

# Vícevláknové aplikace

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 5

A0B36PR2 – Programování 2



# Část 1 – Paralelní programování

## Parallelismus a operační systém

Výpočetní 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 – Využití vláken v GUI

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker



# Část I

## Část 1 – Paralelní programování



# Obsah

## Paralelismus a operační systém

Výpočetní proces a stavy procesu

Víceprocesorové systémy

## Synchronizace výpočetních toků



# 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.



# 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“*



# 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“*



# Výpočetní proces

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

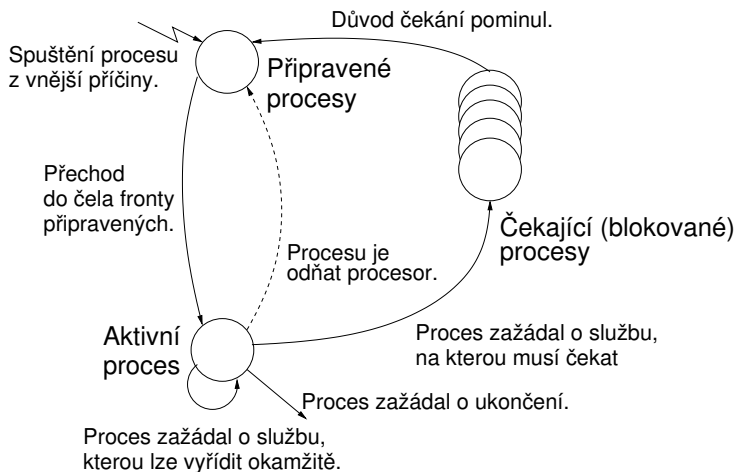
Stavy procesu:

- **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



# 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.*



# 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.



# 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.

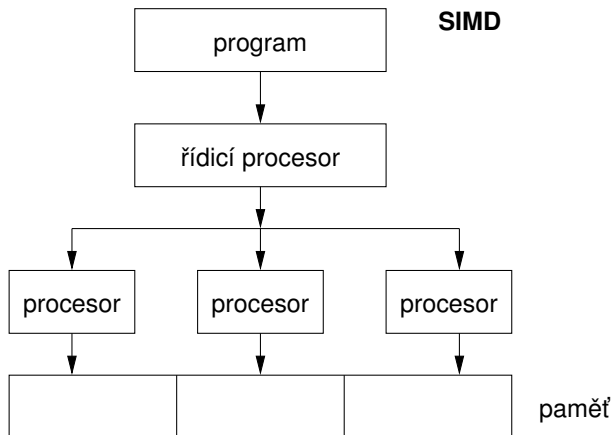
Řízení přístupu k paměti.

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

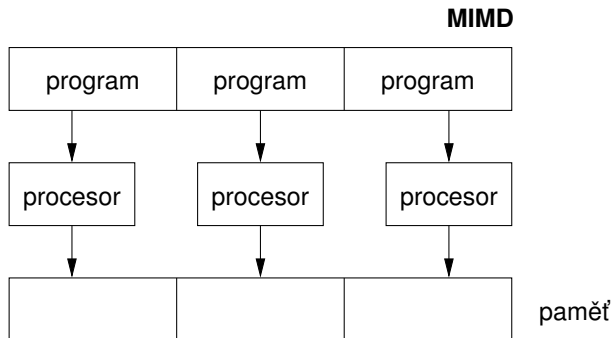
*Informativní*



# SIMD



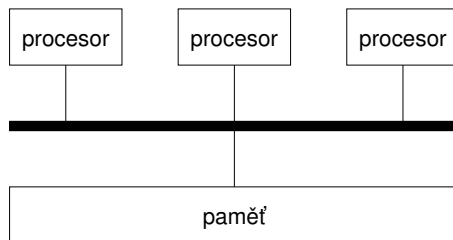
# MIMD



*Informativní*



## Systémy se sdílenou pamětí

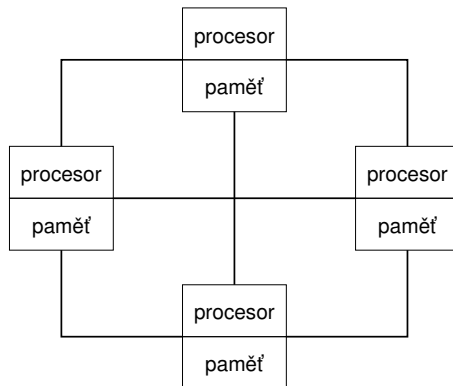


- Procesory komunikují prostřednictvím sdíleného paměťového prostoru.
- Mohou tak také synchronizovat své aktivity → problém exkluzivního přístupu do paměti.

*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*



# 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*



# 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*



# 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 „vektorizace”*
    - 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.



# Obsah

## Paralelismus a operační systém

Výpočetní proces a stavy procesu

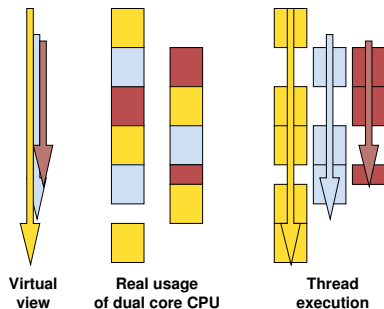
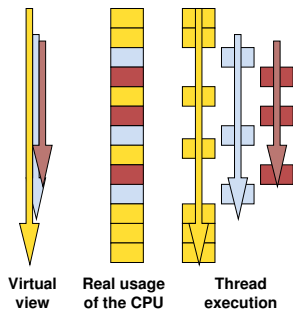
Víceprocesorové systémy

## Synchronizace výpočetních toků



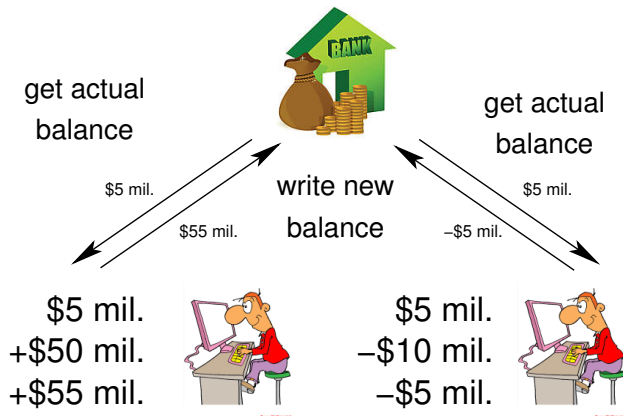
# Synchronizace výpočetních toků

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



## Problém souběhu – příklad

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

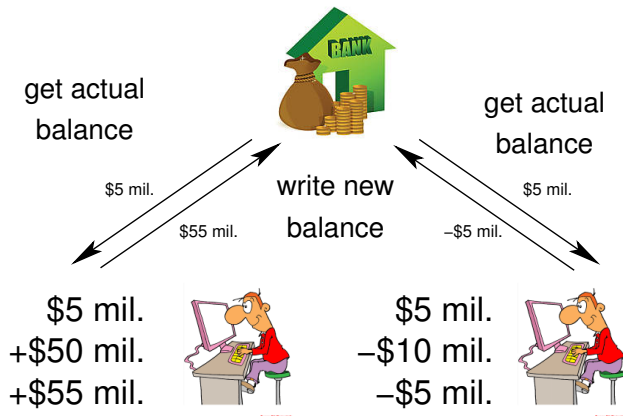


Je nutné zajistit alokování zdrojů a exkluzivní (synchronizovaný) přístup jednotlivých procesů ke sdílenému prostředku (bankovnímu účtu).



## 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).



# Semaforey

- Základním 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* -  $\begin{cases} \text{probudí nějaký čekající proces pokud existuje} \\ \text{jinak} - S = S + 1. \end{cases}$
- Semaforey 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.



# 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*



## Použití semaforů

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

### Příklad ošetření kritické sekce semaforu

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

- Synchronizace procesů semaforu.

### Příklad synchronizace procesů

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

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

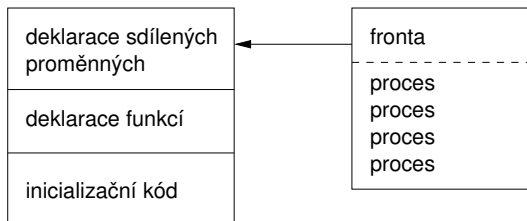
Proces p čeká na ukončení procesu q.

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 vystupovat libovolný objekt



# Část II

## Část 2 – Vícevláknové aplikace



# Obsah

Vlákna - terminologie, použití

Vícevláknové aplikace v operačním systému

Vlákna v Javě



# 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*).



## 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.



## 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.



## 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.



## 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.



# Obsah

Vlákna - terminologie, použití

Vícevláknové aplikace v operačním systému

Vlákna v Javě



# 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.



## 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).

I na jednoprocessorových systémech vícevláknové aplikace lépe využívají CPU.



# Příklad výpisu procesů a jim příslušejících vláken

```

TOP
last pid: 1667; load averages: 2.52, 1.13, 0.62 up 0+00:21:43 20:37:44
135 processes: 5 running, 130 sleeping
CPU: 99.0% user, 0.2% nice, 0.6% system, 0.2% interrupt, 0.0% idle
Mem: 340M Active, 2720M Inact, 717M Wired, 12M Cache, 407M Buf, 27M Free
ARC: 25M Total, 33K MFU, 24M MRU, 48K Anon, 112K Header, 720K Other
Swap: 2048M Total, 2444K Used, 2045M Free

```

| PID  | USERNAME | PRI | NICE | SIZE   | RES    | STATE  | C | TIME | WCPU   | COMMAND        |
|------|----------|-----|------|--------|--------|--------|---|------|--------|----------------|
| 1612 | jf       | 92  | 0    | 1058M  | 21156K | CPU0   | 0 | 0:55 | 71.29% | java{java}     |
| 1612 | jf       | 92  | 0    | 1058M  | 21156K | RUN    | 1 | 0:55 | 62.60% | java{java}     |
| 1612 | jf       | 91  | 0    | 1058M  | 21156K | RUN    | 1 | 0:55 | 62.06% | java{java}     |
| 1569 | jf       | 27  | 5    | 323M   | 97100K | select | 1 | 0:05 | 2.88%  | gimp-2.8{gimp- |
| 874  | root     | 21  | 0    | 919M   | 45916K | select | 0 | 0:17 | 1.95%  | Xorg           |
| 1023 | jf       | 21  | 0    | 323M   | 41204K | select | 1 | 0:19 | 1.27%  | owncloud{owncl |
| 1125 | jf       | 20  | 0    | 216M   | 16124K | uwait  | 0 | 0:05 | 0.29%  | mocp{mocp}     |
| 1095 | jf       | 28  | 0    | 61508K | 7512K  | select | 1 | 0:56 | 0.00%  | mc             |
| 1088 | jf       | 25  | 5    | 90424K | 13896K | select | 0 | 0:15 | 0.00%  | xpdf           |
| 1014 | jf       | 20  | 0    | 201M   | 33888K | select | 0 | 0:06 | 0.00%  | gkrellm{gkrell |
| 1081 | jf       | 20  | 0    | 109M   | 19544K | select | 1 | 0:02 | 0.00%  | urxvt          |
| 1092 | jf       | 20  | 0    | 23908K | 2800K  | select | 0 | 0:02 | 0.00%  | tmux           |
| 997  | jf       | 20  | 0    | 183M   | 29684K | select | 1 | 0:01 | 0.00%  | openbox        |
| 1023 | jf       | 20  | 0    | 323M   | 41204K | select | 1 | 0:01 | 0.00%  | owncloud{owncl |
| 1023 | jf       | 21  | 0    | 323M   | 41204K | kqread | 0 | 0:01 | 0.00%  | owncloud{owncl |
| 1572 | jf       | 52  | 5    | 193M   | 33892K | select | 1 | 0:01 | 0.00%  | script-fu      |

*Jeden proces může být rozdělen na více vláken, která jsou v tomto případě rozvrhována operačním systémem na dostupné procesory.*



# 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.



# 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.

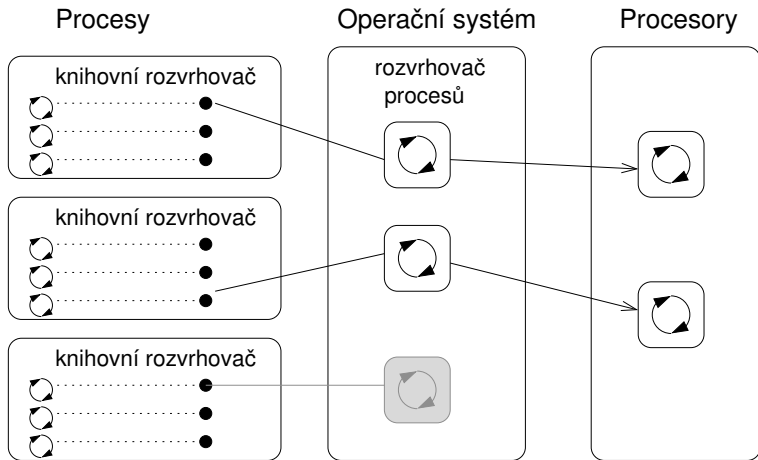


# Vlákna v operačním systému

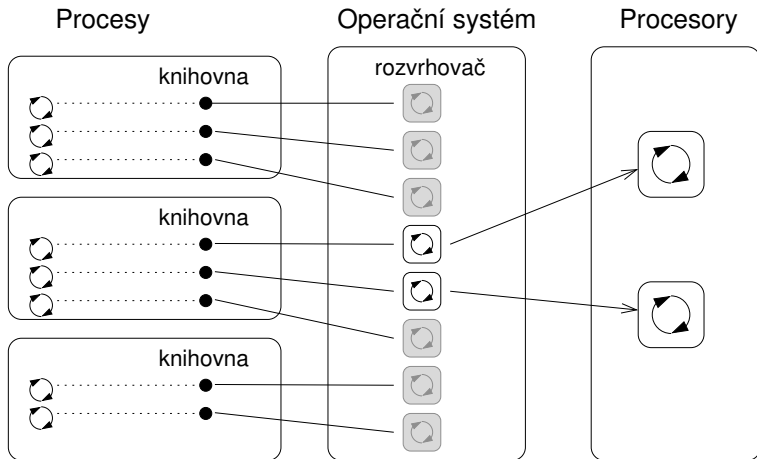
- Vlákna běží v rámci výpočetního toku procesu.
- S ohledem na realizaci se mohou nacházet:
  - V uživatelském prostoru procesu. Realizace vláken je na úrovni knihovních funkcí. Vlákna nevyžadují zvláštní podporu OS, jsou rozvrhována uživatelským knihovním rozvrhovačem. Nevyužívají více procesorů.
 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



# Vlákna v uživatelském prostoru



# Vlákna v prostoru jádra operačního systému



# 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.

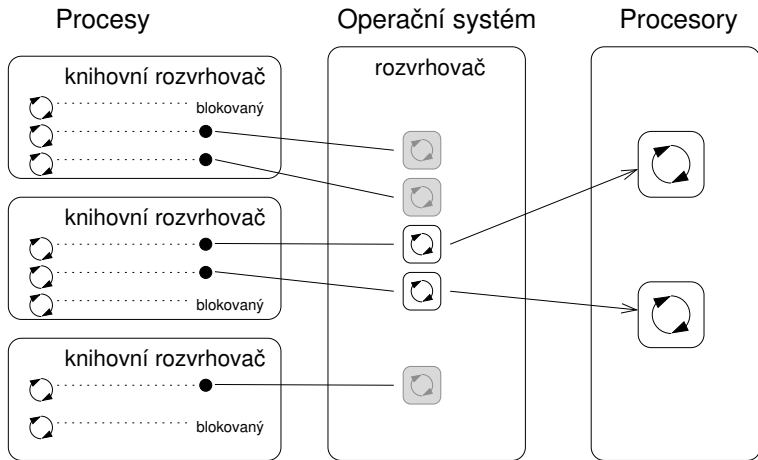
Vyšší počet vláken, která jsou rozvrhována OS mohou zvyšovat režii.  
Moderní operační systémy implementují „ $O(1)$  rozvrhovače”.

## Prostor jádra

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



# Kombinace uživatelského a jaderného prostoru



# Obsah

Vlákna - terminologie, použití

Vícevláknové aplikace v operačním systému

Vlákna v Javě



# 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 třídě 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();  
    }  
}
```



## 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()` objektu `thread` běží v samostatném vlákně.



## 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() { ... }  
}
```



## 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 – 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)`



## 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 – 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



## Ukončení činnosti vlákna

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

*V zásadě jediný korektní způsob!*

- Ve vlákne **musíme** implementovat mechanismus 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 ve více vláknem —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;
```



## 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) { ... }
    }
}
```



## 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 vláknu „zpřístupní“ 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**



## Kritická sekce – **synchronized**

- Kritickou sekci deklarujeme příkazem **synchronized** s 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 vláknu
  - Vláknem, které první vstoupí do kritické sekce může používat zdroje „chráněné“ daným monitorem
  - Ostatní vlákna čekají, dokud aktivní vláknem neopustí kritickou sekci a tím uvolní zámek

*Případně zavolá **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.*



## Komunikace mezi vlákny

- Vlákna jsou objekty a mohou si 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*



## Priority vláken

- `setPriority` – nastavení priority
- `getPriority` – zjištění priority
- Hodnoty priority – `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`
- Předání řízení lze vynutit voláním `yield()`



# Část III

## Část 3 – Využití vláken v GUI



# Obsah

## Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker



## 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>



# Obsah

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker



# Samostatné výpočetní vlákno pro výpočetní model v aplikaci DemoBarComp

- Třidu `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`*

`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`



# Obsah

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy SwingWorker



# 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 třída
- **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();
        lec05/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);  
}
```

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(); //arg can be further processed  
    }  
    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



# Obsah

Vlákna v GUI (Swing)

Rozšíření výpočetního modulu v aplikaci DemoBarComp o vlákno

Návrhový vzor Observer

Využití třídy 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** 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();  
    }  
}
```

lec05/DemoBarComp-swingworker



## Příklad použití třídy **SwingWorker** 2/3

- Všechna ostatní rozšíření realizujeme 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 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();  
        }  
    }  
}
```

lec05/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();  
    }  
}
```

lec05/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



# 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
- Příklady vláken v GUI (Swing)
  - Návrhový vzor **Observer**
  - **SwingWorker**
- **Příště: Modely vícevláknových aplikace, příklady**

