_{CPU} je cílové využití CPU, W je čas čekání, C je výpočetní čas

- `Runtime.getRuntime().availableProcessors();`

Vytváření a ukončování vláken v TPE

- **corePoolSize** – cílová velikost zásobárny vláken
 - startují se, až jsou potřeba (default policy)
 - **prestartCoreThread()** – nastartuje jedno core vlákno a vrátí **boolean**, zda se povedlo
 - **prestartAllCoreThreads()** – nastartuje všechna core vlákna a vrátí jejich počet
- **maximumPoolSize** – maximální velikost zásobárny vláken
- **keepAliveTime** – doba lelkujícího života
 - od Javy 6: **allowCoreThreadTimeOut** – dovoluje timeout i core vláknům

Správa front v TPE

- Kdy se množí vlákna v TPE?
 - pokud je fronta **plná**
 - co se stane, pokud `corePoolSize = 0` a používáme neomezenou frontu?
- Použití synchronní fronty
 - `SynchronousQueue` není fronta v pravém slova smyslu!
 - synchronní předávání dat mezi úlohami
 - pokud žádné vlákno na předání úlohy nečeká, TPE natvoří nové
 - při dosažení limitu se postupuje podle saturační politiky
 - lze použít při neomezeném počtu vláken (`Executors.newCachedThreadPool`) nebo pokud je akceptovatelné použití saturační politiky
 - efektivní (čas i zdroje) – `Executors.newCachedThreadPool` je efektivnější než `Executors.newCachedThreadPool`, který využívá `LinkedBlockingQueue`
 - implementováno pomocí neblokujícího algoritmu v Java 6, 3× větší výkon než Java 5

Správa front v TPE

- Použití prioritní fronty
 - task musí implementovat `Comparable` (přirozené pořadí) nebo `Comparator`
- Saturační politiky
 - nastupuje v okamžiku zaplnění fronty
 - nastavuje se pomocí `setRejectedExecutionHandler` nebo konstruktoru TPE
 - `AbortPolicy` – default, úloha dostane `RejectedExecutionException`
 - `CallerRunsPolicy` – využití volajícího vlákna
 - ◆ řízení formou zpětné vazby
 - `DiscardPolicy` – vyhodí nově zaslanou úlohu
 - `DiscardOldestPolicy` – vyhodí „nejstarší“ úlohu
 - ◆ vyhazuje z hlavy front \implies nevhodné pro použití s prioritními frontami
 - ◆ pomáhá vytlačit problém do vnějších vrstev: např. pro web server – nemůže zavolat další `accept` – spojení čekají v TCP stacku

Správa front v TPE

```

2      ThreadPoolExecutor tpe =
          new ThreadPoolExecutor(1, 10, 60, TimeUnit.SECONDS,
          new LinkedBlockingQueue<Runnable>(100));
4      tpe.setRejectedExecutionHandler
          (new ThreadPoolExecutor.CallerRunsPolicy());
  
```

- Implementace omezení plnění fronty pomocí semaforu
 - semafor se nastaví na požadovanou velikost fronty + počet běžících úloh

```

1  @ThreadSafe
   public class BoundedExecutor {
3      private final Executor exec;
       private final Semaphore semaphore;

5

       public BoundedExecutor(Executor exec, int bound) {
7          this.exec = exec;
           this.semaphore = new Semaphore(bound);
9      }
  
```

Správa front v TPE

```
public void submitTask(final Runnable command)
2     throws InterruptedException {
3     semaphore.acquire();
4     try {
5         exec.execute(new Runnable() {
6             public void run() {
7                 try {
8                     command.run();
9                 } finally {
10                    semaphore.release();
11                }
12            }
13        });
14    } catch (RejectedExecutionException e) {
15        semaphore.release();
16    }
}
```


Kvízy

1. Zkuste navrhnout a implementovat thread pool, který se bude dynamicky zvětšovat/zmenšovat podle počtu čekajících požadavků ve frontě.
2. Zkuste rozmyslet a navrhnout, jak by bylo možno implementovat afinitu k procesoru u Javovských vláken a za jakých okolností by tato konstrukce fungovala.

Java NIO

- Zavedeno v Javě 1.4 (JSR 51)
- Abstraktní třída **Buffer**
 - umožňuje držet pouze primitivní typy

```
ByteBuffer  
CharBuffer  
DoubleBuffer  
FloatBuffer  
IntBuffer  
LongBuffer  
ShortBuffer
```

- direct vs. non-direct buffery
 - přímé buffery se snaží vyhnout zbytečným kopiím mezi JVM a systémem
- vytváření pomocí metod
 - `allocate` – alokace požadované velikosti
 - `allocateDirect` – alokace požadované velikosti typu `direct`
 - `wrap` – zabalí existující pole bytů (`bytearray`)



Java NIO

- **ByteBuffer**

- <http://download.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html>
- přístup k binárním datům, např.

```
float getFloat()  
float getFloat(int index)  
void putFloat(float f)  
void putFloat(int index, float f)
```

- mapování souborů do paměti (`FileChannel`, metoda `map`)
- čtení/vložení z/do bufferu bez parametru `index` (`get/put`) inkrementuje pozici
- pokud není řečeno jinak, metody vrací odkaz na buffer – řetězení volání

```
buffer.putShort(10).putInt(0x00ABBCCD).putShort(11);
```

Java NIO

- Vlastnosti bufferů

capacity celková kapacita bufferu

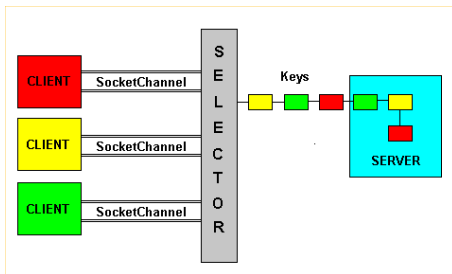
limit umělý limit uvnitř bufferu, využití s metodami `flip`
(nastaví limit na současnou pozici a skočí na pozici 0)
Či `remaining`

mark pomocná značka, využití např. s metodou `reset` (skočí na označovanou pozici)

```
buffer.position(10);  
2 buffer.flip();  
while (buffer.hasRemaining()) {  
4 byte b = buffer.get();  
  // neco  
6 }
```

Java NIO

- Selektor
 - serializace požadavků
 - výběr požadavků
- Klíč
 - identifikace konkrétního spojení



Zdroj: <http://onjava.com/lpt/a/2672>

Java NIO – Server

- Generický postup

```
1 create SocketChannel;
2 create Selector
3 associate the SocketChannel to the Selector
4 for(;;) {
5     waiting events from the Selector;
6     event arrived; create keys;
7     for each key created by Selector {
8         check the type of request;
9         isAcceptable:
10            get the client SocketChannel;
11            associate that SocketChannel to the Selector;
12            record it for read/write operations
13            continue;
14        isReadable:
15            get the client SocketChannel;
16            read from the socket;
17            continue;
18        isWritable:
19            get the client SocketChannel;
20            write on the socket;
21            continue;
22    }
23 }
```



Java NIO – Server

```
1 // Create the server socket channel
  ServerSocketChannel server = ServerSocketChannel.open();
3 // nonblocking I/O
  server.configureBlocking(false);
5 // host-port 8000
  server.socket().bind(new java.net.InetSocketAddress(host, 8000));
7 // Create the selector
  Selector selector = Selector.open();
9 // Recording server to selector (type OP_ACCEPT)
  server.register(selector, SelectionKey.OP_ACCEPT);
```

Zdroj: <http://onjava.com/lpt/a/2672>

Java NIO – Server

```
2 // Infinite server loop
3 for(;;) {
4     // Waiting for events
5     selector.select();
6     // Get keys
7     Set keys = selector.selectedKeys();
8     Iterator i = keys.iterator();
9
10    // For each keys...
11    while(i.hasNext()) {
12        SelectionKey key = (SelectionKey) i.next();
13
14        // Remove the current key
15        i.remove();
16
17        // if isAcettable = true
18        // then a client required a connection
19        if (key.isAcceptable()) {
20            // get client socket channel
21            SocketChannel client = server.accept();
22            // Non Blocking I/O
23            client.configureBlocking(false);
24            // recording to the selector (reading)
25            client.register(selector, SelectionKey.OP_READ);
26            continue;
27        }
28    }
29 }
```


Java NIO – Server

```
2 // if isReadable = true
3 // then the server is ready to read
4 if (key.isReadable()) {
5
6     SocketChannel client = (SocketChannel) key.channel();
7
8     // Read byte coming from the client
9     int BUFFER_SIZE = 32;
10    ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
11    try {
12        client.read(buffer);
13    }
14    catch (Exception e) {
15        // client is no longer active
16        e.printStackTrace();
17        continue;
18    }
19
20    // Show bytes on the console
21    buffer.flip();
22    Charset charset=Charset.forName('' ISO-8859-1'' );
23    CharsetDecoder decoder = charset.newDecoder();
24    CharBuffer charBuffer = decoder.decode(buffer);
25    System.out.print(charBuffer.toString());
26    continue;
27 }
28 }
```

Java NIO

- Další čtení:

- <http://onjava.com/lpt/a/2672>
- <http://onjava.com/lpt/a/5127>
- <http://download.oracle.com/javase/6/docs/api/java/nio/channels/Selector.html>
- <http://download.oracle.com/javase/6/docs/api/java/nio/channels/SelectionKey.html>

Asynchronní programování versus vlákna

- Asynchronní programování
 - + umožňuje obsluhovat řádově větší množství klientů
 - za cenu zvýšení latence
 - složitější, náchylnější na chyby
- Vlákňové programování
 - + jednodušší
 - + poměrně efektivní do „rozumného“ počtu vláken
 - nativní vlákna nejsou stavěna na (deseti)tisíce vláken a více
- Potenciálně lze kombinovat