Otázka 31 - Y36PJV

Zadání

Vlákna, jejich atributy, metody, organizace a stavy. Možnosti synchronizace. (Y36PJV)

Procesy a vlákna

Proces

Každá aplikace je vlastně běžící proces. Pokud je aplikace spuštěna vícekrát, vytvoří se více procesů (instancí programu). Jeho vytvoření je vcelku náročné a má nezanedbatelnou režii.

Stav procesu zahrnuje přidělené místo v paměťi, zásobník, proces ID (PID), proces group ID, user ID, group ID, environment, pracovní adresář, heap, descriptory souborů, sdílené knihovny, atd.

Procesy nemohou přistupovat do paměti jiných procesů, ale mohou si navzájem data posílat.

Vlákno

Každý proces má alespoň jedno vlákno. Oproti procesu má omnoho nižší režii při vytváření (až o několik řádů). Vlákna sdílejí zdroje procesu (především paměťový prostor), který je vytvořil. Mohou tedy přistupovat ke stejným datům současně. Každé vlákno má vlastní zásobník, plánovací vlastnosti (prioritu), identifikační ID. Každé vlákno je zodpovědné za správný přístup ke sdíleným zdrojům v rámci procesu (např. paměti).

Vlákna

Vlákna slouží pro souběžné provádění více částí jediného programu. Jejich použití je někdy nezbytné, např. u serverového programu, který musí komunikovat s víc klienty. Na jednoprocesorovém počítači v jednu chvíli samozřejmě nemůže běžet více než jeden výpočet, můžou se ale časově překrývat I/O operace se zpracováním (např. práce se soubory, databází apod.). Dále lze např. vlákna použít v animacích, kdy je kód provádějící vykreslování řízen časovačem, aby se prováděl např. 20krát za sekundu. Pokud má běžet v jednu chvíli více animací, potom každá běží ve svém vlastním vláknu.

Příklad:

Program, který dělá tři aktivity:

1. Čte nějaké bloky dat ze souboru.
2. Provádí nějaké výpočty s každým blokem z těchto dat.
3. Zapisuje výsledek výpočtu do dalšího souboru.

Pokud by provádění probíhalo postupně 1-2-3-1-2-3 atd., tak by celkový čas provádění byl jednoduše součet časů všech jednotlivých kroků. Rozdělením programu na tři samostatná vlákna jsme schopni dosáhnout rychlejšího dokončení úlohy. Čtení ze souboru a zápis do souboru jsou časově náročné operace a velmi málo zatěžují CPU. Tento volný čas CPU je vhodné využít, např. pro provádění výpočtu ve chvíli, kdy dochází k I/O operaci.

Vytvoření vláken

Každý program v Javě má alespoň jedno vlákno. To je vytvořené při spuštění programu a začíná od metody **main()**. U apletu je hlavním vláknem prohlížeč. Když náš program vytvoří vlákno, je toto vlákno vytvořené navíc k vláknu, které jej vytvořilo.

Nové vlákno se spustí metodou **start()** objektu typu **Thread**. Kód, který se v novém vláknu provádí, je vždy public metoda zvaná **run()**. Aby vlákno něco dělalo je tedy třeba metodu **run()** implementovat. V těle této metody lze volat jakékoli jiné metody. Metoda **run()** se nevolá přímo, ale používá se již zmíněná metoda **start()**. Tím se vlákno dostane do stavu **runnable**, tzn. že je připraveno k běhu. Běžet začne až ve chvíli, kdy operační systém vybere nějaké vlákno ve stavu **runnable** a začne ho provádět. Pokud je nějaké vlákno prováděno má stav **running**. K zastavení vlákna stačí ukončit metodu **run()**.

V praxi se pak použije např. kód:

```
private boolean stop = false;

public void stop() {
  stop = true;
}

public void run() {
  while (!stop) {
    // kod provadejici se za behu vlakna
  }
}
```


PDF created with pdfFactory trial version www.pdffactory.com

While cyklus běží ve smyčce, dokud není zavolaná metoda **stop()**, která nastaví proměnou *stop* na *true* a tak se *while* cyklus ukončí a metoda **run()** skončí. Vlákno je doběhnutím metody **run()** ukončeno a dostane se do stavu **dead**.

Spouštění vláken je řízeno OS. Pokud by se metoda **run()** zavolala přímo, tak by se chovala jako jakákoli jiná metoda a běžela by ve stejném vláknu jako program, který ji zavolal.

Třídu, která bude představovat vlákno lze definovat dvěma způsoby. Buď nová třída bude podtřídou třídy **Thread** (*Vlakno extends Thread*) nebo bude implementovat rozhraní **Runnable** (*Vlakno implements Runnable*).

V prvním případě se pak nové vlákno spustí např. takto:

```
Thread vlakno1 = new Vlakno();
vlakno1.start();
```

Druhá možnost je většinou lepší, než odvozování od **Thread**. Třídu v Javě lze odvodit (**extends**) pouze od jediné třídy. Pokud se implementuje rozhraní **Runnable**, může třída vlákna stále odvozovat od třídy, kterou potřebujeme.

Spuštění vlákna *Vlakno* implementujici **Runnable**

```
Thread vlakno1 = new Thread(new Vlakno());
vlakno1.start();
```

Atributy vláken

Jméno vlákna

Každé vlákno má implicitní jméno sestávající z řetězce „**Thread***“, kde na konci je pořadové číslo. Konstruktor typu **Thread** umožňuje vlákno pojmenovat svým názvem. Tento název se použije jen při zobrazování informací o vláknu nebo při zavolání metody **getName()** a nemá žádný vztah k identifikátoru typu **Thread**.

Priority vláken

Všechna vlákna mají nějakou prioritu, která určuje, které z vláken bude prováděno, pokud jich několik čeká, až na ně dojde řada. To umožňuje dát jednomu vláknu více přístupu ke zdroji procesoru než druhému. Například, když existuje více vláken a jedno z nich provádí náročný výpočet, tak je možné mu dát menší prioritu, aby méně zatěžoval procesor a ostatní vlákna měla větší prostor k provádění.

Možné hodnoty priority vláken jsou definovány ve statických datových členech třídy **Thread**. Jsou typu **int** a jsou deklarovány jako **final**. Maxim. hodnota je definována členem **MAX_PRIORITY**, který má hodnotu *10*. Minimální priorita je **MIN_PRIORITY**, která je *0*. Když se vytvoří nějaké vlákno, jeho priorita bude stejná jako priorita vlákna, které ho vytvořilo. Hlavní vlákno **main()** má prioritu *5*.

Prioritu lze měnit metodou **setPriority()** a zjistit se dá metodou **getPriority()**.

Vlákno démona a vlákno uživatele

Metoda **setDaemon(boolean b)** třídy **Thread** nastavuje vlákno jako **vlákno démona**. Musí být zavolána před spuštěním vlákna. Vlákno démona je prosté vlákno v pozadí a je podřízené vláknu, které ho vytvořilo, takže když vlákno, které vytvořilo vlákno démona skončí, zanikne toto vlákno démona také.

Vlákno, které není vlákno démonů, se nazývá **vlákno uživatele**. Vlákno uživatele není závislé na vláknu, které jej vytvořilo. Implicitní vlákno, které obsahuje metodu **main()**, je uživatelské vlákno a jiná další uživ. vlákna běží i po skončení metody **main()**.

Metody

Sleep()

Metoda **sleep()** objektu **Thread** slouží k pozdržení provádění vlákna na počet milisekund, které jsou zadány v argumentu. Když je vlákno pozdrženo, dostanou prostor k provádění jiná vlákna. Jaké vlákno dostane prostor, resp. jaké z čekajících vláken se začne provádět, záleží na časovém plánovači JVM, takže se může stroj od stroje lišit.

Interrupt()

Vlákno může signalizovat jinému vláknu, aby zastavilo provádění, voláním metody **interrupt()** objektu typu **Thread**. To samo o sobě příslušné vlákno nezastaví, jen mu nastaví příznak tak, aby ukazoval na požadavek na přerušení. Aby tento příznak měl nějaký smysl, tak musí být kontrolován. To zajišťuje např. metoda **sleep()**, která kontroluje, jestli byl požadavek na přerušení nastaven, a pokud ano, tak vyvolá výjimku **InterruptedException**. Tuto kontrolu provádí i např. metoda **wait()**. Tuto kontrolu lze provést i ručně voláním metody **isInterrupted()** pro odpovídající vlákno. To vrací **true**, pokud byla metoda **interrupt()** na tomto vlákně volána. Vyvoláním výjimky **InterruptedException** se zruší požadavek na přerušení a metoda **isInterrupted()** vrací **false**.

isAlive()

PDF created with pdfFactory trial version www.pdffactory.com

Protože samotným voláním metody **interrupt()** nedojde k ukončení běhu vlákna, lze jeho ukončení zjistit voláním metody **isAlive()**. To vrací **true**, pokud vlákno běží.

join()

Pokud je potřeba v jednom vlákně čekat, až zanikne (doběhne) nějaké jiné vlákno, tak lze použít metodu **join()** pro vlákno, které má zaniknout. Volání **join()** bez argumentu zadrží aktuální vlákno na tak dlouho, dokud určené vlákno nezanikne.

```
vlakno1.join();  // Zadrzi aktualni vlakno, dokud nezanikne vlakno1
```

Metodě **join()** lze předat hodnotu typu **long**, která určí počet milisekund, jak maximálně dlouho se má čekat na zánik vlákna.

```
vlakno1.join(1000);
```

yield()

Tato metoda umožňuje provádění jiným vláknům. Použije se, pokud chceme povolit běh jiným vláknům, pokud čekají, ale nechceme pozastavit provádění aktuálního vlákna na určitý časový interval. Zavoláním metody **sleep()** se vlákno pozastaví na určenou dobu, i když žádná ostatní vlákna nečekají. Na druhé straně volání metody **yield()** způsobí, že pokud žádná jiná vlákna nečekají na provádění, chod aktuálního vlákna se okamžitě obnoví.

Možnosti synchronizace

JVM scheduler přiděluje CPU čas nespecifikovaným způsobem některému vláknu ve stavu runnable s nejvyšší prioritou. Vlákna se tedy vykonávají v náhodném pořadí. V určitých situacích (např. při přístupu ke sdílené paměti) je však potřeba vykonávání vláken řídit, abychom zamezili chybám, které jinak mohou nastat (deadlock, race condition – viz níže).

Běh vlákna může být na nějakou dobu přerušen kdykoli během jeho provádění. Pak se může stát, že nějaká data, se kterými jsme pracovali, se mohla změnit vlivem jiného vlákna. Protože my o tom nevíme, pracujeme pak nad neplatnými daty. To samozřejmě může způsobit velmi vážné problémy.

Například se může stát něco takového:

1. **Úředník** zkontroluje stav **účtu** zákazníka, který je 500 Kč.
2. Zákazník si ve stejnou dobu vybírá z bankomatu. **Bankomat** si také zjistí stav **účtu**.
3. **Úředník** připíše částku 100 na **účet** a navýší ho na 600 Kč.
4. Z **bankomatu** se vybere 100 Kč, takže změní **účet** na 400 Kč.
5. **Úředník** zapíše stav **účtu** na 600 Kč.
6. **Bankomat zapíše stav** účtu **na 400 Kč.**

Každý bod představuje nějakou dobu běhu vlákna **Úředník** nebo vlákna **Bankomat**. Problém je zřejmý. Dotaz na aktuální stav a zápis nového stavu **účtu** jsou dvě rozdílné operace. Dokud tomu tak je, nelze zaručit, že k uvedenému problému nedojde.

Proto je třeba pro sdílené zdroje (zde to byl **účet**), ke kterým má přístup více vláken, zavést opatření, aby jiné vlákno neměnilo zdroj, během doby, co se zdrojem pracuje aktuální vlákno. Jedním ze způsobů je použití synchronizace vláken.

Synchronizace

Synchronizace řeší, aby v případě, že několik vláken přistupuje k jednomu zdroji, mohlo tento přístup získat jen jedno vlákno.

Synchronizaci lze provést dvěma způsoby:

1. synchronizovat metody
2. synchronizovat bloky kódu

Synchronizování metod

Jakékoli metody nějakého objektu lze nastavit tak, aby v daném časovém bodu mohla běžet jen jedna z nich. Toho se dosáhne použitím klíčového slova **synchronized**.

```
public synchronized void metoda1() {
}

public synchronized void metoda2() {
}
```

Nyní může v jednom okamžiku běžet jen jedna ze synchronizovaných metod dané instance třídy. Teprve když se synchronizovaná metoda, která se právě provádí, dokončí, může se začít provádět další synchronizovaná metoda stejné instance třídy. Smyslem je, aby každá synchr. metoda měla exkluzivní přístup k objektu během svého provádění.

Proces synchronizace využívá vnitřní **zámek**, který je přiřazen každému objektu. Tento zámek je jakýsi druh příznaku, který je nastaven na začátku provádění synchr. metody. Každá synchr. metoda kontroluje, zda byl nějakou jinou metodou nastaven zámek a nezačne se provádět, dokud není tento zámek uvolněn. Pozor, to nebrání, aby nemohlo souběžně běžet

PDF created with pdfFactory trial version www.pdffactory.com

více vláken dvou různých instancí stejné třídy. Vždy se kontroluje jen souběžný přístup k jedné instanci.

V případě synchronizovaných *statických metod* se používá zámek na úrovni třídy, který nemá vliv na zámky instancí této třídy.

Synchronizace bloků kódu

Možnosti synchronizace bloku kódu nebo příkazu jsou mnohem větší, protože lze nejenom přesněji určit důležitou část, která vyžaduje synchronizaci (čím menší část kódu je synchronizována, tím méně dochází k blokování ostatních vláken), ale lze také určit objekt, který se má k synchronizaci použít. Synchronizovaný blok tedy můžeme uzamknout pomocí zámku, kteréhokoli objektu. Tím je zajištěno, že žádný jiný blok nebo metoda, synchronizované zámkem stejného objektu se nemůžou provádět.

```
synchronized(nejakyObjekt) {
  // synchr. blok
}
```

Velkým používáním synchronized bloků nebo metod je program značně omezován, protože jsou ostatní vlákna nucena čekat. Mělo by se tedy používat jen v nezbytné míře.

Deadlock (uváznutí)

Deadlock je když dvě (nebo více vláken) uvázne a žádné nemůže pokračovat, protože navzájem čekají až to druhé (nebo ta ostatní) uvolní zámky na společně synchronizovaném objektu.

Viz příklad:

Dva chudí lešetínští kováři v mají jedno společné nářadí: kladivo a kleště. Nedohodnou-li se, dojde časem k zlobně umrtvujícímu nicnedělání, tzv. deadlocku čili smrtelnému zaklesnutí.

Přihodí se to takto:

1. První uchopil kladivo a potřebuje ještě kleště.
2. Mezitím však druhý uchopil kleště a čeká až bude kladivo volné.
3. Nikdo to za ně nevyřeší a tak oba čekají a čekají ... čímž živnosti uvíznou

Přitom stačí rozumná dohoda - budeš-li potřebovat oba nástroje:

1. Nejdříve uchop kladivo a teprve pak sháněj kleště.
2. Pracuj.
3. Pak nejdříve pusť kleště, potom kladivo.

Ta zaručuje momentálnímu držiteli kladiva, že kleště budou časem volné.

Kovářů může být ve městě i více.

Přepsaným řešením je:

```
synchronized ( kladivo ) {
  synchronized ( kleste ) {
    //  work
  }
}
```

Deadlock je vážný problém a u složitých programů je velmi složité zajistit, aby k jeho docházení nedocházelo, proto je důležitý správný návrh aplikace.

Race condition (souběh) a atomicita

Race condition (souběh) je situace, kdy při přístupu dvou nebo více vláken/procesů ke sdíleným datům (soubor nebo proměnná) dojde k chybě, přestože každé z vláken/procesů se samostatně chová korektně.

Problémy se souběhem mohou být rozděleny do dvou kategorií:

- Zásah, který byl způsoben důvěryhodným procesem či vláknem, tj. procesem či vláknem, které je součástí naší aplikace.
- Zásah, který byl způsoben nedůvěryhodným procesem, tj. procesem který není součástí naší aplikace. Tímto procesem může být např. jiná aplikace zapisující do stejného souboru nebo útočný program, který chce souběh vyvolat.

Chybné chování je způsobeno kritickou závislostí na relativním načasování událostí v jednotlivých procesech. Data jsou modifikována některým procesem v době, kdy s nimi jiný proces provádí několik operací, o kterých se předpokládalo, že budou provedeny jako jeden nedělitelný celek – **atomicky**.

Atomicita operace neboli nedělitelnost znamená, že daná činnost (operace) se provede najednou. Nemůže tedy být přerušena nějakou jinou operací a později dokončena. V Javě je zajištěna atomicita zápisu a čtení 32-bitových či menších proměnných.

V následujícím kódu Java zajišťuje atomicitu přiřazení **b = a;**:

PDF created with pdfFactory trial version www.pdffactory.com

```
int a = 5;
int b;
b = a;
```

Následná ukázka inkrementace **a++;** však již atomickou operací není. Inkrementace se skládá ze tří operací – přečtení hodnoty proměnné **a**, inkrementace o jedna a zápisu zpět do proměnné **a**.

```
int a = 5;
a++;
```

Pro zajištění atomicity proměnných a polí lze využít balíček **java.util.concurrent.atomic**, který obsahuje třídy **AtomicBoolean**, **AtomicLong**, **AtomicIntegerArray** apod. Každá z těchto tříd poskytuje běžné operace, u kterých je zajištěna jejich atomicita.

Komunikace mezi vlákny

Používat cyklus čekání

```
while (true) {
  //neco delej
  sleep(100);
}
```

není příliš efektivní. Jsou efektivnější způsoby. Třída **Object** definuje metody **wait()**, **notify()**, **notifyAll()**. Tyto metody lze volat pouze zevnitř synchronizovaných metod nebo bloků. Pokud jsou zavolány odjinud, bude vyvolána výjimka **IllegalMonitorStateException**.

wait()

Existují tři přetížené verze této metody. Tato metoda pozastaví aktuální vlákno až do volání metody **notify()** nebo **notifyAll()** pro objekt, ke kterému patří zmíněná metoda **wait()**. Když se zavolá jakákoli verze metody **wait()**, vlákno uvolní synchronizační zámek, který má na objektu, takže se může provádět jakákoli jiná metoda nebo blok kódu, které jsou synchronizovány stejným objektem. Umožní se tak nejen volání metod **notify()** nebo **notifyAll()** jiným vláknem, ale jiné vlákno může také volat **wait()** pro tentýž objekt. **Wait()** může volat **InteruptedException**, musí být tedy v bloku **try-catch**.

wait(long timeMs)

Pozastaví vlákno, dokud neuplyne počet milisekund v argumentu nebo pokud není dříve volána metoda **notify()** nebo **notifyAll()** objektu, ke kterému **wait()** patří.

wait(long timeMs, int nano)

Funguje stejně jako předešlá metoda. Navíc umožňuje určit čas v nanosekundách.

notify()

Tímto se znovu spustí vlákno, které volalo metodu **wait()** objektu, kterému patří metoda **notify()**. Pokud pro tento objekt volalo **wait()** více objektů, není žádná kontrola nad tím, kterému z těchto vláken bude **notify()** určeno, a v takovém případě bude lepší použít **notifyAll()**. Pokud neexistují žádná čekající vlákna, metoda nedělá nic.

Jak to funguje

Jeden blok může volat **wait()** k pozastavení svých operací, dokud nějaká jiná metoda nebo blok kódu, synchronizovaný stejným objektem, jej nějakým způsobem nezmění a nezavolá **notify()**, čímž naznačí dokončení změny. Typicky bude **wait()** vláknem voláno, protože není nastavena nějaká vlastnost objektu, kterým je toto vlákno synchronizováno, nebo není splněna nějaká podmínka závisející na akci nějakého jiného vlákna.

Hlavním rozdílem mezi **sleep()** a **wait()** je, že **wait()** uvolňuje všechny objekty, které jsou aktuálním vláknem uzamknuty, zatímco **sleep()** tak nečiní.

Typickým použitím je:

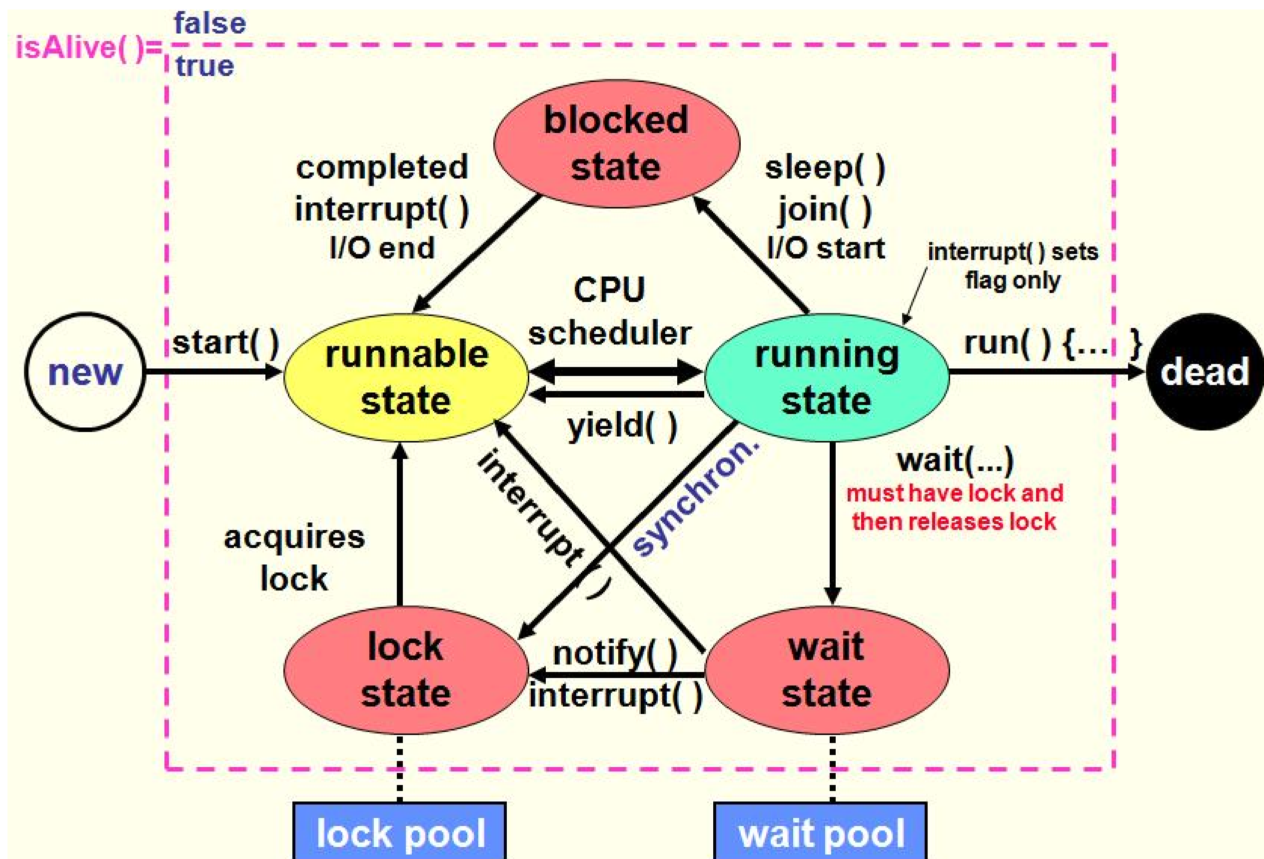
```
synchronized(aObjekt) {
  while(podmínka-není-splněna)
    objekt.wait();

  //podmínka splněna, lze pokračovat
}
```

Zavoláním **wait()** se vlákno pozastaví. Když jiné vlákno synchroniz. stejným objektem zavolá **notify()** nebo **notifyAll()**, tak je umožněno znovu provést kontrou ve *while* cyklu. Toto celé se opakuje, dokud není podmínka splněna a pak pokračuje kód vlákna.

Pokud existují více než dvě vlákna synchronizovaná jedním objektem, je obecně lepší používat metodu **notifyAll()**. Kdyby se volalo jen **notify()**, tak je otevřena možnost, že vlákno, které bylo spuštěno, zavolá opět **wait()** a tak budou všechna vlákna čekat jedno na druhé bez možnosti dalšího pokračování.

PDF created with pdfFactory trial version www.pdffactory.com

Stavy vláken



Vlákno si musí dynamicky pamatovat:

- v které metodě se právě nalézá a kam se pak vrátit,
- své lokální proměnné (obé je v aktivačním záznamu na zásobníku),
- adresu aktuálního příkazu v aktuální metodě (program counter).

Příklady

Příklad 1 - použití rozhraní Runnable

Spustí tři vlákna. Každé vlákno má své jméno, příjmení a čas čekání. Program kombinuje křestní jména a příjmení všech vláken dohromady.

```
import java.io.IOException;

public class JumbleNames implements Runnable
{
  private String firstName;                    // Store for first name
  private String secondName;                   // Store for second name
  private long aWhile;                         // Delay in milliseconds

  // Constructor
  public JumbleNames(String firstName, String secondName, long delay)
  {
    this.firstName = firstName;                // Store the first name
    this.secondName = secondName;              // Store the second name
    aWhile = delay;                            // Store the delay
  }

  // Method where thread execution will start
  public void run()
  {
    try
    {
      while(true)                              // Loop indefinitely...
      {
        System.out.print(firstName);           // Output first name
        Thread.sleep(aWhile);                  // Wait aWhile msec.
        System.out.print(secondName+"\n");     // Output second name
      }
    }
    catch(InterruptedException e)              // Handle thread interruption
    {
```


PDF created with pdfFactory trial version www.pdffactory.com

```
      System.out.println(firstName + secondName + e);     // Output the exception
    }
  }

  public static void main(String[] args)
  {
    // Create three threads
    Thread first = new Thread(new JumbleNames("Hopalong ", "Cassidy ", 200L));
    Thread second = new Thread(new JumbleNames("Marilyn ", "Monroe ", 300L));
    Thread third = new Thread(new JumbleNames("Slim ", "Pickens ", 500L));

    // Set threads as daemon
    first.setDaemon(true);
    second.setDaemon(true);
    third.setDaemon(true);

    System.out.println("Press Enter when you have had enough...\n");
    first.start();                                  // Start the first thread
    second.start();                                 // Start the second thread
    third.start();                                  // Start the third thread
    try
    {
      System.in.read();                             // Wait until Enter key pressed
      System.out.println("Enter pressed...\n");
    }
    catch (IOException e)                           // Handle IO exception
    {
      System.out.println(e);                        // Output the exception
    }
    System.out.println("Ending main()");
    return;
  }
}
```

Výstup:

```
Press Enter when you have had enough...

Hopalong Slim Marilyn Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Pickens
Slim Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Pickens
Slim Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Pickens
Slim Monroe
Marilyn Cassidy
Hopalong Cassidy
Hopalong Monroe
Marilyn
Pickens
Enter pressed...

Ending main()
```

Příklad 2 - banka bez použití synchronizovaných metod

Jedná se o jednoduchý příklad, který simuluje procesy v bance. Několik úředníků (vláken) pracuje se stejným účtem. Před skončením programu se vypíše statistika, ze které lze vidět, že na účtu je jiná hodnota (Final Balance), než by měla být (Should be). Příklad má za účel ukázat, kam může vést nesprávné použití vláken. Na konci příkladu je naznačená možná náprava drobnou úpravou třídy Bank. Ačkoli to napravuje chybu a program poté bude s účtem pracovat správně, tak je program (banka) málo efektivní. Jedno vlákno (úředník) nic neprovádí, když druhý provádí transakci. Vždy pracuje jen jedno vlákno. Tím jsou vlastně ostatní vlákna zbytečná. V dalších příkladech se ukáže, co se s tím dá dělat.

Banka:

```
// Define the bank
class Bank
{
  // Perform a transaction
  public void doTransaction(Transaction transaction)
  {
    int balance = transaction.getAccount().getBalance();   // Get current balance

    switch(transaction.getTransactionType())
    {
      case Transaction.CREDIT:
        // Credits require a lot of checks...
        try
        {
          Thread.sleep(100);
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
        balance += transaction.getAmount();           // Increment the balance
        break;

      case Transaction.DEBIT:
        // Debits require even more checks...
        try
```


PDF created with pdfFactory trial version www.pdffactory.com

```
        {
          Thread.sleep(150);
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
        balance -= transaction.getAmount();          // Decrement the balance
        break;

      default:                                       // We should never get here
        System.out.println("Invalid transaction");
        System.exit(1);
   }
    transaction.getAccount().setBalance(balance);    // Restore the account balance
  }
}
```

Účetní transakce:

```
class Transaction
{
  // Transaction types
  public static final int DEBIT = 0;
  public static final int CREDIT = 1;
  public static String[] types = {"Debit","Credit"};

  // Constructor
  public Transaction(Account account, int transactionType, int amount)
  {
    this.account = account;
    this.transactionType = transactionType;
    this.amount = amount;
  }

  public Account getAccount()
  {  return account;  }

  public int getTransactionType()
  {  return transactionType;  }

  public int getAmount()
  {  return amount;  }

  public String toString()
  {
    return types[transactionType] + " A//C: " + ": $" + amount;
  }

  private Account account;
  private int amount;
  private int transactionType;
}
```

Bankovní účet:

```
// Defines a customer account
public class Account
{
  // Constructor
  public Account(int accountNumber, int balance)
  {
    this.accountNumber = accountNumber;            // Set the account number
    this.balance = balance;                        // Set the initial balance
  }

  // Return the current balance
  public int getBalance()
  {  return balance;  }

  // Set the current balance
  public void setBalance(int balance)
  {  this.balance = balance;  }

  public int getAccountNumber()
  {  return accountNumber;  }

  public String toString()
  {
    return "A//C No. "+accountNumber+" : $"+balance;
  }

  private int balance;                             // The current account balance
  private int accountNumber;                       // Identifies this account
}
```

Bankovní úředník:

```
public class Clerk implements Runnable
{
  private Bank theBank;               // The employer - an electronic marvel
  private Transaction inTray;         // The in-tray holding a transaction

  // Constructor
  public Clerk(Bank theBank)
  {
    this.theBank = theBank;           // Who the clerk works for
    inTray = null;                    // No transaction initially
  }

  // Receive a transaction
  public void doTransaction(Transaction transaction)
```


PDF created with pdfFactory trial version www.pdffactory.com

```
  {  inTray = transaction;  }

  // The working clerk...
  public void run()
  {
    while(true)
    {
      while(inTray == null)           // No transaction waiting?
      {
        try
        {
          Thread.sleep(150);          // Then take a break...
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
      }

      theBank.doTransaction(inTray);
      inTray = null;                  // In-tray is empty
    }
  }
  // Busy check
  public boolean isBusy()
  {
    return inTray != null;            // A full in-tray means busy!
  }
}
```

Bankovní operace:

```
import java.util.Random;

public class BankOperation
{
  public static void main(String[] args)
  {
    int initialBalance = 500;     // The initial account balance
    int totalCredits = 0;         // Total credits on the account
    int totalDebits =0;           // Total debits on the account
    int transactionCount = 20;    // Number of debits and credits

    // Create the account, the bank and the clerks...
    Bank theBank = new Bank();                        // Create a bank
    Clerk clerk1 = new Clerk(theBank);                // Create the first clerk
    Clerk clerk2 = new Clerk(theBank);                // Create the second clerk
    Account account = new Account(1, initialBalance); // Create an account

    // Create the threads for the clerks as daemon, and start them off
    Thread clerk1Thread = new Thread(clerk1);
    Thread clerk2Thread = new Thread(clerk2);
    clerk1Thread.setDaemon(true);                   // Set first as daemon
    clerk2Thread.setDaemon(true);                   // Set second as daemon
    clerk1Thread.start();                           // Start the first
    clerk2Thread.start();                           // Start the second

    // Generate the transactions of each type and pass to the clerks
    Random rand = new Random();                     // Random number generator
    Transaction transaction;                        // Stores a transaction
    int amount = 0;                                 // stores an amount of money
    for(int i = 1; i <= transactionCount; i++)
    {
      amount = 50 + rand.nextInt(26);               // Generate amount of $50 to $75
      transaction = new Transaction(account,           // Account
                              Transaction.CREDIT, // Credit transaction
                              amount);            //  of amount
      totalCredits += amount;                   // Keep total credit tally

      // Wait until the first clerk is free
      while(clerk1.isBusy()) {
        try
        {
          Thread.sleep(25);                           // Busy so try later
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
      }
      clerk1.doTransaction(transaction);            // Now do the credit

      amount = 30 + rand.nextInt(31);               // Generate amount of $30 to $60
      transaction = new Transaction(account,           // Account
                              Transaction.DEBIT,  // Debit transaction
                              amount);            //  of amount
      totalDebits += amount;                        // Keep total debit tally
      // Wait until the second clerk is free
      while(clerk2.isBusy()) {
        try
        {
          Thread.sleep(25);                           // Busy so try later
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
      }
      clerk2.doTransaction(transaction);            // Now do the debit
    }

    // Wait until both clerks are done
    while(clerk1.isBusy() || clerk2.isBusy())
    {
      try
      {
```


PDF created with pdfFactory trial version www.pdffactory.com

```
        Thread.sleep(25);
      }
      catch(InterruptedException e)
      {
        System.out.println(e);
      }
    }

    // Now output the results
    System.out.println(
          "Original balance : $" + initialBalance+"\n" +
          "Total credits    : $" + totalCredits+"\n" +
          "Total debits     : $" + totalDebits+"\n" +
          "Final balance    : $" + account.getBalance() + "\n" +
          "Should be        : $" + (initialBalance + totalCredits - totalDebits));
  }
}
```

Výstup (protože jsou částky transakcí generovány náhodně, tak jsou ve výstupu pokaždé jiná čísla):

```
Original balance : $500
Total credits    : $1262
Total debits     : $914
Final balance    : $-15
Should be        : $848
```

Program lze napravit deklarováním operace na účtu jako synchronizované. Tím se zamezí tomu, aby se jeden úředník dostal k účtu, když druhý ještě pracuje:

```
// Define the bank
class Bank
{
  // Perform a transaction
  synchronized public void doTransaction(Transaction transaction)
  {
    // The same code as before
  }
}
```

Slovníček pojmů

Zdroje

https://webdev.felk.cvut.cz/courses/Y36PJV/prednasky/prednaska06 [https://webdev.felk.cvut.cz/courses/Y36PJV/prednasky/prednaska06]

https://webdev.felk.cvut.cz/courses/Y36PJV/cviceni/cviceni04#vlakna [https://webdev.felk.cvut.cz/courses/Y36PJV/cviceni/cviceni04#vlakna]

kniha Java 5, Ivor Horton

http://moon.felk.cvut.cz/~xballner/vyuka/x36api/lectures/API_3_synchronizace.pdf [http://moon.felk.cvut.cz/~xballner/vyuka/x36api/lectures/API_3_synchronizace.pdf]

http://digitalne.centrum.cz/procesy-vlakna-a-process-monitor/ [http://digitalne.centrum.cz/procesy-vlakna-a-process-monitor/]

spolecne/spol31.txt · Poslední úprava: 2010/06/06 17:20 autor: Kucera5

PDF created with pdfFactory trial version www.pdffactory.com