

Vícevláknové aplikace

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 5

A0B36PR2 – Programování 2

Část 1 – Paralelní programování

Paralelismus a operační systém
Proces a stavy procesu
Víceprocesorové systémy

Synchronizace výpočetních toků

Část 2 – Vícevláknové aplikace

Vlákna - terminologie, použití

Vícevláknové aplikace v operačním systému

Vlákna v Javě

Část 3 – Modely vícevláknových aplikací

Modely více-vláknových aplikací

Prostředky ladění

Část I

Část 1 – Paralelní programování

Motivace

„Proč se vůbec paralelním programováním zabývat?“

- Navýšení výpočetního výkonu.
Paralelním výpočtem nalezneme řešení rychleji.
- Efektivní využívání strojového času.
Program sice běží, ale čeká na data.
- Zpracování více požadavků najednou.
Například obsluha více klientů v architektuře klient/server.

Základní výpočetní jednotkou je proces – „program“

Paralelní programování

Idea pochází z 60-tých let spolu s prvními multiprogramovými a pseudoparalelními systémy.

- Můžeme rozlišit dva případy paralelismu:
 - hardwarový,
 - softwarový - pseudoparalelismus.

I programy s paralelními konstrukcemi mohou běžet v pseudoparalelním prostředí a to i na víceprocesorovém výpočetním systému.

Proces

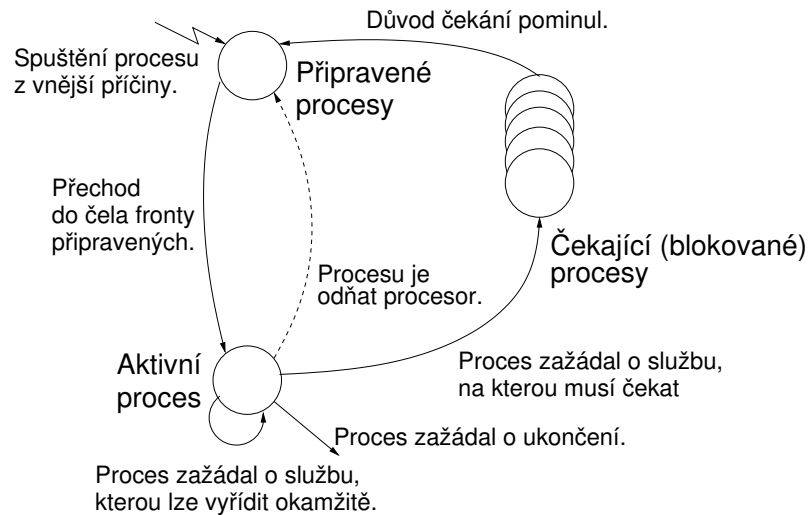
Proces je spuštěný program ve vyhrazeném prostoru paměti. Jedná se o entitu operačního systému, která je plánována pro nezávislé provádění.

Stavy procesu:

- **Executing** - právě běžící na procesoru.
- **Blocked** - čekající na periferie.
- **Waiting** - čekající na procesor.

Proces je identifikován v systému identifikačním číslem PID. Plánovač procesů řídí efektivní přidělování procesoru procesům na základně jejich vnitřního stavu.

Stavy procesu



Víceprocesorové systémy

- Víceprocesorové (jádrové) systémy umožňují skutečný paralelismus.
- Musí být řešena synchronizace procesorů (výpočetních toků) a jejich vzájemná datová komunikace
 - Prostředky k **synchronizaci** aktivit procesorů.
 - Prostředky pro komunikaci mezi procesory.

Příklad výpisu procesů

```

TOP
Last pid: 1666; load averages:  2.34,  1.00,  0.56  up 0+00:21:21  20:37:22
87 processes:  1 running, 86 sleeping
CPU: 97.1% user,  1.4% nice,  1.0% system,  0.6% interrupt,  0.0% idle
Mem: 331M Active, 2720M Inact, 714M Wired, 28M Cache, 404M Buf, 23M Free
ARC: 25M Total, 33K MFU, 24M MRU, 48K Anon, 112K Header, 720K Other
Swap: 2048M Total, 2444K Used, 2045M Free

PID USERNAME  THR PRI  NICE   SIZE   RES STATE  C  TIME  WCPU COMMAND
1612 jf          16  52   0 1058M 21156K uwait   1   2:03 184.42% java
 874 root         1  25   0  919M 45892K select  1   0:16   7.96% Xorg
1569 jf           5  52   5  315M 89640K kqread  0   0:04   1.37% gimp-2.8
1125 jf           4  20   0  216M 16104K uwait   0   0:06   0.29% mocp
1664 root         1  22   0 81508K 8684K  select  1   0:00   0.29% xterm
1666 root         1  21   0 21924K 2584K  CPU1    1   0:00   0.29% top
1023 jf           4  20   0  323M 41148K select  0   0:25   0.20% owncloud
 997 jf           1  20   0  183M 29680K select  1   0:01   0.10% openbox
1095 jf           1  28   0 61508K 7512K  select  1   0:56   0.00% mc
1088 jf           1  25   5 90424K 13896K select  1   0:15   0.00% xpdf
1014 jf           1  21   0  201M 33888K select  1   0:06   0.00% gkrellm
1081 jf           1  20   0  109M 19544K select  0   0:02   0.00% urxvt
1092 jf           1  20   0 23908K 2800K  select  1   0:02   0.00% tmux
1572 jf           1  52   5  193M 33892K select  1   0:01   0.00% script-fu
1319 jf           2  22   0 58900K 11036K select  0   0:01   0.00% vim
 867 root         1  20   0  110M 8312K  wait    0   0:01   0.00% slim
  
```

V současných operačních systémech typicky běží celá řada procesů v pseudoparalelní/paralelním režimu.

Architektury

Řízení vykonávání jednotlivých instrukcí.

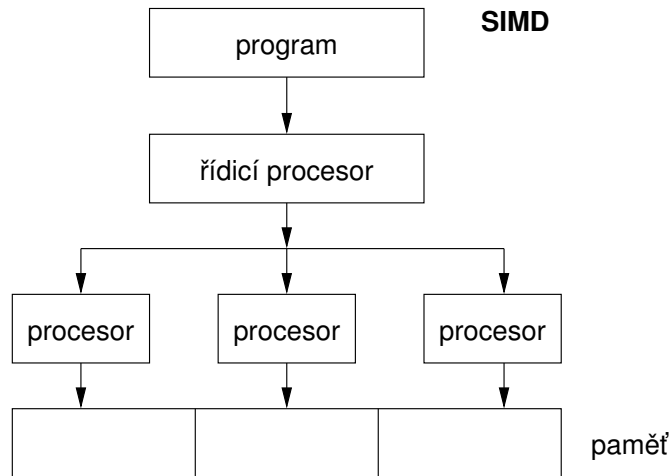
- SIMD (single-instruction, multiple-data) - stejné instrukce jsou vykonávány na více datech. Procesory jsou identické a pracují synchronně. *Příkladem může být vykonávání MMX, SSE, 3dnow! instrukcí, „vektorizace“.*
- MIMD (multiple-instruction, multiple-data) - procesory pracují nezávisle a asynchronně.

Řízení přístupu k paměti.

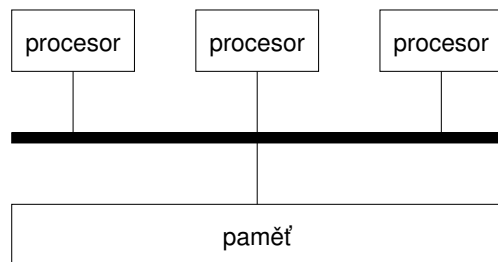
- Systémy se sdílenou pamětí - společná centrální paměť.
- Systémy s distribuovanou pamětí - každý procesor má svou paměť.

Informativní

SIMD

*Informativní*

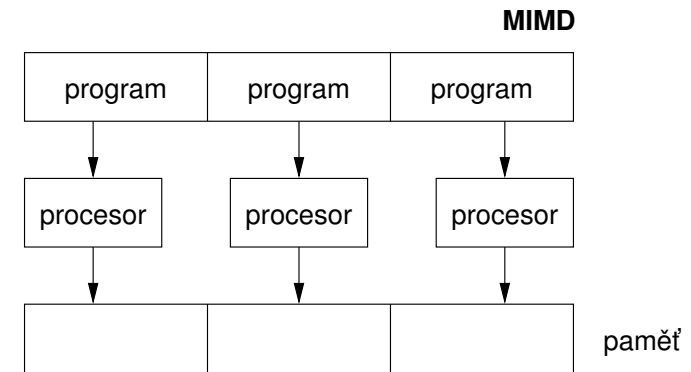
Systémy se sdílenou pamětí



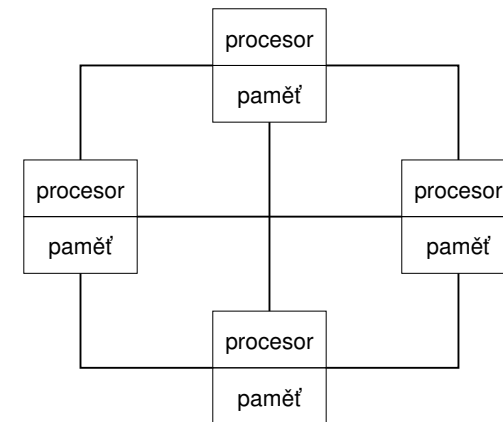
- Procesory komunikují prostřednictvím sdíleného paměťového prostoru.
- Mohou tak také synchronizovat své aktivity → problém exkluzivního přístupu do paměti.

Informativní

MIMD

*Informativní*

Systémy s distribuovanou pamětí



Není problém s exkluzivitou přístupu do paměti, naopak je nutné řešit komunikační problém přímými komunikačními kanály mezi procesory.

Informativní

Úloha operačního systému

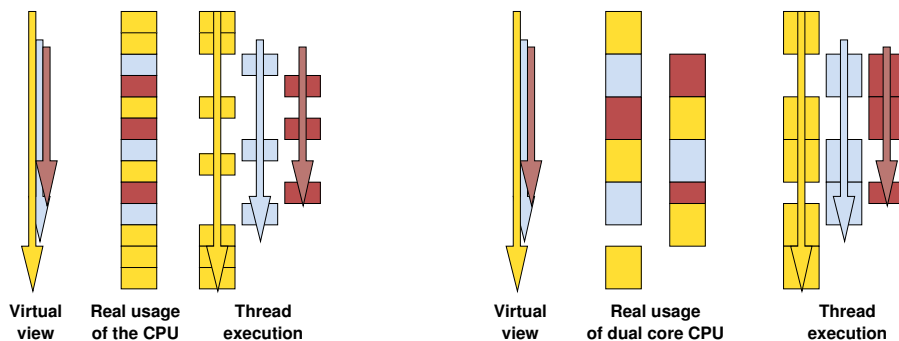
- Operační systém integruje a synchronizuje práci procesorů, odděluje uživatele od fyzické architektury.
- Operační systém poskytuje:
 - Prostředky pro tvorbu a rušení procesů.
 - Prostředky pro správu více procesorů a procesů, rozvrhování procesů na procesory.
 - Systém sdílené paměti s mechanismem řízení.
 - Mechanismy mezi-procesní komunikace.
 - Mechanismy synchronizace procesů.
- V rámci spuštěného Java programu plní virtuální stroj **JVM** spolu se základními knihovnami JDK roli operačního systému

Zapouzdřuje přístup k hw (službám OS)
- To co platí pro procesy na úrovni OS platí analogicky pro samostatné výpočetní toky v rámci **JVM**

V Javě se jedná o vlákna

Synchronizace výpočetních toků

- Klíčovým problémem paralelního programování je jak zajistit efektivní sdílení prostředků a zabránit kolizi
- Je nutné řešení problémů vzniklých z možného paralelního běhu bez ohledu na to zdali se jedná o skutečně paralelní nebo pseudo-paralelní prostředí



Paralelní zpracování a programovací jazyky

- Z pohledu paralelního zpracování lze programovací jazyky rozdělit na dvě skupiny
 1. Jazyky bez explicitní podpory paralelismu
 - Paralelní zpracování ponechat na překladači a operačním systému

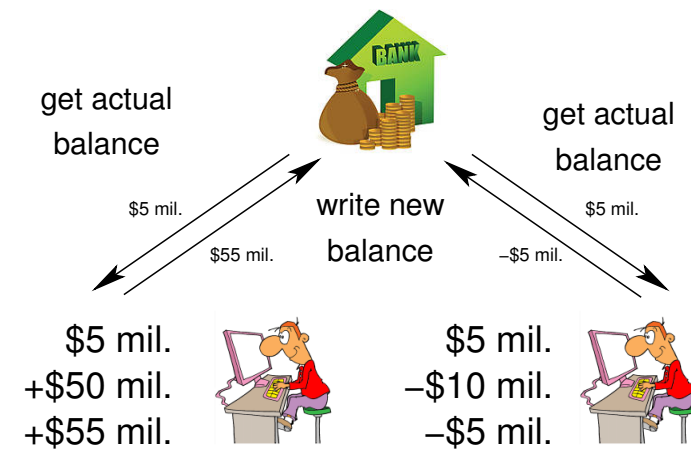
Např. automatická „vektizace“
 - Paralelní konstrukce explicitně označit pro kompilátor.

Např. OpenMP
 - Využití služeb operačního systému pro paralelní zpracování.
 2. Jazyky s explicitní podporou paralelismu
 - Nabízejí výrazové prostředky pro vznik nového procesu (výpočetního toku)

Granularita procesů - od paralelismu na úrovni instrukcí až po paralelismus na úrovni programů.

Problém souběhu – příklad

- Současná aktualizace zůstatku na účtě může vést bez exkluzivního přístupu k různým výsledkům



Je nutné zajistit alokování zdrojů a exkluzivní (synchronizovaný) přístup jednotlivých procesů ke sdílenému prostředku (bankovnímu účtu).

Semafor

- Základní prostředkem pro synchronizaci v modelu se sdílenou pamětí je **Semafor** *E. W. Dijkstra*
- Semafor je proměnná typu integer, přístupná operacemi:
 - *InitSem* - inicializace.
 - *Wait* - $\begin{cases} S > 0 - S = S - 1 \\ \text{jinak} - \text{pozastavuje činnost volajícího procesu.} \end{cases}$
 - *Signal* - $\begin{cases} \text{probudí nějaký čekající proces pokud existuje} \\ \text{jinak} - S = S + 1. \end{cases}$
- Semafor se používají pro přístup ke sdíleným zdrojům.
 - $S < 0$ - sdílený prostředek je používán. Proces žádá o přístup a čeká na uvolnění.
 - $S > 0$ - sdílený prostředek je volný. Proces uvolňuje prostředek.

Použití semaforů

- Ošetření kritické sekce, tj. části programu vyžadující výhradní přístup ke sdílené paměti (prostředku).

Příklad ošetření kritické sekce semafor

```
InitSem(S,1);
Wait(S);
/* Kód kritické sekce */
Signal(S);
```

- Synchronizace procesů semafor.

Příklad synchronizace procesů

```
/* process p */
InitSem(S,0)
Wait(S); ...
exit();

/* process q */
Signal(S); exit();
```

Proces p čeká na ukončení procesu q.

Informativní

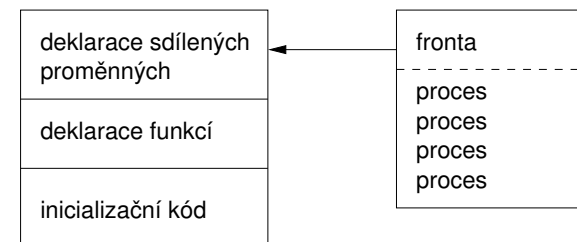
Implementace semaforů

- Práce se semaforem musí být atomická, procesor nemůže být přerušen.
- Strojová instrukce *TestAndSet* přečte a zapamatuje obsah adresované paměťové lokace a nastaví tuto lokaci na nenulovou hodnotu.
- Během provádění instrukce *TestAndSet* drží procesor sběrnici a přístup do paměti tak není povolen jinému procesoru.

Informativní

Monitory

- Monitor - jazyková konstrukce zapouzdřující data a operace nad daty s exkluzivním přístupem.
- Přístup k funkcím v monitoru má v daném okamžiku pouze jediný proces.



- Přístup k monitoru je realizován podmínkovými proměnnými. Ke každé proměnné existuje fronta čekajících procesů.

V Javě je synchronizace řešena právě mechanismem monitorů – jako monitor může výstupovat libovolný objekt

Část II

Část 2 – Vícevláknové aplikace

Kdy vlákna použít?

„Vlákna jsou lehčí variantou procesů, navíc sdílejí paměťový prostor.“

- Efektivnější využití zdrojů.

Příklad

Čeká-li proces na přístup ke zdroji, předává řízení jinému procesu. Čeká-li vlákno procesu na přístup ke zdroji, může jiné vlákno téhož procesu využít časového kvanta přidělené procesu.

- Reakce na asynchronní události.

Příklad

Během čekání na externí událost (v blokováném režimu), může proces využít CPU v jiném vlákně.

Co jsou vlákna?

- Vlákno - Thread.
- Vlákno je **samostatně** prováděný **výpočetní tok**.
- Vlákna běží v rámci procesu.
- Vlákna jednoho procesu běží v rámci stejného prostoru paměti.
- Každé vlákno má vyhrazený prostor pro specifické proměnné (*runtime prostředí*).

Příklady použití vláken

- **Vstupně výstupní operace.**

Příklad

Vstupně výstupní operace mohou trvat relativně dlouhou dobu, která většinou znamená nějaký druh čekání. Během komunikace, lze využít přidělený procesor na výpočetně náročné operace.

- **Interakce grafického rozhraní.**

Příklad

Grafické rozhraní vyžaduje okamžité reakce pro příjemnou interakci uživatele s naší aplikací. Interakce generují události, které ovlivňují běh aplikace. Výpočetně náročné úlohy, nesmí způsobit snížení interakce rozhraní s uživatelem.

Vlákna a procesy

Procesy

- Výpočetní tok.
- Běží ve vlastním paměťovém prostoru.
- Entita OS.
- Synchronizace entitami OS (IPC).
- Přidělení CPU, rozvrhovačem OS.
- Časová náročnost vytvoření procesu.

Vlákna procesu

- Výpočetní tok.
- Běží ve společném paměťovém prostoru.
- Uživatelská nebo OS entita.
- Synchronizace exkluzivním přístupem k proměnným.
- Přidělení CPU, v rámci časového kvanta procesu.
- + Vytvoření vlákna je méně časově náročné.

Vlákna v operačním systému

- Vlákna běží v rámci výpočetního toku procesu.
- S ohledem na realizaci se mohou nacházet:
 - **V uživatelském prostoru procesu.** Realizace vláken je na úrovni knihovních funkcí. Vlákna nevyžadují zvláštní podporu OS, jsou rozvrhována uživatelským knihovním rozvrhovačem. Nevyužívají více procesorů.
 - **V prostoru jádra OS.** Tvoří entitu OS a jsou také rozvrhována systémovým rozvrhovačem. Mohou paralelně běžet na více procesorech.

Vícevláknové a víceprocesové aplikace

Vícevláknová aplikace má oproti více procesové aplikaci výhody:

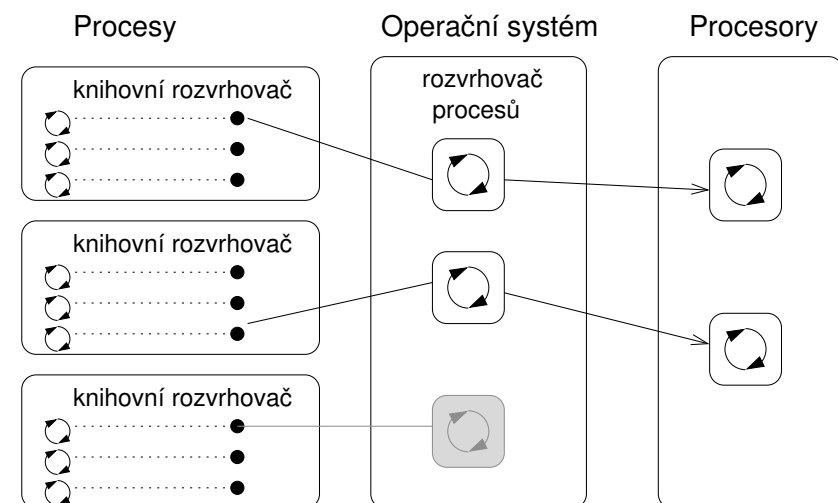
- Aplikace je mnohem interaktivnější.
- Snadnější a rychlejší komunikace mezi vlákny (stejný paměťový prostor).

Nevýhody:

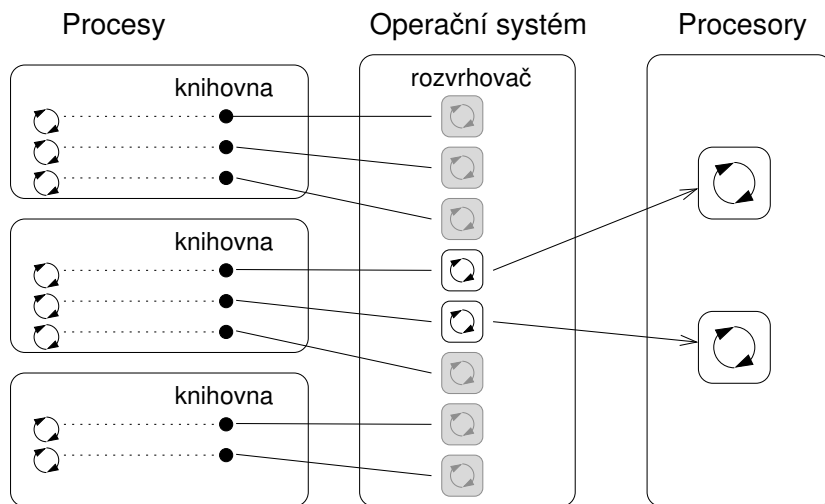
- Distribuce výpočetních vláken na různé výpočetní systémy (počítače).

Na jednoprocessorových systémech vícevláknové aplikace lépe využívají CPU.

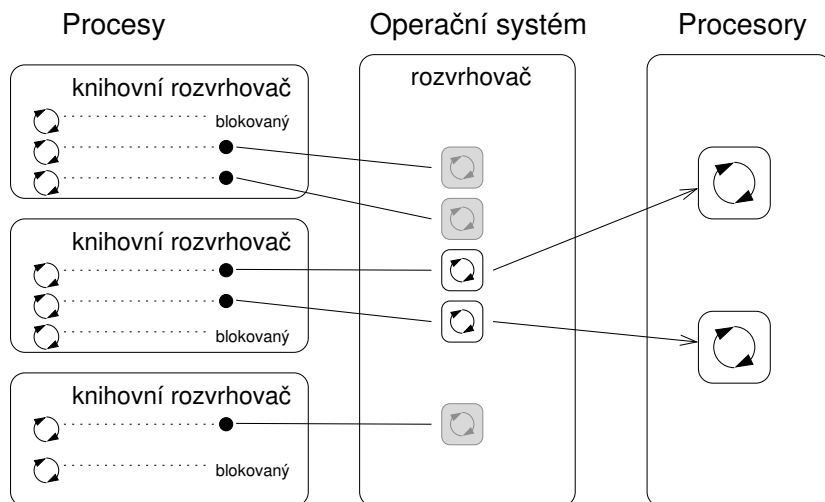
Vlákna v uživatelském prostoru



Vlákna v prostoru jádra operačního systému



Kombinace uživatelského a jaderného prostoru



Uživatelský vs jaderný prostor vláken

Uživatelský prostor

- + Není potřeba podpory OS.
- + Vytvoření nepotřebuje náročné systémové volání.
- Priority vláken se uplatňují pouze v rámci přiděleného časového kvanta procesu.
- Nemohou běžet paralelně.

Vyšší počet vláken, které jsou rozvrhována OS mohou zvyšovat režii. Moderní operační systémy implementují „ $O(1)$ rozvrhovače”.

Prostor jádra

- + Vlákna jsou rozvrhována kompetitivně v rámci všech vláken v systému.
- + Vlákna mohou běžet paralelně.
- Vytvoření vláken je časově náročnější.

Vlákna v Javě

- Objekt třídy odvozené od třídy **Thread**
- Tělo nezávislého výpočetního toku vlákna definujeme v metodě **public void run()**

Overriding
- Metodu **run** nespouštíme přímo!
- Pro spuštění vlákna slouží metoda **start()**, která zajistí vytvoření vlákna a jeho rozvrhování
- Vlákno můžeme pojmenovat předáním jména nadřazené třídy v konstruktoru

<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

Příklad vlákna

```
public class Worker extends Thread {
    private final int numberOfJobs;

    public Worker(int id, int jobs) {
        super("Worker " + id);
        myID = id;
        numberOfJobs = jobs;
        stop = false;
        System.out.println("Worker id: " + id + " has
            been created threadID:" + getId());
    }

    public void run() {
        doWork();
    }
}
```

Vytvoření vlákna implementací rozhraní **Runnable** 1/2

- V případě, že nelze použít dědění od `Thread`, implementujeme rozhraní `Runnable` předepisující metodu `run()`

```
public class WorkerRunnable implements Runnable {
    private final int id;
    private final int numberOfJobs;

    public WorkerRunnable(int id, int jobs) {
        this.id = id;
        numberOfJobs = jobs;
    }
    public String getName() {
        return "WorkerRunnable " + id;
    }
    @Override
    public void run() { ... }
}
```

Příklad vytvoření a spuštění vlákna

- Vlákno vytvoříme novou instancí třídy `Worker`
- Spuštění vlákna provedeme metodou `start()`

```
Worker thread = new Worker(1, 10);
thread.start(); //new thread is created
System.out.println("Program continues here");
```

- Po spuštění vlákna pokračuje program ve vykonávání další instrukce.
- Tělo metody `run()` vlákna `thread` běží v samostatném vlákně.

Vytvoření vlákna implementací rozhraní **Runnable** 2/2

- Vytvoření vlákna a spuštění je přes instanci třídy `Thread`

```
WorkerRunnable worker = new WorkerRunnable(1, 10);
Thread thread = new Thread(worker, worker.getName());
thread.start();
```

- Aktuální výpočetní tok (vlákno) lze zjistit voláním `Thread.currentThread()`

```
public void run() {
    Thread thread = Thread.currentThread();
    for (int i = 0; i < numberOfJobs; ++i) {
        System.out.println("Thread name: " + thread.
            getName());
    }
}
```

lec05/WorkerRunnable

Vlákna v Javě – metody třídy Thread

- `String getName()` – jméno vlákna
- `boolean isAlive()` – test zdali vlákno běží
- `void join()` – pozastaví volající vlákno dokud příslušné vlákno není ukončeno
- `static void sleep()` – pozastaví vlákno na určenou dobu
- `int getPriority()` – priorita vlákna
- `static void yield()` – vynutí předání řízení jinému vláknu

Příklad čekání na ukončení činnosti vlákna – 2/2

- Nastavíme vlákna před spuštěním


```
for (Thread thread : threads) {
    thread.setDaemon(true);
    thread.start();
}
```

V tomto případě se aplikace ihned ukončí.
- Pro čekání na ukončení vláken můžeme explicitně použít metodu `join()`.


```
try {
    for (Thread thread : threads) {
        thread.join();
    }
} catch (InterruptedException e) {
    System.out.println("Waiting for the thread ...");
}
```

`lec05/DemoThreads`

Příklad čekání na ukončení činnosti vlákna – 1/2

- Vytvoříme třídu `DemoThreads`, která spustí „výpočet“ v `numberOfThreads` paralelně běžících vláknech


```
ArrayList<Worker> threads = new ArrayList();
for (int i = 0; i < numberOfThreads; ++i) {
    threads.add(new Worker(i, 10));
}
// start threads
for (Thread thread : threads) {
    thread.start();
}
```
- Po skončení hlavního vlákna program (JVM) automaticky čeká až jsou ukončena všechna vlákna
- Tomu můžeme zabránit nastavením vlákna do tzv. `Daemon` režimu voláním `setDaemon(true)`

Ukončení činnosti vlákna

- Činnost vlákna můžeme ukončit „zasláním (vlastní) zprávy“ výpočetnímu toku s „žádostí“ o přerušení činnosti

V zásadě jediný korektní způsobem
- Ve vláknech **musíme** implementovat mechanismu detekce žádosti o přerušení činnosti, např. nastavení příznakové proměnné `stop` a rozdělením výpočtu na menší části


```
public class Worker extends Thread {
    ...
    private boolean stop;

    public Worker(int id, int jobs) {
        ...
        stop = false;
    }

    public void run() {
        for (int i = 0; i < numberOfJobs; ++i) {
            if (stop) {
                break;
            }
            doWork();
        }
    }
}
```

Přístup ke „sdílené proměnné“ z více vláken

- Žádost o ukončení implementujeme v metodě **shutdown**, kde nastavíme proměnnou **stop**

```
public void shutdown() {
    stop = true;
}
```

- Přístup k základní proměnné je atomický a souběh tak „netřeba“ řešit
- Překladač a virtuální stroj (JVM) musíme informovat, že se hodnota proměnné může nezávisle měnit použitím klíčového slova **volatile**

<http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html>

<http://www.root.cz/clanky/>

[pohled-pod-kapotu-jvm-zaklady-optimalizace-aplikaci-naprogramovanych-v-jave-4/](#)

- Například:

```
private volatile boolean stop;
```

Synchronizace činnosti vláken – **monitor**

- V případě spolupracujících vláken je nutné řešit problém sdílení datového prostoru
 - Řešení problému souběhu – tj. problém současného přístupu na datové položky z různých vláken
- Řešením je využít kritické sekce – **monitor**
 - Objekt, který vládku „zprístupní“ sdílený zdroj
 - Můžeme si představit jako zámek.*
 - V daném okamžiku aktivně umožní monitor používat jen jedno vlákno
 - Pro daný časový interval vlákno vlastní příslušný monitor – monitor smí „vlastnit“ vždy jen jedno vlákno
 - Vlákno běží, jen když vlastní příslušný monitor, jinak čeká
- V Javě mohou mít všechny objekty svůj monitor
- Libovolný objekt tak můžeme použít pro definici **kritické sekce**

Příklad – Odložené ukončení vláken

- Příklad s vlákny **DemoThreads** rozšíříme o explicitní ukončení vláken po definované době
- Vytvoříme třídu **ThreadKiller**, která ukončí vlákna po **timeout** sekundách

```
public class ThreadKiller implements Runnable {
    ArrayList<Worker> threads;
    int timeout;
    public ThreadKiller(ArrayList<Worker> threads, int time) {
        ...
    }
    @Override
    public void run() {
        try {
            Thread.sleep(timeout * 1000);
            System.out.println("ThreadKiller ...");
            for (Worker thread : threads) {
                thread.shutdown();
            }
            for (Worker thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) { ... }
    }
}
```

lec05/ThreadKiller

Kritická sekce – **synchronized**

- Kritickou sekci deklarujeme příkazem **synchronized** s jediným argumentem objektu (referenční proměnné) definující příslušným monitor

```
Object monitor = new Object();
synchronized(monitor) {
    //Critical section protected
    //by the monitor
}
```

- Vstup do kritické sekce je umožněn pouze jedinému vládku
 - Vlákno, které první vstoupí do kritické sekce může používat zdroje „chráněné“ daným monitorem
 - Ostatní vlákna čekají, dokud aktivní vlákno neopustí kritickou sekci a tím uvolní zámek

*Případně zavolat **wait***

Synchronizované metody

- Metody třídy můžeme deklarovat jako synchronizované, např.

```
class MyObject {
    public synchronized void useResources() {
        ...
    }
}
```

- Přístup k nim je pak chráněn monitorem objektu příslušné instance třídy (**this**), což odpovídá definování kritické sekce

```
public void useResources() {
    synchronized(this) {
        ...
    }
}
```

Deklarací metody jako synchronizované informujeme uživatele, že metoda je synchronizovaná bez nutnosti čtení zdrojového kódu.

Část III

Část 3 – Modely vícevláknových aplikací

Komunikace mezi vlákny

- Vlákna jsou objekty a ty si mohou zasílat zprávy (volání metod)
- Každý objekt (monitor) navíc implementuje metody pro explicitní ovládání a komunikaci mezi vlákny:
 - wait** – dočasně pozastaví vlákno do doby než je probuzeno metodou **notify** nebo **notifyAll**, nebo po určené době
Uvolňuje příslušný zablokovaný monitor
 - notify** – probouzí pozastavené vlákno metodou **wait()**, čeká-li více vláken není určeno, které vlákno převezme monitor
 - notifyAll** – probouzí všechna vlákna pozastavena metodou **wait()**

Monitoru se zmocní vlákno s nejvyšší prioritou

Kdy použít vlákna?

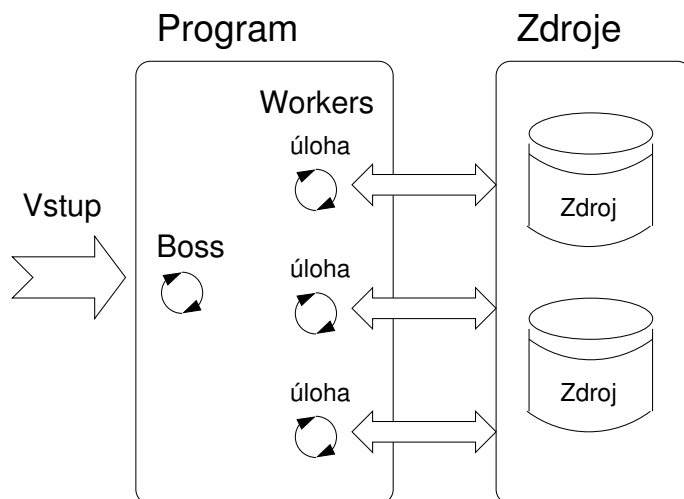
Vlákna je výhodné použít, pokud aplikace splňuje některé následující kritérium:

- 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 úlohu na více procesorů.
- **Aplikace reálného času** - lze využít specifických rozvrhovačů. Vícevláknová aplikace je výkonnější než složitě asynchronní programování, neboť vlákno čeká na příslušnou událost namísto explicitního přerušování vykonávání kódu a přepínání kontextu.

Boss/Worker model



Modely vícevláknových aplikací

Modely řeší způsob vytváření a rozdělování práce mezi vlákna.

- **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 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       {
6       case taskX :
7         create_thread(taskX
8         );
9       case taskY :
10        create_thread(taskY
11        );
12      }
13    }
14 }

```

```

1 //Worker
2 taskX() {
3   řešení úlohy,
4   synchronizace v
5   případě sdílených
6   zdrojů;
7   done;
8 }
9
10 taskY() {
11  řešení úlohy,
12  synchronizace v
13  případě sdílených
14  zdrojů;
15  done;
16 }

```

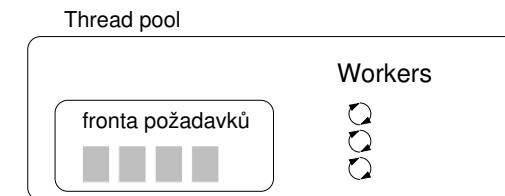
Thread Pool - vlastnosti

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

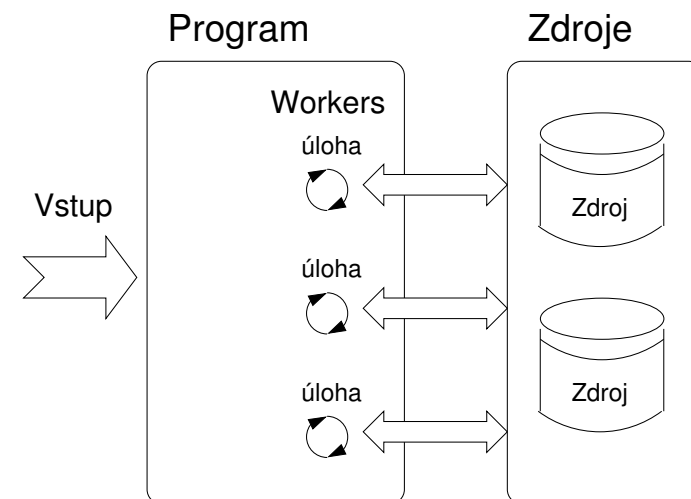
Například blokování příchozích požadavků.

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.



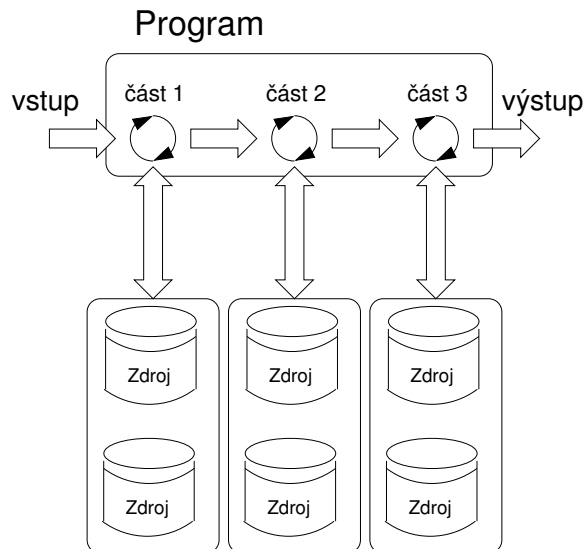
Peer model



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.

Zpracování proudu dat - Pipeline



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     čekání na spuštění;
4     řešení úlohy,
5     synchronizace v
6     případě sdílených
7     zdrojů;
8     done;
9 }
10 task2() {
11     čekání na spuštění;
12     řešení úlohy,
13     synchronizace v
14     případě sdílených
15     zdrojů;
16     done;
17 }

```

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      ;
4      create_thread(stage2)
5      ;
6      ;
7      wait for all pipeline
8      ;
9  }
10 stage1() {
11     while(input) {
12         get next program
13         input;
14         process input;
15         pass result to next
16         stage;
17     }
18 }
19
20 stage2() {
21     while(input) {
22         get next input from
23         thread;
24         process input;
25         pass result to next
26         stage;
27     }
28 }
29
30 stageN() {
31     while(input) {
32         get next input from
33         thread;
34         process input;
35         pass result to
36         output;
37     }
38 }

```

Označení kritické sekce – Mutex

- Mutex představuje „zámek“ kritické sekce – analogie **synchronized** monitoru
- Jedná se vlastně o semafor s hodnotou 1 nebo 0
- Základní operace:
 - **Lock** - uzamknutí mutexu (přiřazení mutexu vláknu). Pokud nelze mutex získat, vlákno přechází do blokováného režimu a čeká na uvolnění zámku.
 - **Unlock** - uvolnění zámku. Pokud jiná vlákna čekají na uvolnění, je vybráno jedno vlákno, které mutex získá.
- **Rozšířené modely:**
 - Rekursivní - vícenásobné zamykání stejným vláknem.
 - Try - okamžitý návrat pokud není možné mutex získat.
 - Timed - pokus o získání zámku s omezenou dobou čekání.

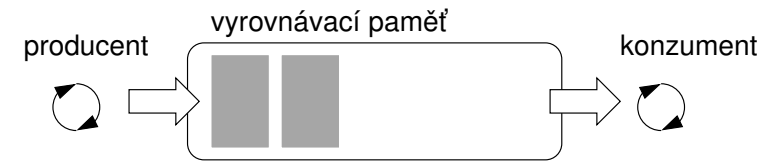
Informativní

Producent a konzument

Předávání dat mezi vlákny je realizováno vyrovnávací pamětí bufferem.

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

Přístup do vyrovnávací paměti 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 simultánně.
- **thread-safe** - Funkce je možné simultánně 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.

Podpora ladění

Debugger:

- Výpis běžících vláken.
- Výpis stavu synchronizačních primitiv.
- Přístup k proměnným vláken.
- Pozastavení běhu konkrétního vlákna.

Logování:

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

Prostředky ladění

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

Poznámky - „problémy souběhu“

Problémy souběhu jsou typicky způsobeny nedostatkem synchronizace.

- **Vlákna jsou asynchronní.** *Nespoléhat na to, že na jednoprocetovém 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 nespolehejte na to, že vlákno po vytvoření počká, může být spuštěno velmi brzy.**
- **Pokud nspecifikujete 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 problémem je tzv. *mutex deadlock* způsobený pořadím získávání mutexů.
- 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.

Vlákna v GUI (Swing)

- Vlákna můžeme 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>

Část IV

Část 4– Využití vláken v GUI

Samostatné výpočetní vlákno pro model

- Třidu `Model` rozšíříme o rozhraní `Runnable`
- Vytvoříme novou třídu `ThreadModel`
 - Voláním metody `compute` spustíme samostatné vlákno
 - Musíme zabránit opakovanému vytváření vlákna
 - Príznak `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`*
 - `lec05/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
- Pozorovatelé jsou předplatiteli (*subscribers*) 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 2/4

- Musíme zajistit rozhraní pro přihlašování a odhlašování pozorovatelů
 - Zároveň nechceme měnit typ výpočetní model 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);
}
```

lec05/DemoBarComp-observer

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

- **Observable** je abstraktní třídy, proto vytvoříme nový **Observable** objekt jako instanci privátní třídy `UpdateNotificator`
- `UpdateNotificator` slouží 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 registered
            observers
        }
    }
    UpdateNotificator updateNotificator;

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

lec05/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();
    }
    private void updateProgress() {
        if (computation != null) {
            bar.setValue(computation.getProgress());
        }
    }
}
```

lec05/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.

lec05/DemoBarComp-observer

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 modelu `ThreadModel`

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

lec05/DemoBarComp-swingworker

Výpočetní vlákno ve Swing

- Alternativně můžeme využít třídu `SwingWorker`
- Ta definuje metodu `doInBackground()`, která zapouzdřuje výpočet na „pozadí“ v samostatném vlákně
 - 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** 2/3

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

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

    MySwingWorker worker;
}
```

- Tlačítko `Compute` připojíme 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();
        }
    }
}
```

lec05/DemoBarComp-swingworker

Příklad použití třídy `SwingWorker` 3/3

- Ve `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();
    }
}
```

lec05/DemoBarComp-swingworker

- S výhodou využíváme přímého přístupu k `updateProgress`

Shrnutí přednášky

Diskutovaná témata

- Paralelní programování
 - Procesy a role operačního systému
 - Vlákna v operačním systému
 - Problém souběhu, synchronizace vláken a monitor
- Vlákna v Javě
 - Vytvoření, synchronizace a komunikace mezi vlákny
- Modely vícevláknových aplikací
- Paralelní programování a ladění
 - Problém uváznutí a problém souběhu
- Příklady vláken v GUI (Swing)
 - Návrhový vzor `Observer`
 - `SwingWorker`
- **Příště: dokončení a ukázky aplikací**