Vícevláknové aplikace – modely a příklady

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 8

B6B36PJV – Programování v JAVA

Část 1 – Využití vláken v GUI

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker

Část 2 – Modely vícevláknových aplikací

Modely více-vláknových aplikací

Prostředky ladění

Část 3 – Příklad – GUI aplikace Simulátor/Plátno

GUI s plátnem

- Struktura aplikace
- Struktura simulátoru
- Struktura grafického rozhraní
- Praktické ukázky

Část I

Část 1 – Využití vláken v GUI

Vlákna v GUI (Swing)

- Vlákna můžeme použít v libovolné aplikaci a tedy i v aplikaci s GUI.
- Vykreslování komponent Swing se děje v samostatném vlákně vytvořeném při inicializaci toolkitu
- Proto je vhodné aktualizaci rozhraní realizovat notifikací tohoto vlákna z jiného

  Snažíme se pokud možno vyhnout asynchronnímu překreslování z více vláken – race condition
- Zároveň se snažíme oddělit grafickou část od výpočetní (datové) části aplikace (MVC)

`http://docs.oracle.com/javase/tutorial/uiswing/concurrency`

Samostatné výpočetní vlákno pro výpočetní model v aplikaci DemoBarComp

- Třídu `Model` rozšíříme o rozhraní **Runnable**
- Vytvoříme novou třídu `ThreadModel`
  - Voláním metody `compute` spustíme samostatné vlákno
  - Musíme zabránit opakovanému vytváření vlákna

    Příznak `computing`
  - Metodu uděláme synchronizovanou
- Po stisku tlačítka stop ukončíme vlákno

  Implementujeme třídu `StopListener`
- Ve třídě `ThreadModel` implementuje metodu `stopComputation`

  Nastaví příznak ukončení výpočetní smyčky `end`

`lec08/DemoBarComp-simplethread`

Po spuštění výpočtu je GUI aktivní, ale neaktualizuje se *progress bar*, je nutné vytvořit vazbu s výpočetního vlákna – použijeme návrhový vzor **Observer**

Návrhový vzor **Observer**

- Realizuje abstraktní vazbu mezi objektem a množinou pozorovatelů
- Pozorovatel je předplatitel (*subscriber*) změn objektu
- Předplatitelé se musejí registrovat k pozorovanému objektu
- Objekt pak informuje (notifikuje) pozorovatele o změnách
- V Javě je řešen dvojicí třídy **Observable** a **Observer**

Výpočetní model jako **Observable** objekt 1/4

- **Observable** je abstraktní třídy
- `ThreadModel` již dědí od `Model`, proto vytvoříme nový **Observable** objekt jako instanci privátní třídy `UpdateNotificator`
- Objekt `UpdateNotificator` použijeme k notifikaci registrovaných pozorovatelů

```
public class ThreadModel extends Model implements
    Runnable {
   private class UpdateNotificator extends Observable {
      private void update() {
         setChanged();      // force subject change
         notifyObservers(); // notify reg. observers
      }
   }
   UpdateNotificator updateNotificator;

   public ThreadModel() {
      updateNotificator = new UpdateNotificator();
```

lec08/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 2/4

- Musíme zajistit rozhraní pro přihlašování a odhlašování pozorovatelů
- Zároveň nechceme měnit typ výpočetního modelu ve třídě `MyBarPanel`
- Musíme proto rozšířit původní výpočetní model `Model`

```
public class Model {
 public void unregisterObserver(Observer observer) {...}
 public void registerObserver(Observer observer) {...}
 ...
```

- Ve třídě `ThreadModel` implementujeme přihlašování/odhlašování odběratelů

```
@Override
public void registerObserver(Observer observer) {
   updateNotificator.addObserver(observer);
}
@Override
public void unregisterObserver(Observer observer) {
   updateNotificator.deleteObserver(observer);
}
```

lec08/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 3/4

- Odběratele informujeme po dílčím výpočtu v metodě `run` třídy `ThreadModel`

```
public void run() {
    ...
    while (!computePart() && !finished) {
        updateNotificator.update();
```

- Panel `MyBarPanel` je jediným odběratelem a implementuje rozhraní **Observer**, tj. metodu **update**

```
public class MyBarPanel extends JPanel implements
    Observer {
    @Override
    public void update(Observable o, Object arg) {
        updateProgress(); //arg can be further processed
    }
    private void updateProgress() {
        if (computation != null) {
            bar.setValue(computation.getProgress());
        }
    }
```

lec08/DemoBarComp-observer

Výpočetní model jako **Observable** objekt 4/4

- Napojení pozorovatele `MyBarPanel` na výpočetní model `Model` provedeme při nastavení výpočetního modelu

```
public class MyBarPanel extends JPanel implements
    Observer {
public void setComputation(Model computation) {
        if (this.computation != null) {
            this.computation.unregisterObserver(this);
        }
        this.computation = computation;
        this.computation.registerObserver(this);
    }
```

- Při změně modelu nesmíme zapomenout na odhlášení od původního modelu

Nechceme dostávat aktualizace od původního modelu, pokud by dál existoval.

`lec08/DemoBarComp-observer`

Výpočetní vlákno ve Swing

- Alternativně můžeme využít třídu **SwingWorker**
- Ta definuje metodu **doInBackground()**, která zapouzdřuje výpočet na „pozadí" v samostatném vláknu
  - V těle metody můžeme publikovat zprávy voláním metody `publish()`
- Automaticky se také „napojuje" na události v „grafickém vlákně" a můžeme předefinovat metody
  - `process()` – definuje reakci na publikované zprávy
  - `done()` – definuje reakci po skočení metody `doInBackground()`

`http://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html`

Příklad použití třídy SwingWorker 1/3

- Vlákno třídy `SwingWorker` využijeme pro aktualizaci GUI s frekvencí 25 Hz
- V metodě `doInBackground` tak bude periodicky kontrolovat, zdali výpočetní vlákno stále běží
- Potřebujeme vhodné rozhraní třídy `Model`, proto definujeme metodu `isRunning()`

```
public class Model {
    ...
    public boolean isRunning() { ... }
}
```

Není úplně vhodné, ale vychází z postupného rozšiřování původně nevláknového výpočtu. Lze řešit využitím přímo `ThreadModel`.

- Metodu `isRunning` implementujeme ve vláknovém výpočetním modelu `ThreadModel`

```
public class ThreadModel ...
    public synchronized boolean isRunning() {
        return thread.isAlive();
    }
```

`lec08/DemoBarComp-swingworker`

Příklad použití třídy SwingWorker 2/3

- Všechna ostatní rozšíření realizujeme pouze v rámci GUI třídy `MyBarPanel`
- Definujeme vnitřní třídy **MySwingWorker** rozšiřující **SwingWorker**

```
public class MyBarPanel extends JPanel {
    public class MySwingWorker extends SwingWorker<Integer
     , Integer> { ... }

    MySwingWorker worker;
```

- Tlačítko **Compute** připojíme k instanci `MySwingWorker`

```
private class ComputeListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (!worker.isDone()) { //only single worker
            status.setText("Start computation");
            worker.execute();
        }
    }
}
```

lec08/DemoBarComp-swingworker

Příklad použití třídy SwingWorker 3/3

- Ve třídě `MySwingWorker` definujeme napojení periodické aktualizace na *progress bar*

```
public class MySwingWorker extends SwingWorker {
    @Override
    protected Integer doInBackground() throws Exception {
        computation.compute();
        while (computation.isRunning()) {
            TimeUnit.MILLISECONDS.sleep(40); //25 Hz
            publish(new Integer(computation.getProgress()));
        }
        return 0;
    }
    protected void process(List<Integer> chunks) {
        updateProgress();
    }
    protected void done() {
        updateProgress();
    }
}
```

lec08/DemoBarComp-swingworker

- S výhodou využíváme přímého přístupu k `updateProgress`

Zvýšení interaktivity aplikace

- Po stisku tlačítka **Stop** aplikace čeká na doběhnutí výpočetního vlákna
- To nemusí být důvod k zablokování celého GUI
- Můžeme realizovat „vypnutí" tlačítek `Compute` a `Stop` po stisku `Stop`
- Jejich opětovnou aktivaci můžeme odložit až po ukočení běhu výpočetního vlákna

Část II

Část 2 – Modely vícevláknových aplikací

Kdy použít vlákna?

Vlákna je výhodné použít, pokud aplikace splňuje některé následující kritérium:

- Je složena z nezávislých úloh.
- Může být blokována po dlouho dobu.
- Obsahuje výpočetně náročnou část.
- Musí reagovat na asynchronní události.
- Obsahuje úlohy s nižší nebo vyšší prioritou než zbytek aplikace.

Typické aplikace

- **Servery** - obsluhují více klientů najednou. Obsluha typicky znamená přístup k několika sdíleným zdrojům a hodně vstupně výstupních operací (I/O).
- **Výpočetní aplikace** - na víceprocesorovém systému lze výpočet urychlit rozdělením úlohy na více procesorů.
- **Aplikace reálného času** - lze využít specifických rozvrhovačů. Vícevláknová aplikace je výkonnější než složité asynchronní programování, neboť vlákno čeká na příslušnou událost namísto explicitního přerušování vykonávání kódu a přepínání kontextu.

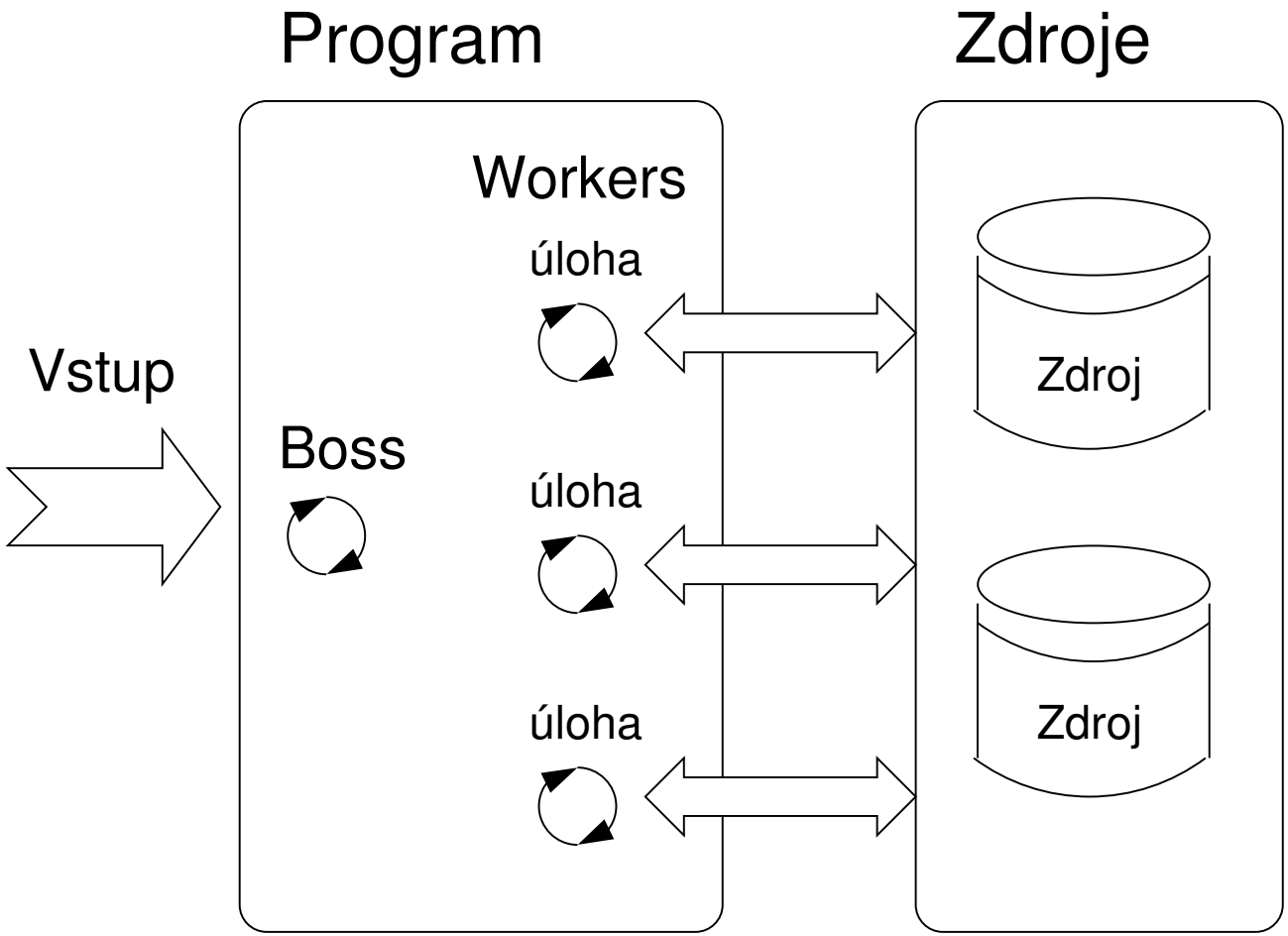
Modely vícevláknových aplikací

Modely řeší způsob vytváření a rozdělování práce mezi vlákny.

- **Boss/Worker** - hlavní vlákno řídí rozdělení úlohy jiným vláknům.
- **Peer** - vlákna běží paralelně bez specifického vedoucího.
- **Pipeline** - zpracování dat sekvencí operací.

Předpokládá dlouhý vstupních proud dat.

Boss/Worker model



Boss/Worker rozdělení činnosti

- Hlavní vlákno je zodpovědné za vyřizování požadavků.
- Pracuje v cyklu:
  1. příchod požadavku,
  2. vytvoření vlákna pro řešení příslušného úkolu,
  3. návrat na čekání požadavku.
- Výstup řešení úkolu je řízen:
  - Příslušným vláknem řešícím úkol.
  - Hlavním vláknem, předání využívá synchronizační mechanismy.

Boss/Worker příklad

Příklad Boss/Worker model

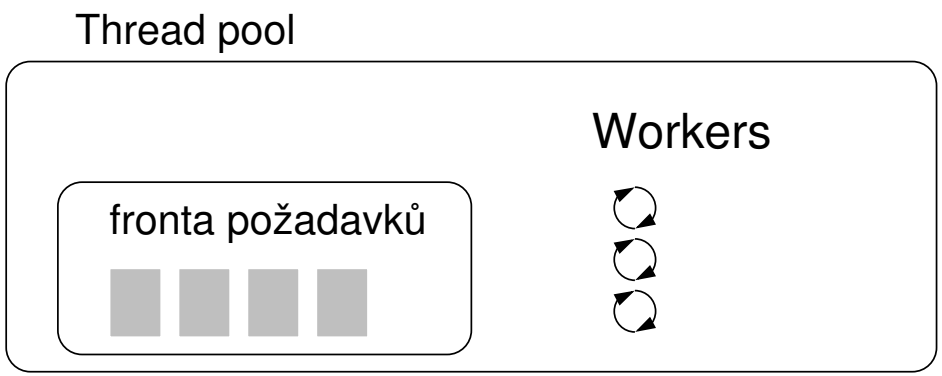
```
1  //Boss
2  main() {
3   while(1) {
4     switch(getRequest()) {
5       case taskX :
        create_thread(taskX);
6       case taskY :
        create_thread(taskY);
7           .
8           .
9     }
10   }
11 }
```

```
1  //Worker
2  taskX() {
3    řešení úlohy,
       synchronizace v
       případě sdílených
       zdrojů;
4    done;
5  }
6
7  taskY() {
8    řešení úlohy,
       synchronizace v
       případě sdílených
       zdrojů;
9    done;
10 }
```

C style

Thread Pool

- Hlavní vlákno vytváří vlákna dynamicky podle příchozích požadavků.
- Režii vytváření lze snížit, vytvořením vláken dopředu (Thread Pool).
- Vytvořená vlákna čekají na přiřazení úkolu.



Thread Pool - vlastnosti

- Počet vytvořených vláken.
- Maximální počet požadavků ve frontě požadavků.
- Definice chování v případě plné fronty požadavků a žádného volného vlákna.

Například blokování příchozích požadavků.

Java Thread Pool - příklad

- `Executor` z balíku `java.util.concurrent` nabízí knihovní zprávu vláken
- Příklad – `ThreadPoolExecutor` nebo `ScheduledThreadPoolExecutor`
- Přijímá `Callable` objekty
  - Podobné `Runnable`
  - Mohou vracet nějakou hodnotu a vyhazovat vyjímky
- Při vložení vrací `Future` objekt obsahující stav a návratovou hodnotu
  - `isDone` testuje, zda je úloha již ukončena
  - `get` vrací návratovou hodnotu
  - `cancel` pošle žádost o zrušení úlohy

`lec08/ThreadPoolExample`

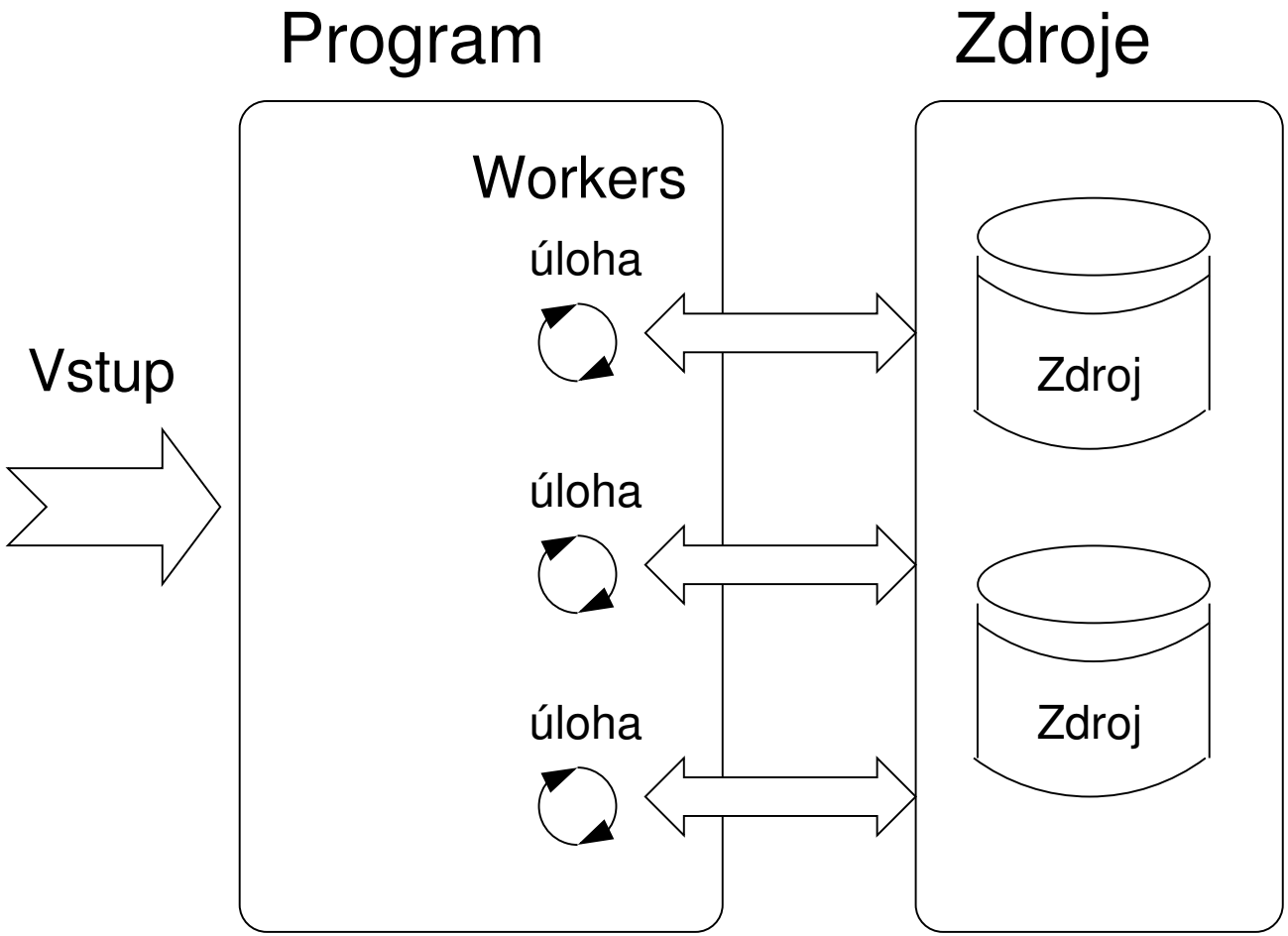
Java Thread Pool - příklad

```
private class CallableImpl implements Callable<String> {

 String name;

 public CallableImpl(int number) {
     name = "Callable " + number;
 }

 @Override
 public String call() {
     for (int i = 0; i < 5; i++) {
         System.out.println(name + " is doing something")
    ;
         try {
             Thread.sleep(200);
         } catch (InterruptedException ex) {
             System.out.println("Not sleeping ?? ...");
         }
     }
     return name + " is done !";
 }
}
```

Java Thread Pool - příklad

```
public class ThreadPoolExample {

 ExecutorService threadPool;

 public void test(int numberOfThreads) {

  threadPool = newFixedThreadPool(numberOfThreads);
  List<Future> futures = new ArrayList<>();

  for (int i = 1; i <= 10; i++) {
      Callable<String> aCall = new CallableImpl(i);
      Future<String> submit = threadPool.submit(aCall);
      futures.add(submit);
  }

  try {
      for (Future future : futures) {
          System.out.println(future.get());
      }
  } catch (Exception ex) { ... }

  threadPool.shutdown();
 }
}
```

Peer model


Program
Zdroje
Workers
úloha
úloha
úloha
Vstup
Zdroj
Zdroj

Peer model - vlastnosti

- Neobsahuje hlavní vlákno.
- První vlákno po vytvoření ostatních vláken:
  - se stává jedním z ostatních vláken (rovnocenným),
  - pozastavuje svou činnost do doby než ostatní vlákna končí.
- Každé vlákno je zodpovědné za svůj vstup a výstup.

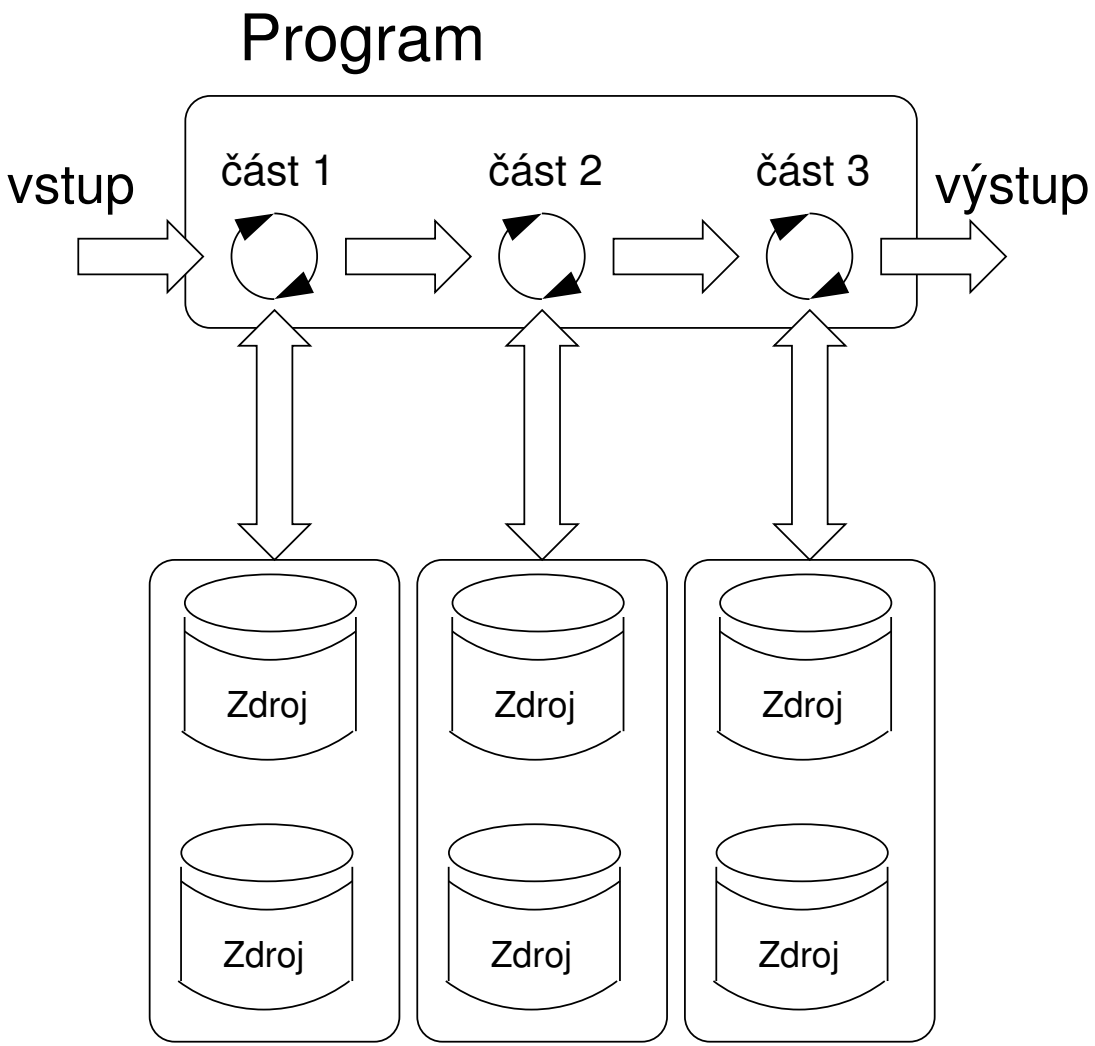
Peer model - příklad

Příklad Peer model

```
1  //Boss
2  main() {
3    create_thread(task1);
4    create_thread(task2);
5    .
6    .
7    start all threads;
8    wait for all threads;
9  }
```

```
1  //Worker
2  task1() {
3    čekáná na spuštení;
4    řešení úlohy,
       synchronizace v
       případě sdílených
       zdrojů;
5    done;
6  }
7
8  task2() {
9    čekáná na spuštení;
10   řešení úlohy,
       synchronizace v
       případě sdílených
       zdrojů;
11   done;
12 }
```

Zpracování proudu dat - Pipeline



Pipeline

- Dlouhý vstupní proud dat.
- Sekvence operací (částí zpracování), každá vstupní jednotka musí projít všemi částmi zpracování.
- V každé části jsou v daném čase, zpracovávány různé jednotky vstupu (nezávislost jednotek).

Pipeline model - příklad

Příklad Pipeline model

```
1  main() {
2    create_thread(stage1);
3    create_thread(stage2);
4    .
5    .
6    wait for all pipeline;
7  }
8  stage1() {
9   while(input) {
10    get next program
       input;
11    process input;
12    pass result to next
       stage;
13  }
14 }
```

```
1  stage2() {
2   while(input) {
3     get next input from
       thread;
4     process input;
5     pass result to next
       stage;
6   }
7  }
8  stageN() {
9   while(input) {
10    get next input from
       thread;
11    process input;
12    pass result to output;
13  }
14 }
```

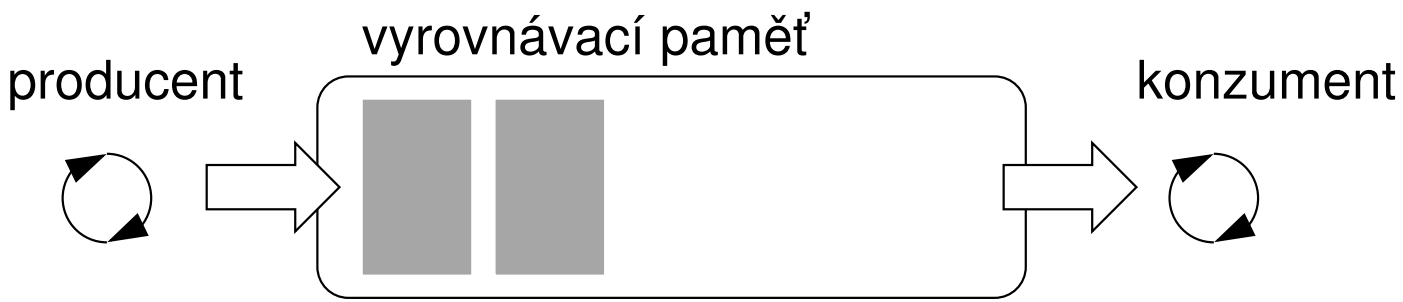
C style

Producent a konzument

Předávání dat mezi vlákny je realizováno vyrovnávací pamětí bufferem.

- Producent - vlákno, které předává data jinému vláknu.
- Konzument - vlákno, které přijímá data od jiného vlákna.

Přístup do vyrovnávací pamětí musí být synchronizovaný (exkluzivní přístup).



Funkce a paralelismus

Při paralelním běhu programu mohou být funkce volány vícenásobně. Funkce jsou :

- **reentrantní** - V jediném okamžiku může být stejná funkce vykonávána současně

  Např. vnořená obsluha přerušení

- **thread-safe** - Funkci je možné současně volat z více vláken

Dosažení těchto vlastností:

- Reentrantní funkce nezapisují do statických dat, nepracují s globálními daty.
- Thread-safe funkce využívají synchronizačních primitiv při přístupu ke globálním datům.

Vícevláknové aplikace a ladění

Hlavní problémy vícevláknových aplikací souvisí se synchronizací:

- **Uváznutí** – **deadlock**.
- **Souběh** (**race conditions**) - přístup více vláken ke sdíleným proměnným a alespoň jedno vlákno nevyužívá synchronizačních mechanismů. Vlákno čte hodnotu zatímco jiné vlákno zapisuje. Zápis a čtení nejsou atomické a data mohou být neplatná.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je nepotřebovat ladit.
- Toho lze dosáhnou kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je

  nepotřebovat ladit.

- Toho lze dosáhnou kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Prostředky ladění

- Nejlepším prostředkem ladění vícevláknových aplikací je

  nepotřebovat ladit.

- Toho lze dosáhnou kázní a obezřetným přístupem ke sdíleným proměnným.
- Nicméně je vhodné využívat ladící prostředí s minimální množinou vlastností.

Podpora ladění

Debugger:

- Výpis běžících vláken.
- Výpis stavu synchronizačních primitiv.
- Přístup k proměnným vláken.
- Pozastavení běhu konkrétního vlákna.
- Záznam průběhu běhu celého programu (kompletní obsah paměti a vstupů/výstup) a procházení záznamu

Logování:

- Problém uváznutí souvisí s pořadím událostí, logováním přístupu k zámkům lze odhalit případné špatné pořadí synchronizačních operací.

Poznámky - „problémy souběhu"

Problémy souběhu jsou typicky způsobeny nedostatkem synchronizace.

- Vlákna jsou asynchronní.

  Nespoléhat na to, že na jednoprocesorovém systému je vykonávání kódu synchronní.

- Při psaní vícevláknové aplikace předpokládejte, že vlákno může být kdykoliv přerušeno nebo spuštěno.

  Části kódu, u kterých je nutné zajistit pořadí vykonávání jednotlivými vlákny vyžadují synchronizaci.

- Nikdy nespoléhejte na to, že vlákno po vytvoření počká, může být spuštěno velmi brzy.

- Pokud nespecifikujete pořadí vykonávání vláken, žádné takové neexistuje.

  „*Vlákna běží v tom nejhorším možném pořadí. Bill Gallmeister*"

Poznámky - „problém uváznutí"

Problémy uváznutí souvisí s mechanismy synchronizace.

- Uváznutí (deadlock) se na rozdíl o souběhu mnohem lépe ladí.
- Častým problém je tzv. *mutex deadlock* způsobený pořadím získávání mutexů (zámku/monitorů).
- Mutex deadlock nemůže nastat, pokud má každé vlákno uzamčený pouze jeden mutex (*chce uzamknout*).
- Není dobré volat funkce s uzamčeným mutexem, obzvláště zamyká-li volaná funkce jiný mutex.
- Je dobré zamykat mutex na co možná nejkratší dobu.

V Javě odpovídá zámek krické sekci monitoru `synchronized(mtx){}`

http://www.javaworld.com/article/2076774/java-concurrency/programming-java-threads-in-the-real-world--part-1.html

Část III

Část 3 – Příklad – GUI aplikace Simulator/Plátno

Zadání

- Naším cílem je vytvořit simulátor „herního" světa
  - Ve světě mohou být různé objekty, které se mohou nezávisle pohybovat
  - Simulátor je „nezávislý" na vizualizaci
  - Simulátor běží v diskrétních krocích
- Vizualizaci herního světa se pokusíme oddělit od vlastního simulátoru
  - Každému objektu simulátoru přiřadíme grafický tvar, který se bude umět zobrazit na plátno
- Simulátor chceme ovládat tlačítky „Start/Stop"
- Svět simulátoru vizualizujeme na plátně
- Vizualizace plátna bude probíhat „nezávisle" na běhu simulátoru

Interaktivní hra vs Simulace

Základní struktura aplikace

- **Simulátor** – obsahuje svět a objekty
  - V zásadě se chová jako kolekce simulačních objektů

    `Iterable`
  - Simulace běží v samostatném vlákně s periodou `PERIOD`
  - Simulace probíhá v diskrétních časových okamžicích voláním metody `doStep` simulačních objektů
  - Má metodu pro zastavení vlákna `shutdown`
- Grafické rozhraní a vizualizace – obsahuje
  - Základní kontrolní tlačítka pro ovládání simulace (start/stop)
  - Plátno pro vykreslení dílčích objektů
  - Standardní vykreslovací Swing vlákno
  - Samostatné vlákno pro překreslování stavu simulátoru

    `SwingWorker s přeposíláním zpráv hlavnímu Swing vláknu`
  - Grafickou reprezentaci vykreslovaných objektů

    Vlastní vykreslení grafickými primitivy.

Simulator – World – SimObject

- `Simulator` – kolekce simulačních objektů
- `World` – definuje rozměry světa

  Pro jednoduchost identické jako rozměry okna/plátna
- `SimObject` – jednotné rozhraní simulačního objektu
  - Aktuální pozice objektu – `public Coords getPosition();`
  - Provedení jednoho simulačního kroku – objekt má definované chování `public void doStep();`
- Simulace probíhá ve smyčce:

```
while(!quit) {
    for(SimObject obj : objects) {
        obj.doStep();
    }
    Thread.sleep(PERIOD);
}
```

Struktura grafického rozhraní

- Hlavní okno aplikace obsahuje kontrolní tlačítka
  - Tlačítko `start` spouští simulaci
  - Tlačítko `stop` zastavuje běžící simulaci
- Vizualizace simulace probíhá ve vlastním plátně `MyCanvas`
- Simulační objekty mají svůj grafický tvar `Shape`
- Překreslení plátna probíhá periodicky

`SwingWorker()`

```
while(sim.isRunning()) {
    if (sim.isChanged()) {
        MyCanvas canvas = getSimCanvas();
        canvas.redraw();
        Thread.sleep(CANVAS_REFRESH_PERIOD);
    }
}
```

Základní koncept překreslení – neodpovídá kódu

Struktura plátna MyCanvas a vizualizace

- MyCanvas – reprezentuje kolekci vykreslitelných objektů – instance Drawable
- Každý objekt se umí vykreslit do grafického kontextu

```
public void redraw() {
    Graphics2D gd = getGraphics();
    for (Drawable obj : objects) {
        obj.draw(gd);
    }
}
```

- Vlastní tvar objektu je definován ve třídě Shape

```
abstract public class Shape implements Drawable {
    protected SimObject object;
    public Shape(SimObject object) {
        this.object = object;
    }
}
```

Příklad definice tvaru – ShapeMonster, ShapeNPC

- ShapeMonster

```
public class ShapeMonster extends Shape {
    public ShapeMonster(SimObject object) {
        super(object);
    }
    @Override
    public void draw(Graphics2D g2d) {
        Coords pt = object.getPosition();
        g2d.setColor(Color.RED);
        g2d.fillOval(pt.getX(), pt.getY(), 15, 15);
}   }
```

- ShapeNPC

```
public class ShapeNPC extends Shape {
   public ShapeNPC(SimObject object) {
      super(object);
   }
   @Override
   public void draw(Graphics2D g2d) {
      Coords pt = object.getPosition();
      g2d.setColor(Color.GREEN);
      g2d.fillRect(pt.getX(), pt.getY(), 15, 15);
} }
```

Vytvoření simulačních objektů a jejich tvarů

```
private Simulator sim;
private MyCanvas canvas;

public SimulatorGUI(Simulator sim, MyCanvas canvas) {
    this.sim = sim;
    this.canvas = canvas;
    createObjects();
}

public void createObjects() {
    World world = sim.getWorld();

    SimObject monster = new SimObjectMonster(world, 1, 1);
    sim.addObject(monster);
    canvas.addObject(new ShapeMonster(monster));

    SimObject npc = new SimObjectNPC(world, 400, 200);
    sim.addObject(npc);
    canvas.addObject(new ShapeNPC(npc));
}
```

Příklad – CanvasDemo

- Překreslování prostřednictvím `SwingWorker` vs přímé překreslování ve vlastním vlákně
- **DoubleBuffer** – přepínání vykresleného obrazu
- Časování a zajištění periody
- Plynulé překreslování bez pohybu myši

```
Toolkit.getDefaultToolkit().sync();
```

`lec08/CanvasDemo`

Simulace vs grafická hra

- V simulaci se zpravidla snažíme důsledně oddělit simulované objekty od vizualizace
  1. Na úrovni simulačních objektů a jejich vizuální reprezentace
  2. Na úrovni simulačního času a rychlosti překreslování

     Přesnost simulace má přednost před rychlou a včasnou vizualizací (v reálném čase)
- Hry jsou zpravidla silně svázány s grafickou vizualizací
  - Krok herního světa zpravidla znamená překreslení
  - Kreslící vlákno tak udává také simulační krok
  - Klíčovým aspektem je zachování plynulosti vizualizace a interakce

    V případě pomalejšího překreslování je rychlost herního světa adekvátně zpomalena.
- Interaktivní hry zpravidla využívají individuálního kreslení objektů do plátna (případně 3D kontextu)

  Používají vlastní sadu komponent (widgets), zpravidla vizuálně efektní, princip je však stejný jako například ve Swing.

Chceme-li maximalizovat využití zdrojů a zajistit vysokou interaktivitu zpravidla musíme mít plně pod kontrolou běh aplikace.

Shrnutí přednášky

Diskutovaná témata

- Modely vícevláknových aplikací
- Paralelní programování a ladění
  - Problém uváznutí a problém souběhu
- Příklady vícevláknových aplikací
  - GUI Využití vláken v GUI
  - GUI plátno – simulátor a kreslení do canvasu

- Příště: Sokety a síťování