Vícevláknové aplikace

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 7

B0B36PJV – Programování v JAVA

Část 1 – Paralelní programování

Část 2 – Vícevláknové aplikace

Vlákna - terminologie, použití

Vícevláknové aplikace v operačním systému

Vlákna v Javě

Část I

Část 1 – Paralelní programování

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"

Výpočetní proces

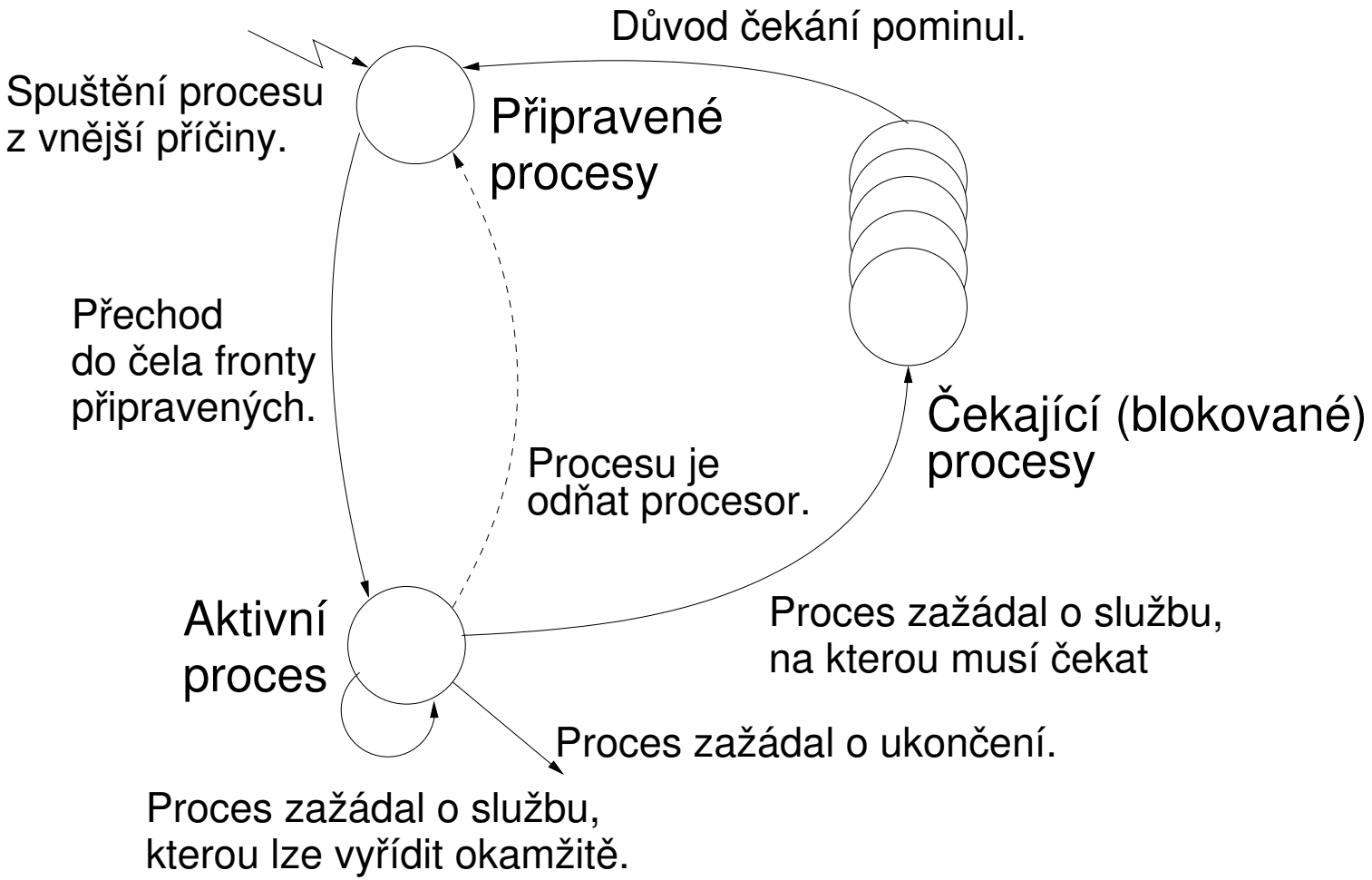
Proces je spuštěný program ve vyhrazeném prostoru paměti. Jedná se o entitu operačního systému, která je plánována pro nezávislé provádění.

Stavy procesu:

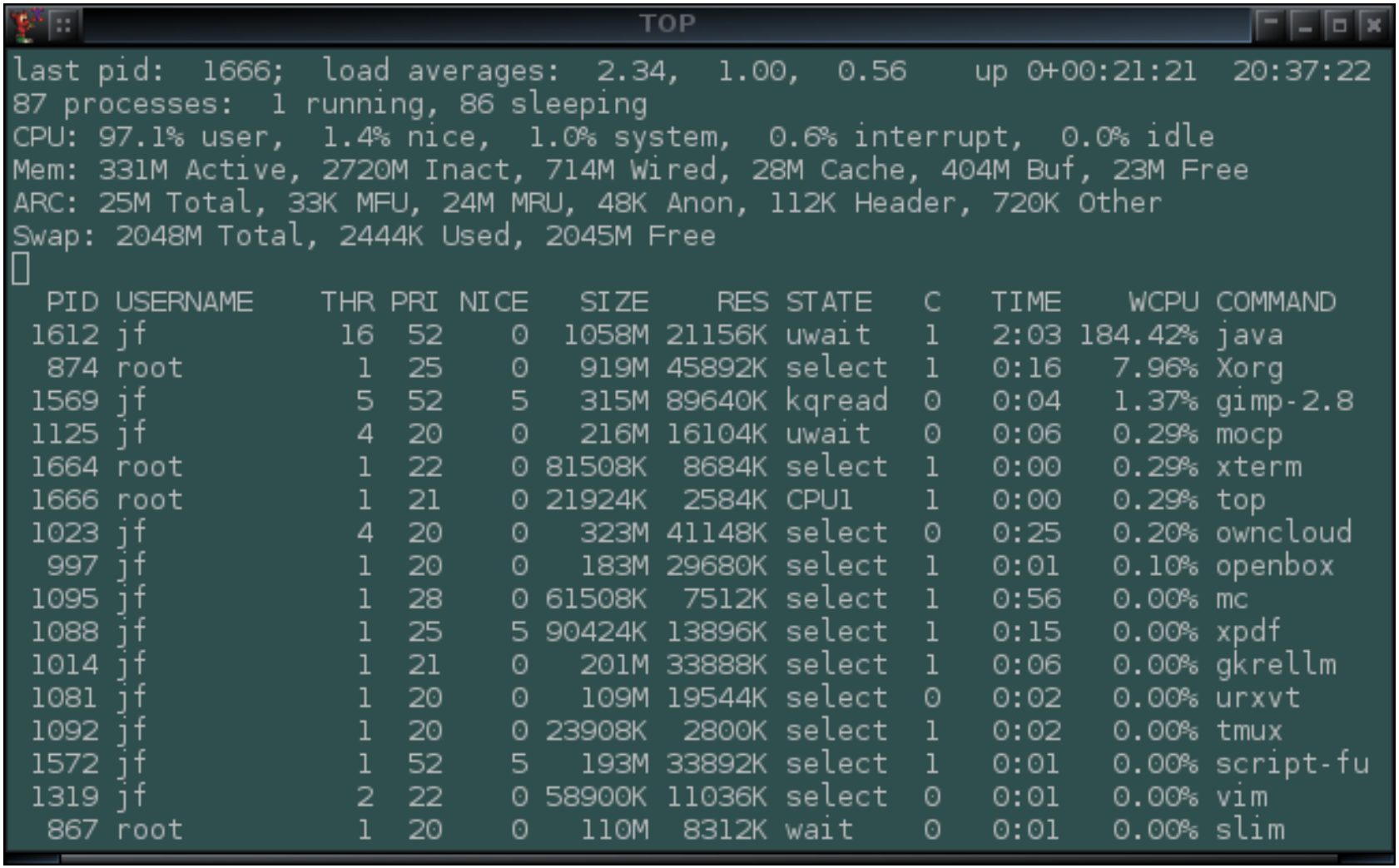
- **Executing** - právě běžící na procesoru.
- **Blocked** - čekající na periferie.
- **Waiting** - čekající na procesor.

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



V současných operačních systémech typicky běží celá řada procesů v pseudoparalní/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.

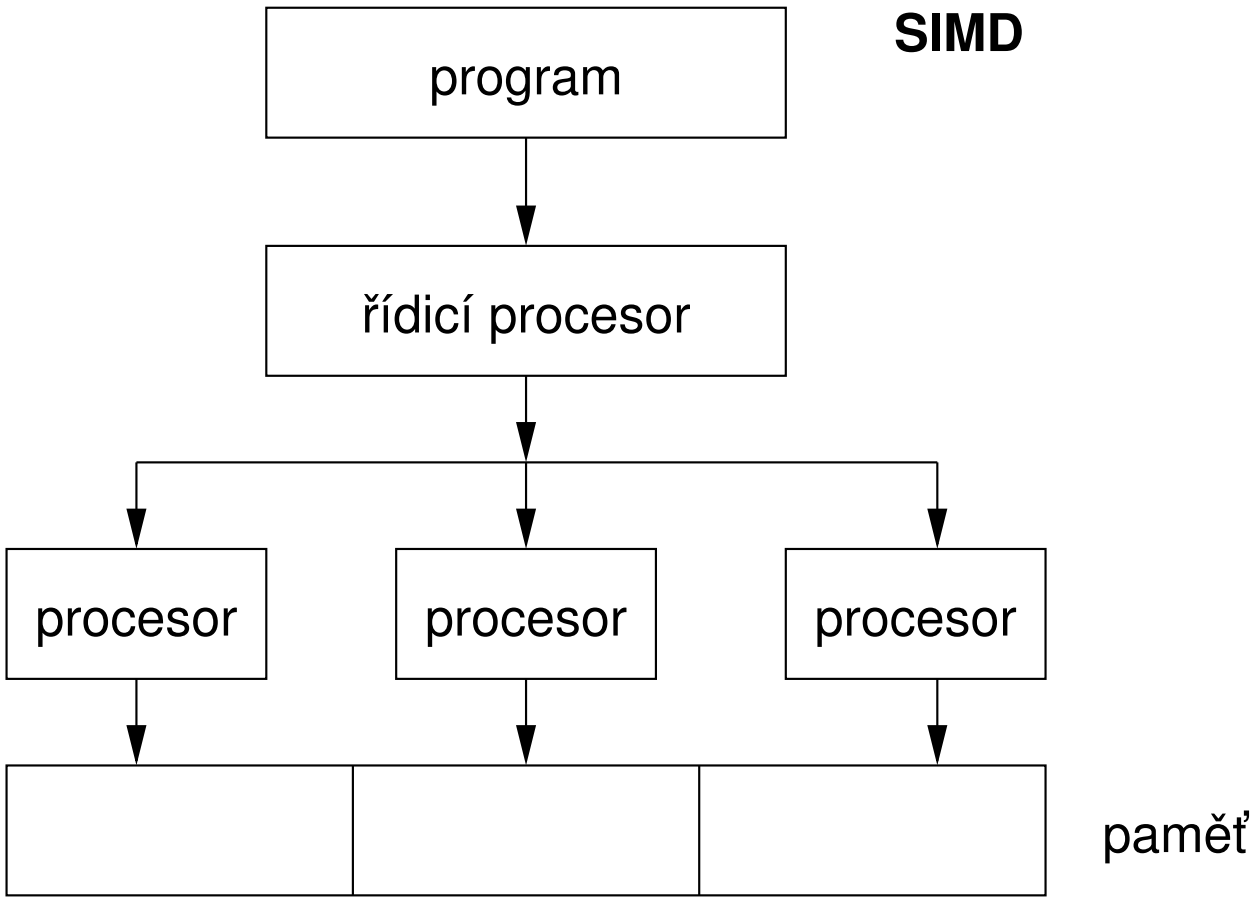
- SIMD (single-instruction, multiple-data) - stejné instrukce jsou vykonávány na více datech. Procesory jsou identické a pracují synchronně. *Příkladem může být vykonávání MMX, SEE, 3dnow! instrukcí, „vektorizace".*
- MIMD (multiple-instruction, multiple-data) - procesory pracují nezávisle a asynchronně.

Řízení přístupu k paměti.

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

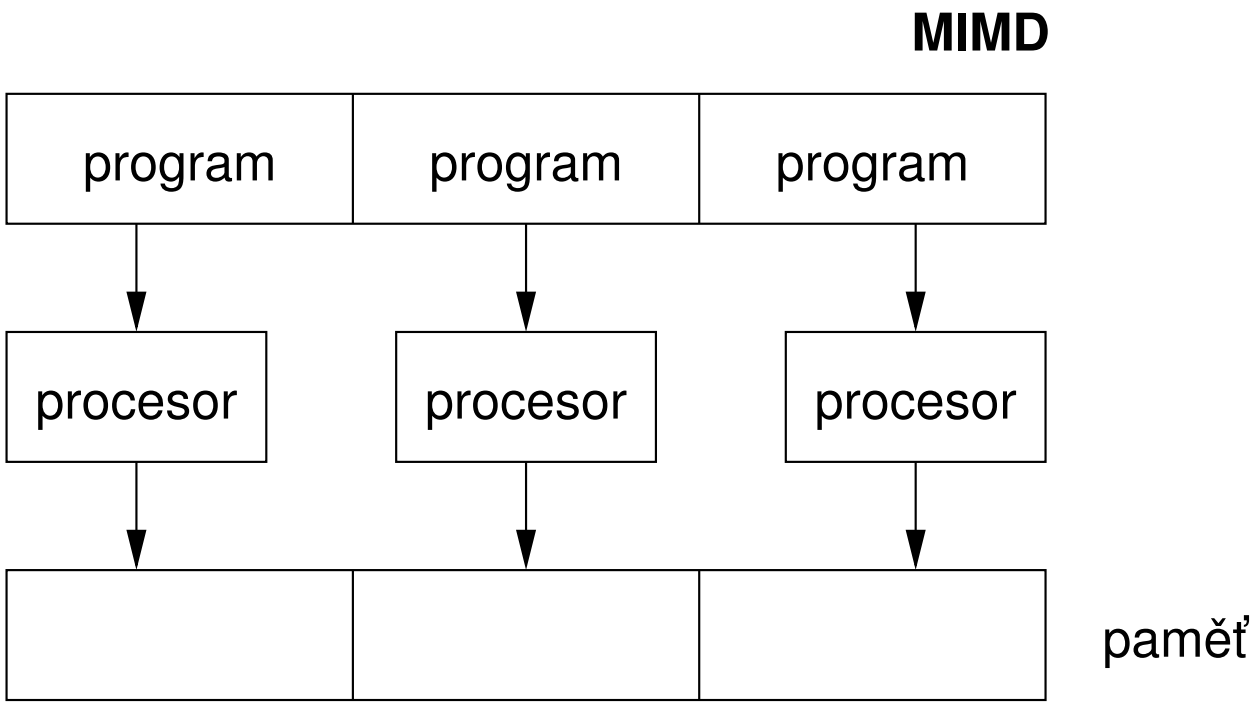
Informativní

SIMD



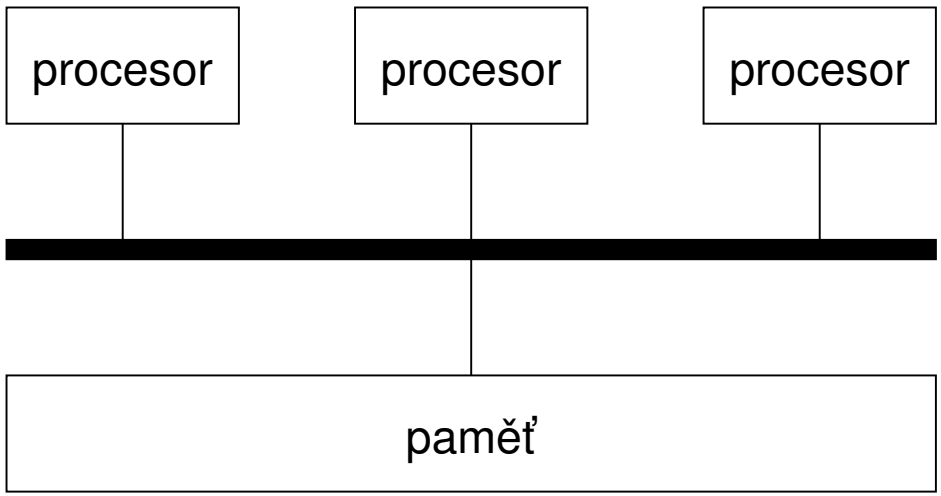
Informativní

MIMD



Informativní

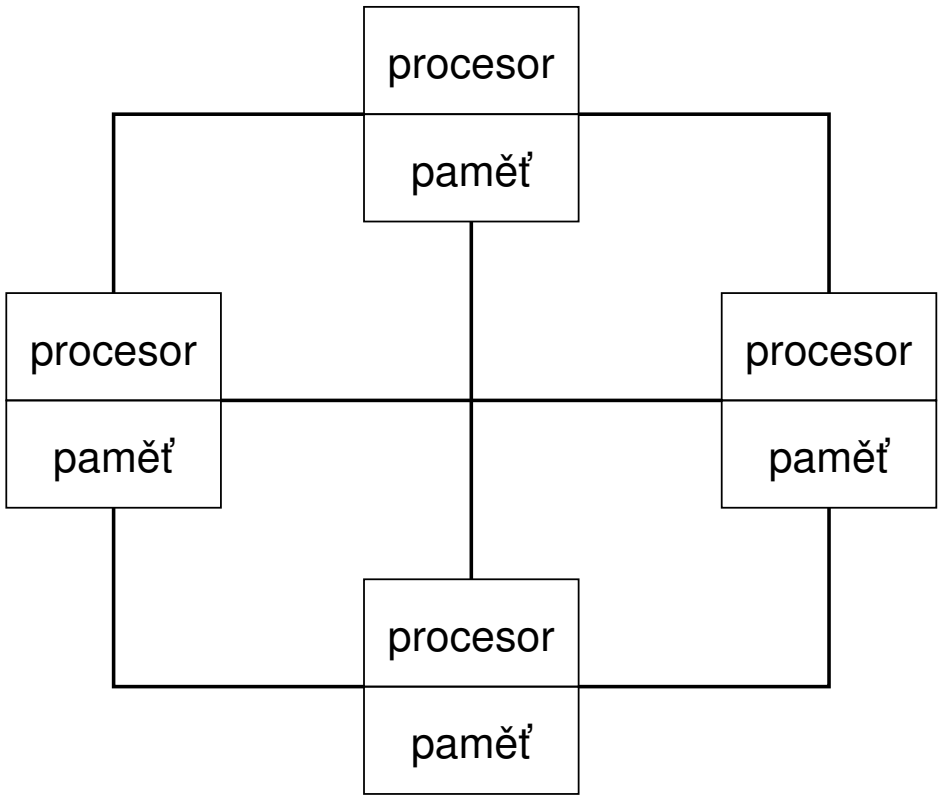
Systémy se sdílenou pamětí



- 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

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

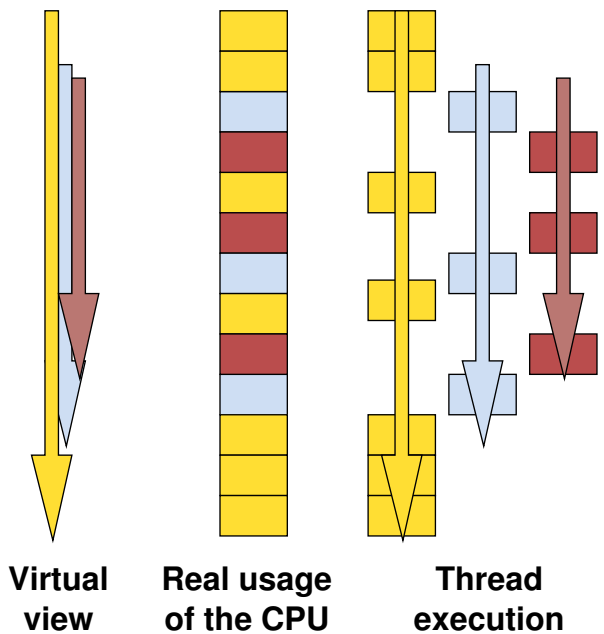
       Např. automatická „vektorizace"
     - Paralelní konstrukce explicitně označit pro kompilátor.

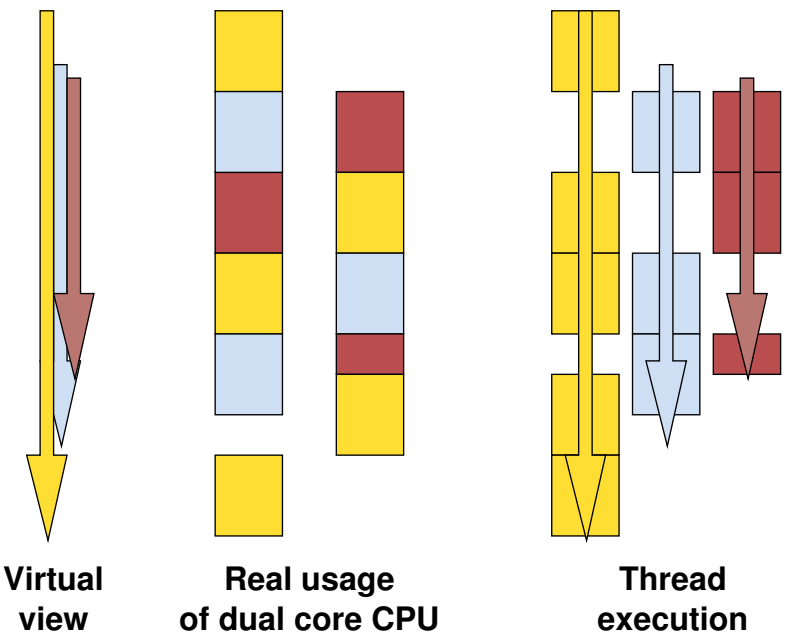
       Např. OpenMP
     - Využití služeb operačního systému pro paralelní zpracování.
  2. Jazyky s explicitní podporou paralelismu
     - Nabízejí výrazové prostředky pro vznik nového procesu (výpočetního toku)

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

Synchronizace výpočetních toků

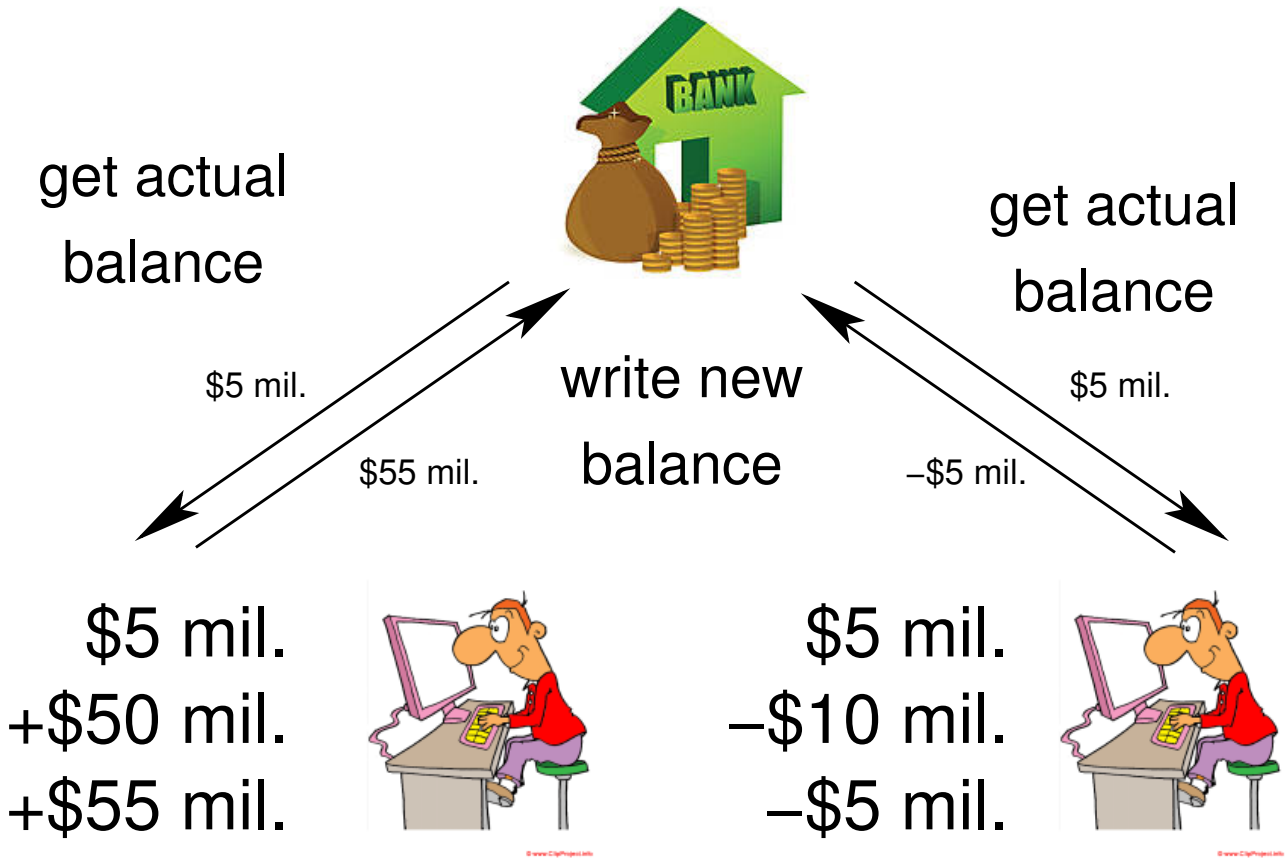
- Klíčovým problémem paralelního programování je, jak zajisti efektivní sdílení prostředků a zabránit kolizím
- Je nutné řešení problémů vzniklých z možného paralelního běhu bez ohledu na to, zdali se jedná o skutečně paralelní nebo pseudoparalelní prostředí





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).

Semafory

- Základním prostředkem pro synchronizaci v modelu se sdílenou pamětí je **Semafor**

E. W. Dijkstra

- Semafor je proměnná typu integer, přístupná operacemi:
  - *InitSem* - inicializace.
  - *Wait* - $\begin{cases} S > 0 - S = S - 1 \\ \text{jinak - pozastavuje činnost volajícího procesu.} \end{cases}$
  - *Signal* - $\begin{cases} \text{probudí nějaký čekající proces pokud existuje} \\ \text{jinak - } S = S + 1. \end{cases}$
- Semafory 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 semafory

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

- Synchronizace procesů semafory.

Příklad synchronizace procesů

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

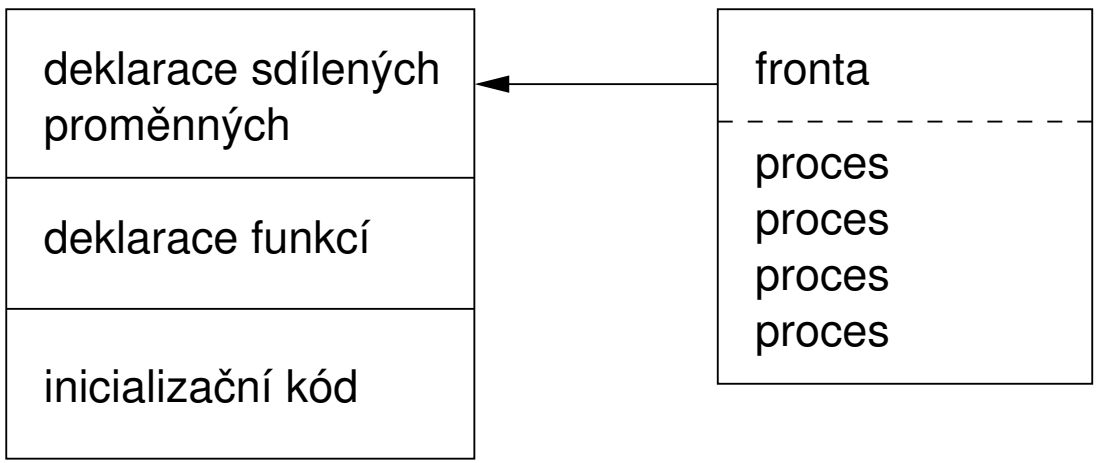
```
/* process q */
...
Signal(S); exit();
```

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

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 blokovaném 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 událostí, 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.

Vícevláknové a víceprocesové aplikace

Vícevláknová aplikace má oproti více procesové aplikaci výhody:

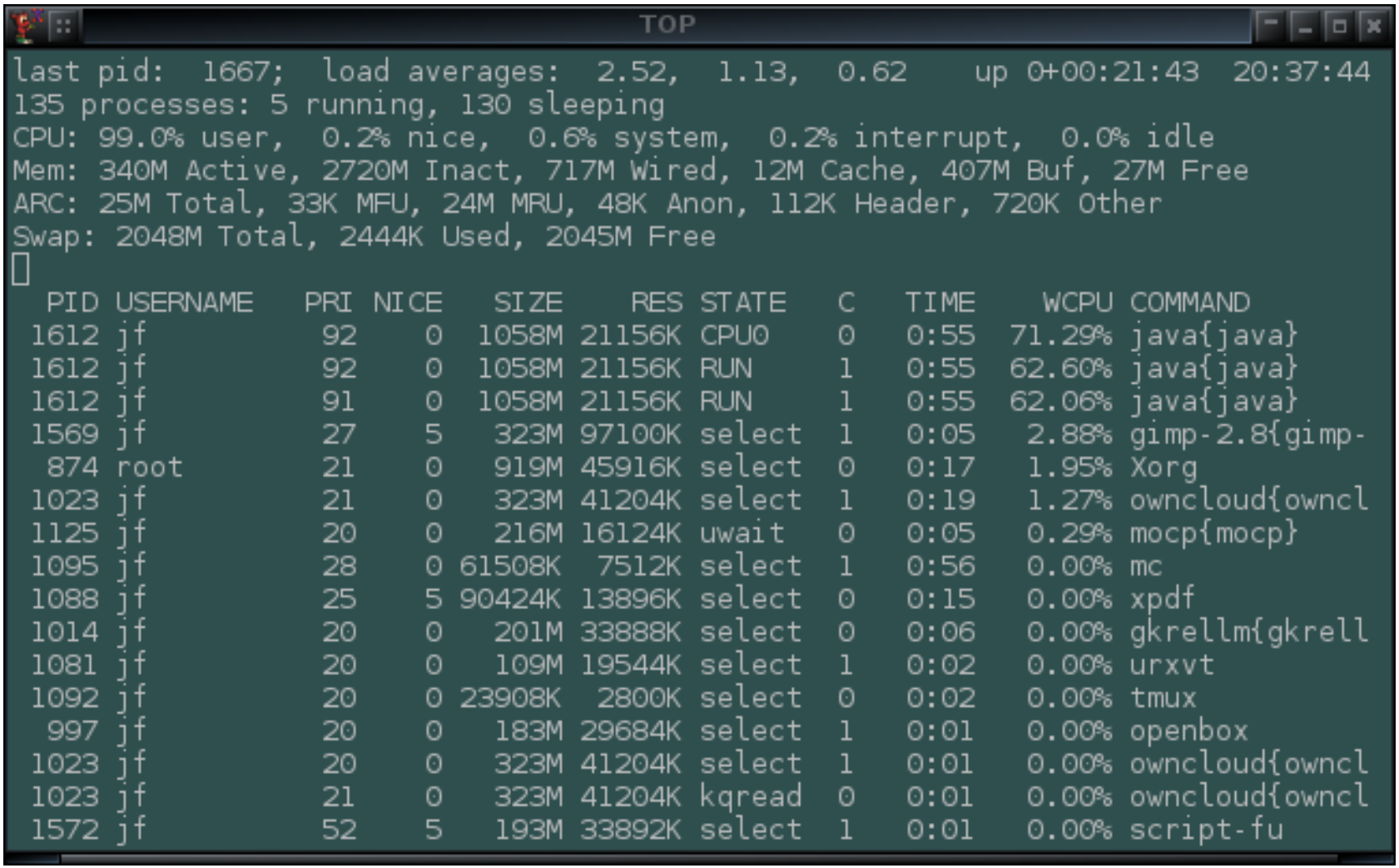
- Aplikace je mnohem interaktivnější.
- Snadnější a rychlejší komunikace mezi vlákny (stejný paměťový prostor).

Nevýhody:

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

I na jednoprocesorových systémech vícevláknové aplikace lépe využívají CPU.

Příklad výpisu procesů a jim příslušejících vláken

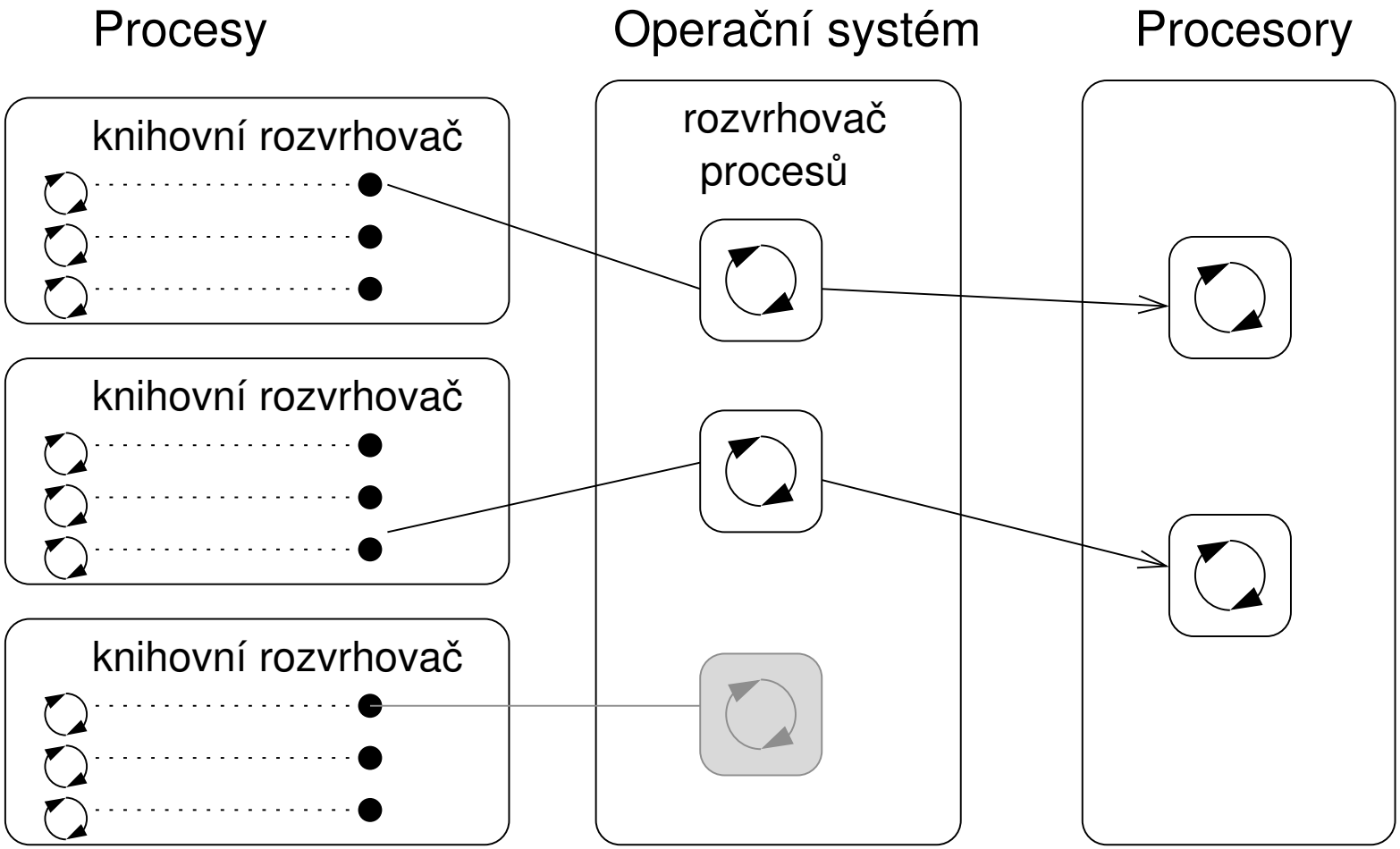


Jeden proces může být rozdělen na více vláken, která jsou v tomto případě rozvrhována operačním systémem na dostupné procesory.

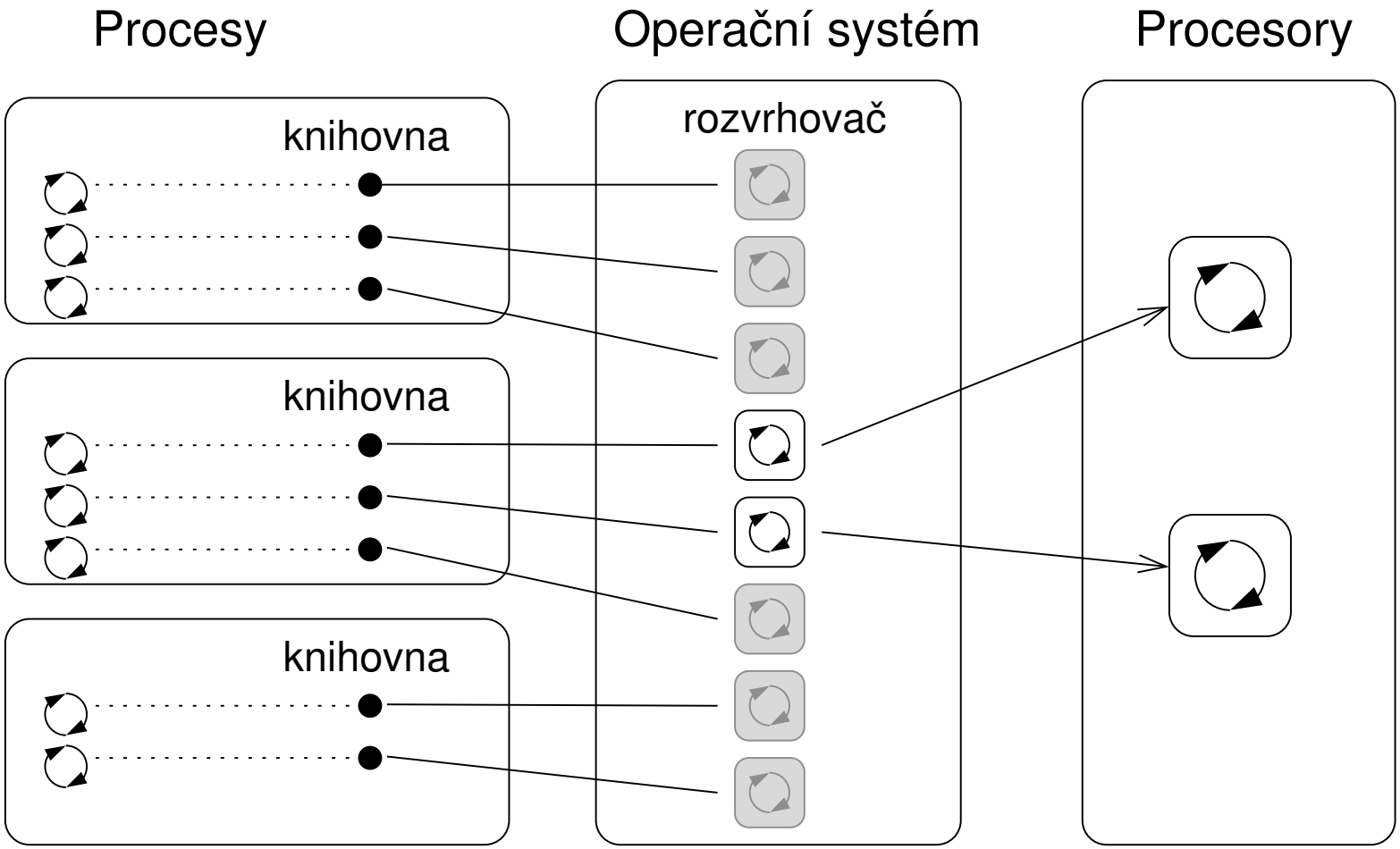
Vlákna v operačním systému

- 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 uživatelském prostoru



Vlákna v prostoru jádra operačního systému



Uživatelský vs jaderný prostor vláken

Uživatelský prostor

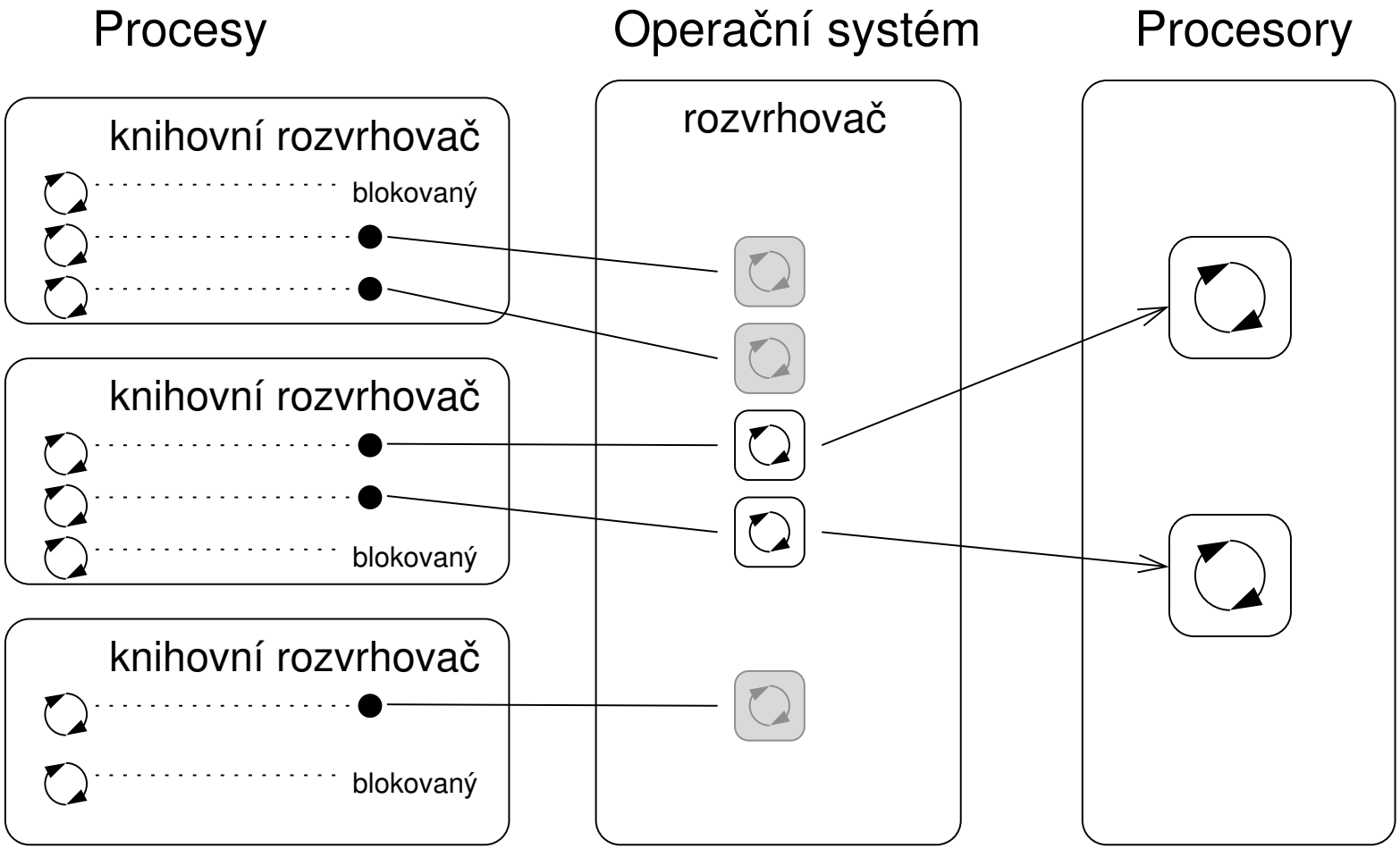
+ Není potřeba podpory OS.
+ Vytvoření nepotřebuje náročné systémové volání.
- Priority vláken se uplatňují pouze v rámci přiděleného časového kvanta procesu.
- Nemohou běžet paralelně.

Prostor jádra

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

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

Kombinace uživatelského a jaderného prostoru



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());
    }
}
```

lec07/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 ukončeny 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 ...");
}
```

lec07/DemoThreads

Ukončení činnosti vlákna

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

  V zásadě jediný korektní způsob!

- Ve vlákně **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) { ... }
} }
```

lec07/ThreadKiller

Synchronizace činnosti vláken – **monitor**

- V případě spolupracujících vláken je nutné řešit problém sdílení datového prostoru
  - Řešení problému souběhu – tj. problém současného přístup 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
  - 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 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()`

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
- Příště: Modely vícevláknových aplikací, příklady