

KONKURENTNÉ PROGRAMOVANIE

7. cvičenie: ForkJoinPool, Ukončovanie úloh, vlákien a exekútorov

Exekútor ForkJoinPool (od Javy 7)

- Špeciálny exekútor pre rekurzívne úlohy
- Využíva návrhový vzor work stealing pre daný počet vlákien
- Akceptuje špeciálne typy úloh (potomkovia ForkJoinTask)
 - ▣ **RecursiveTask<T>** - úloha s návratovou hodnotou
 - ▣ **RecursiveAction** – úloha bez návratovej hodnoty
- Rekurzívna úloha vyrába nové rekurzívne úlohy toho istého typu a posiela ich exekútoru
- Úloha typicky čaká na dobehnutie úloh, čo zavolala, aby zosumarizovala výsledky a mohla tiež skončiť
 - ▣ Počas čakania je jej odňaté vlákno pre iné úlohy, ktoré majú čo robiť

Exekútor ForkJoinPool

- Vytváranie cez konštruktor (defaultne toľko vlákien ako jadier)

```
ForkJoinPool forkJoinPool = new ForkJoinPool();
```

- Alternatívne, cez Executors:

```
ExecutorService forkJoinPool =  
    Executors.newWorkStealingPool();
```

- Zaslanie úlohy do exekútora
 - ▣ Bez čakania na výsledok – `execute(úloha)`
 - ▣ S čakaním na ukončenie úlohy – `invoke(úloha)`
 - ▣ Zaslanie úlohy s neskorším počkaním na ukončenie úlohy – `submit(úloha)`
- Neskoršie počkanie na dokončenie úlohy po volaní `submit(úloha)`
 - `T` výsledok = `úloha.join()`, ak `úloha` je `RecursiveTask<T>`
 - `úloha.join()`, ak `úloha` je `RecursiveAction`

RecursiveTask pre ForkJoinPool

Prekrývame metódu
compute()

```
public class MyTask extends RecursiveTask<MyResult> {  
    public MyResult compute() { //výpočet úlohy  
        MyTask podúloha1 = new MyTask(...);  
        podúloha1.fork(); // odošlem podúlohu exekútoru  
        MyTask podúloha2 = new MyTask(...);  
        podúloha2.fork(); // odošlem podúlohu exekútoru  
        MyResult result1 = podúloha1.join();  
        // čakám na výsledok podúlohy1  
        // vlákno mi je odňaté, kým nepríde výsledok  
        MyResult result2 = podúloha2.join();  
        // čakám na výsledok podúlohy2  
        // vlákno mi je odňaté, kým nepríde výsledok  
        return createResult(result1,result2);  
    }  
}
```

RecursiveTask pre ForkJoinPool

- Dedí od abstraktnej ForkJoinTask
- Implementuje Future, takže s ním vieme administrovať aj monitorovať úlohu
- Vieme ho vyrobiť aj z Callable a Runnable úloh
 - ▣ ForkJoinTask fjtÚloha = ForkJoinTask.adapt(úloha)
- Vieme spustiť veľa úloh a čakať na nich cez invokeAll()
- ... kopa ďalších vychytávok

Zasielanie úloh do ForkJoinPool-u

	Z externého kódu	Z úlohy vo vnútri
Bez čakania na výsledok	<code>execute(forkJoinTask)</code>	<code>forkJoinTask.fork()</code>
S čakaním na výsledok	<code>invoke(forkJoinTask)</code>	<code>forkJoinTask.invoke()</code>
S neskorším čakaním na výsledok pomocou <code>forkJoinTask.join()</code>	<code>submit(forkJoinTask)</code>	<code>forkJoinTask.fork()</code>

Zadanie 1

- Stiahnite si z GitHubu poslednú verziu:
 - ▣ <https://github.com/PeterGursky/kopr2017>
- projekt z minula:
 - ▣ `sk.gursky.kopr.cviko6.priklad4`
- Modifikujte riešenie posledného zadania 4 z minulej hodiny tak, že využijete exekútor ForkJoinPool a úlohu typu RecursiveTask

Predčasné ukončenie úloh

- Používateľská požiadavka
- Časovo obmedzené aktivity
 - ▣ Čakaj na zdroj určitý čas a ak neodpovie, vypíš default
- Udalosti aplikácie
 - ▣ Nejaká časť programu zistila, že výsledok inej úlohy už nebude potrebovať
- Chyby a výnimky
- Vypnutie programu

Chyby a výnimky v úlohe

- Najčastejšie nekontrolované výnimky
- `thread.start(Runnable úloha)`
 - ▣ Výnimka vyletí do `System.err`, ale beh hlavného vlákna neovplyvňuje
- `Future future = exekútor.submit(úloha)`
 - ▣ `future.get()` buď vráti výsledok, alebo
 - ▣ vyhodí výnimku `ExecutionException`, v ktorej je zabalená výnimka vyhodená z úlohy

Chyby a výnimky v úlohe

□ ForkJoinTask

- `úloha.join()` vyhazuje priamo výnimku, ktorá bola vyhodená v kóde
- Musí ísť o `RuntimeException` alebo `Error`
 - metóda `compute()` nemôže vyhadzovať kontrolované výnimky

Zadanie 2

- Stiahnite si z GitHubu poslednú verziu:
 - ▣ <https://github.com/PeterGursky/kopr2017>
- projekt z minula:
 - ▣ `sk.gursky.kopr.cviko6.priklad1`
- Zabezpečte odchyťávanie výnimiek pre podadresáre, na ktoré nemáte dostatočné práva a vykonajte formátovaný výpis o tom do konzoly

Predčasné ukončenie úloh z vonku

- Úloha si beží – vykonáva svoje príkazy
- V Jave je nemožné vlákno násilne ukončiť a zabrániť mu vykonávať ďalšie príkazy
 - ▣ To by súdny človek málokedy chcel
- Má zmysel, aby vlákno pouzatváralo svoju prácu a až potom skončilo
- Nechceme nechať zdieľané štruktúry v nekonzistentnom stave
- Úlohu/vlákno môžeme o ukončenie požiadať

Vlastné riešenie požiadavky ukončenia

```
public class GeneratorPrvocisiel implements Runnable {
    private final List<BigInteger> prvocisla = new ArrayList<>();
    private volatile boolean zrušené;
    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!zrušené) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }
    public void zruš() { zrušené = true; }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

Nedostatky vlastného riešenia

- Vlastné riešenie môže fungovať pokiaľ nemáme blokované operácie
 - ▣ donekonečna čakám na prvok z blokovaneho radu, do ktorého mi nič nepríde
 - ▣ už sa nedostanem k overeniu, či mi niekto nastavil zrušené na true
- Takmer všetky blokované metódy sú citlivé na prerušenie vlákna cez metódu interrupt()
 - ▣ Vyhodia výnimku InterruptedException
 - ▣ A teda: **odblokujú sa!!!**

Thread.interrupted

- Každé vlákno má privátnu inštančnú premennú **interrupted**
 - **void interrupt()**
 - nastaví interrupted na true
 - **boolean isInterrupted()**
 - vráti hodnotu interrupted
 - **static boolean interrupted()**
 - vráti hodnotu interrupted a nastaví interrupted na false
 - jediný spôsob ako zrušiť interrupted stav

Blokované operácie necitlivé na interrupted 1

□ InputStream a OutputStream

- ▣ Metódy read() a write() blokovane čakajú na umožnenie čítania/zapisovania
- ▣ Riešenie – stačí zatvoriť zdroj, s ktorým pracujem - read() a write() vyhodia IOException namiesto InterruptedException

```
public class SocketThread extends Thread {
    private final Socket socket;

    ...

    public void interrupt() {
        try { socket.close(); }
        catch (IOException ignored) { }
        finally { super.interrupt(); }
    }
}
```


Blokované operácie necitlivé na interrupted 2

- Kritická sekcia cez synchronized
 - ▣ Vlákno čakajúce na vstup do kritickej sekcie sa nezobudí na interrupted
 - ▣ Dôsledok: deadlock - mrznutie aplikácie, ktorá sa následne nedá vypnúť (len cez odstrelenie procesu)
- Riešenie: vieme použiť Semaphore(1) alebo Lock
 - ▣ `void lockInterruptibly() throws InterruptedException`
 - ▣ `void unlock()`
 - ▣ Aktuálnosť zdieľaných premenných už musíme riešiť samostatne!

Prerušenie úlohy

- Ak chcem prerušiť úlohu, najspol'ahlivejšie je to cez prerušenie (interruption) vlákna, v ktorom beží
- V úlohe by sme teda mali často
 - ▣ overovať či náhodou `isInterrupted()` nevráti `true` a/alebo
 - ▣ volať blokovanú operáciu a odchytať výnimku `InterruptedException`
- Je na mieste otázka
 - ▣ Chcel som ukončiť úlohu alebo aj vlákno?
 - ▣ Čo keď je to vlákno v exekútore?

Zásada vlastníka

- Beh metódy nejakého objektu by mal rušiť iba jeho vlastník = tvorca objektu
- main → thread pool → vlákno → úloha → podúloha
- main → úloha → podúloha
 - ▣ Reťaze môžu byť aj dlhšie, aj kratšie
- Nikdy nepreskakujeme medzivlastníka pri rušení
 - ▣ Napr. main neruší vlákno priamo, ak je v thread pool-e
- Vlákno nevlastní úlohu, iba ju spúšťa
 - ▣ malo by však vedieť o tom, že úloha je zrušená napr. z mainu
- Ak kód niekto zrušil, o zrušení informujeme toho, kto nás spustil

Negatívny príklad rušenia vlákna, ktoré nevlastníme

```
private ScheduledExecutorService cancelExec = ...;
...
public void timedRun(Runnable r, long timeout, TimeUnit unit) {
    final Thread taskThread = Thread.currentThread();
    cancelExec.schedule(new Zrušiteľ(taskThread), timeout, unit);
    r.run();
}
```

```
public class Zrušiteľ implements Runnable {
    private Thread cudzieVlákno;
    public Zrušiteľ(Thread cudzieVlákno) {
        this.cudzieVlákno = cudzieVlákno;
    }
    public void run() {
        cudzieVlákno.interrupt();
    }
}
```

```
spustacUloh.timedRun(úloha, 1, TimeUnit.SECONDS) }
```

Negatívny príklad rušenia vlákna, ktoré nevlastníme

- Čo sa môže v tomto príklade stať?
 - ▣ Ak úloha, ktorú sme spustili, nezisťuje interrupted stav svojho vlákna,
 - vôbec to nemusí bežať iba 1 sekundu, ale pokojne do nekonečna
 - ▣ Ak úloha skončí skôr, ako nastane interrupt
 - Vlákno už môže robiť úplne inú úlohu, ktorú takto zrušíme
- **Nikdy neprerušujeme cudzie vlákna, ktoré sme nevytvárali!!!**

Informovanie vlákna o zrušení

- Stav interrupted nastavuje
 - ▣ Thread pool (exekútor) alebo iný tvorca vlákna
 - ▣ Tvorca úlohy
- Úloha nevie, kto z nich to spravil !
 - ▣ Svoje vlákno, alebo nadúlohu, by preto mala o zrušení informovať vždy
- S interrupted stavom sa v kóde môžeme korektne vysporiadať nasledovne:
 - ▣ Vyhodíme výnimku InterruptedException
 - ▣ Zachováme interrupted stav na true
- Blokované operácie typicky vyhadzujú InterruptedException a zároveň rušia interrupted (nastavujú mu false)

Informovanie vlastníka o zrušení

- Ak máme kód úlohy, v ktorom môže nastat' `InterruptedException`
 - ▣ Môžeme túto výnimku iba vyhodit' cez `throws`
 - Konečne je pohodlné riešenie to správne 😊
 - V `Runnable` implementáciách sa to nedá – kontrolovaná výnimka (v `Callable` to ide)
 - ▣ Alebo ju odchytíme a znova vyhodíme
 - ▣ Alebo ju odchytíme a nastavíme `interrupted` na `true`
 - `Thread.currentThread().interrupt();`
- `InterruptedException` nikdy nezhl'távame!
 - ▣ Kód, ktorý nás spustil, by nemusel zistiť, že prebieha rušenie

Príklad konzumera

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean zrušené= false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                zrušené = true;
                // vlastník nás prerušuje - začneme nejaké upratovanie
            }
        }
    } finally {
        if (zrušené)
            Thread.currentThread().interrupt();
    }
}
```


Rušenie úlohy cez Future

- návratová hodnota metódy `submit()` exekútorov
 - ▣ `Future future = exekútor.submit(úloha)`
- najpohodlnejší spôsob rušenia úlohy
 - ▣ `future.cancel(boolean prerušiťVlákno)`
 - Ak sa úloha ešte nezačala vykonávať, tak sa ani nevykoná
 - Ak sa začala vykonávať a `prerušiťVlákno` je `false`, úloha sa nechá dobehnúť
 - Ak sa začala vykonávať a `prerušiťVlákno` je `true`, **nastaví vláknu `interrupted` na `true`**

Riešenie časovanej úlohy cez Future

```
void timedRun(Runnable r, long timeout, TimeUnit unit) throws  
    InterruptedException {  
    Future<?> future = exekútor.submit(r);  
    try {  
        future.get(timeout, unit);  
    } catch (TimeoutException e) {  
        // úloha ukončená po timeoute  
    } catch (ExecutionException e) {  
        // úloha vyhodila výnimku, vyhodíme ju tiež  
        throw e.getCause();  
    } finally {  
        // nastaví interrupt, ak úloha ešte beží  
        future.cancel(true);  
    }  
}
```

**Ak vlákno
niekto zrušil
pred
timeoutom**

Ukončovanie exekútorov

- Ukončujeme všetky úlohy poslané do exekútora
- ExecutorService má metódy
 - ▣ void **shutdown()**
 - Exekútor neprijíma ďalšie úlohy
 - submit(úloha) vyhodí RejectedExecutionException
 - Všetky prijaté úlohy dovykonáva a potom skončí
 - ▣ List<Runnable> **shutdownNow()**
 - Exekútor neprijíma ďalšie úlohy
 - Úlohy, ktoré nezačali sú vrátené ako list Runnable úloh
 - Úlohám, ktoré bežia, sú prerušené ich vlákna cez interrupt()

Príklad - Logovacie vlákno

```
public class LogService {
    private final ExecutorService exec = newSingleThreadExecutor();
    ...
    public void stop() throws InterruptedException {
        try {
            exec.shutdown();
            exec.awaitTermination(TIMEOUT, UNIT);
        } finally {
            printWriter.close();
        }
    }

    public void log(String msg) {
        try {
            exec.execute(new WriteTask(msg));
        } catch (RejectedExecutionException ignored) { }
    }
}
```

Čo bolo prijaté
bude
zalogované, ak
nevyprší timeout

Získanie úloh, ktoré boli prerušené

- Na to metódu nemáme, vieme to však nakódiť
 - ▣ Prerušené úlohy si zapamätáme v zdieľanej kolekcii

Zadanie 3

- Modifikujte riešenie zadania 2 tak, aby:
 - sa po danom časovom intervale prestalo prehľadávať
 - okrem ukončených prehľadávaní, nech sa vypíšu aj čiastočne prehľadané podstromy aj s ich s doteraz zistenou veľkosťou

Riešenie zadania 3

- DirAnalyzer testuje v každom adresári `Thread.currentThread().isInterrupted()`
 - ▣ Vyhodí výnimku `DirAnalyzerException`, v ktorej zahrnie adresár, jeho doteraz zistenú veľkosť a dôvod tejto výnimky
- Spustíme doterajší main ako úlohu v novom exekútore (vid' slajd o časovanej úlohe cez Future)
 - ▣ Prečíta si dobehnuté úlohy a list prerušených úloh
 - ▣ Keď odchyťí `InterruptedException` pošle svojmu exekútoru `shutdownNow()`
 - ▣ Volajúci kód rozlíši
 - `future.isDone()` – úloha dobehla ok
 - inak nezačala
 - Výnimka: úloha bola prerušená (bud' nemáme oprávnenia, alebo sme boli prerušení)
 - ▣ Nakoniec nastaví `Interrupted` svojho vlákna a zatvorí svoj exekútor

Sumarizácia – rušenie úloh

- Rušenie úlohy v našich vláknach
 - ▣ `vlákno.interrupt();`
 - ▣ odchytenie výnimiek úlohy v hlavnom vlákne nemožné
- Rušenie jednej úlohy v thread-pool/Exekútore
 - ▣ `future.cancel(true);`
 - ▣ odchytenie výnimiek úlohy cez `future.get()` zabalenej v `ExecutionException`
- Rušenie všetkých úloh v exekútore
 - ▣ `exekútor.shutdownNow();`
 - ▣ odchytenie výnimiek úlohy cez `future.get()` zabalenej v `ExecutionException`

Sumarizácia – odhaľovanie zrušenia

- Dostatočne často overujeme `Thread.currentThread().isInterrupted()`
- Väčšina blokovaných operácií vyhadzuje `InterruptedException`
 - ▣ Rušia tým stav `interrupted`
 - ▣ Potom buď opätovne vyhodíme `InterruptedException` alebo nastavíme `Thread.currentThread().interrupt()`
- Na `interrupted` nie sú citlivé streamy a kritické sekcie spravené cez `synchronized` – dá sa to však obísť