

# Vícevláknové aplikace – modely a příklady

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 6

A0B36PR2 – Programování 2



# Část 1 – Příklad – GUI aplikace Simulátor/Plátno

## GUI s plátnem

- Struktura aplikace

- Struktura simulátoru

- Struktura grafického rozhraní

- Praktické ukázky



## Část 2 – Spuštění externího programu v Javě

Spuštění jiného programu z procesu



# Část 3 – Sokety v Javě

## Základní informace

Stručný úvod do síťování

## Síťová API

Soket

Třídy UDP a TCP soketů

Ošetření výjimečných stavů

## Příklad - Klient / Server

Popis činnosti

Komunikační protokol

Implementace



# Část 4 – Modely vícevláknových aplikací

Modely více-vláknových aplikací

Prostředky ladění



# Část I

## Část 1 – Příklad – GUI aplikace Simulator/Plátno



# Obsah

## GUI s plátnem

- Struktura aplikace

- Struktura simulátoru

- Struktura grafického rozhraní

- Praktické ukázky



# Zadání

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

*Interaktivní hra vs Simulace*





# Základní struktura aplikace

## ■ Simulátor – obsahuje svět a objekty

- V zásadě se chová jako kolekce simulačních objektů

Iterable

- Simulace běží v samostatném vlákně s periodou `PERIOD`
- Simulace probíhá v diskrétních časových okamžicích voláním metody `doStep` simulačních objektů
- Má metodu pro zastavení vlákna `shutdown`

## ■ Grafické rozhraní a vizualizace – obsahuje

- Základní kontrolní tlačítka pro ovládání simulace (start/stop)
- Plátno pro vykreslení dílčích objektů
- Standardní vykreslovací Swing vlákno
- Samostatné vlákno pro překreslování stavu simulátoru

`SwingWorker` s přeposíláním zpráv hlavnímu Swing vláknu

- Grafickou reprezentaci vykreslovaných objektů

*Vlastní vykreslení grafickými primitivy.*



## Simulator – World – SimObject

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

*Pro jednoduchost identické jako rozměry okna/plátna*

- **SimObject** – jednotné rozhraní simulačního objektu
  - Aktuální pozice objektu – `public Coords getPosition();`
  - Provedení jednoho simulačního kroku – objekt má definované chování `public void doStep();`
- Simulace probíhá ve smyčce:

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



## Struktura grafického rozhraní

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

`SwingWorker()`

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

*Základní koncept překreslení – neodpovídá kódu*



## Struktura plátna MyCanvas a vizualizace

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

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

- Vlastní tvar objektu je definován ve třídě **Shape**

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



## Příklad definice tvaru – ShapeMonster, ShapeNPC

### ■ ShapeMonster

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

### ■ ShapeNPC

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



## Vytvoření simulačních objektů a jejich tvarů

```
private Simulator sim;
private MyCanvas canvas;

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

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

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

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



## Příklad – CanvasDemo

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

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

lec06/CanvasDemo



## Simulace vs grafická hra

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

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

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

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

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





## Část II

# Část 2 – Spuštění externího programu v Javě



# Obsah

## Spuštění jiného programu z procesu



## Příklad - spuštění jiného programu z procesu

### Příklad - spuštění programu tdijkstra z Javy

```

1 private boolean callDijkstra() {
2     boolean ret = false;
3     try {
4         String cmd = "./tdijkstra -h -n " + size + " -s " + seed;
5         Process child = Runtime.getRuntime().exec(cmd);
6         child.waitFor();
7         if (child.exitValue() == 0) {
8             BufferedReader out = new BufferedReader(new
9                 InputStreamReader(child.getInputStream()));
10            hash = Integer.parseInt(out.readLine());
11            ret = true;
12        } else {
13            System.out.println("Error in dijsktra");
14        }
15    } catch (Exception e) {
16        System.out.println("Error: Exception : " + e.getMessage());
17    }
18    return ret;
19 }

```

*Co se stane při nedostatečné vyrovnávací paměti pro stdout.*



## Příklad - spuštění jiného programu z procesu

### Příklad - spuštění programu tdijkstra z Javy

```

1 private boolean callDijkstra() {
2     boolean ret = false;
3     try {
4         String cmd = "./tdijkstra -h -n " + size + " -s " + seed;
5         Process child = Runtime.getRuntime().exec(cmd);
6         child.waitFor();
7         if (child.exitValue() == 0) {
8             BufferedReader out = new BufferedReader(new
9                 InputStreamReader(child.getInputStream()));
10            hash = Integer.parseInt(out.readLine());
11            ret = true;
12        } else {
13            System.out.println("Error in dijsktra");
14        }
15    } catch (Exception e) {
16        System.out.println("Error: Exception : " + e.getMessage());
17    }
18    return ret;
19 }

```

*Co se stane při nedostatečné vyrovnávací paměti pro stdout.*



# Část III

## Část 3 – Sokety v Javě



# Obsah

## Základní informace

Stručný úvod do síťování

## Síťová API

Soket

Třídy UDP a TCP soketů

Ošetření výjimečných stavů

## Příklad - Klient / Server

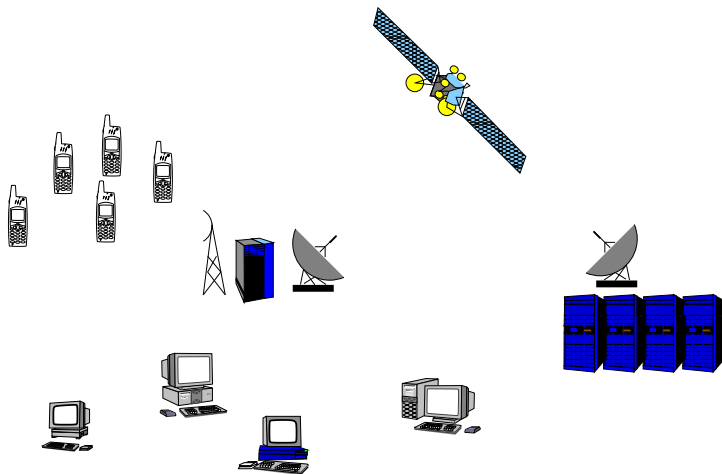
Popis činnosti

Komunikační protokol

Implementace



# Co je síťování?



# Zdroje



Jiří Peterka,

[http://www.earchiv.cz/i\\_prednasky.php3](http://www.earchiv.cz/i_prednasky.php3)



RFC - Request for Comments,  
série poznámek o Internetu.



Martin Majer,

<http://www.root.cz/clanky/sitovani-v-jave-uvod/>



W. Richard Stevens,

*UNIX Network Programming.*

Prentice Hall.



W. Richard Stevens and Stephen A. Rago,

*Advanced Programming in the UNIX Environment.*

Addison Wesley.





# Komunikace

- Komunikace slouží k přenosu informace.
- Přenos informace se děje výměnou zpráv.
- Mechanismus výměny zpráv musí mít definovaná pravidla.
- Typicky lze definovat:
  - zahájení komunikace,
  - předání zprávy,
  - reakce na zprávu,
  - ukončení komunikace.



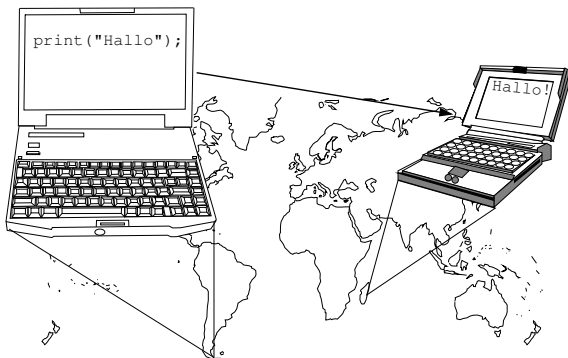
# Protokol

- Způsob komunikace definuje komunikační protokol.
- Protokol definuje:
  - formát zpráv,
  - pořadí výměny zpráv,
  - syntaxi zpráv,
  - sémantiku zpráv,
  - chování při příjmu a vyslání zprávy.



## Přenos bitů/bytů

Z uživatelského hlediska jde o přenos obsahu sdělení.

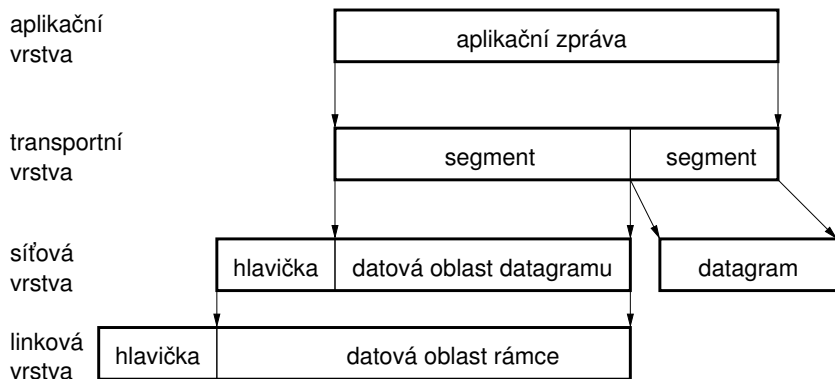


Přenos však vyžaduje další informace související s přenosovou cestou. „Výsledná velikost přenášených dat je vyšší.“

*Příklad telnet připojení k webovému serveru.*



# Zapouzdřování datových jednotek



## Modely komunikace

Typická síťová aplikace se skládá ze dvou částí:

- Server - reprezentuje služby.
- Klient - reprezentuje poptávku po službě.

Modely komunikace jsou:

- klient/server - klient žádá o službu server. *Webový server, poštovní server, Instant Messaging (IM), vzdálená sezení.*
- peer-to-peer (P2P) - každý účastník vystupuje jako klient i jako server. *Služby sdílení souborů, bittorrent, ....*



# Způsoby komunikace

Kritéria dělení komunikace.

- Podle způsobu navazování spojení:
  - spojovaná komunikace,
  - nespojovaná komunikace.
- Podle způsobu přenosu data:
  - proudový přenos,
  - blokový přenos.
- Podle kvality přenosu a garance kvality přenosu:
  - spolehlivý,
  - s garantovanou kvalitou,
  - nespolehlivý,
  - bez řízení kvality.



# Spojovaná komunikace

Spojovaná komunikace (Connection oriented).

Skládá se ze tří kroků:

1. Obě strany nejdříve navazují spojení.

*Obě strany potvrdí zájem o komunikaci případně upřesní parametry vzájemné komunikace.*

2. Vlastní výměna sdělení.
3. Ukončení spojení.



## Vlastnosti spojované komunikace

- Součástí komunikace je přechod stavů účastníků.
- Přechody mezi stavy musí být koordinované.  
*„Obě strany musí být v kompatibilním stavu, aby se domluvily.“*
- Musí být ošetřovány nestandardní situace.  
*Například rozpad spojení.*
- Při přenosu zpráv je zachováno pořadí vysílaných zpráv.  
*Přijímací strana obdrží zprávy ve stejném pořadí v jakém je poslala vysílací strana.*





## Nespojovaná komunikace

- Komunikující strany nenavazují spojení.  
*Nedochází k ověřování existence druhé strany.*
- Komunikace probíhá zasíláním samostatných zpráv (datagramů).  
*Adresování zprávy*
- Není nutné komunikaci ukončovat.

Vlastnosti:

- Komunikace je bezstavová.
- Zprávy jsou přenášeny v samostatném bloku dat (datagramu), které jsou samostatně přenášeny.
- Není zaručené pořadí zpráv.



# Obsah

## Základní informace

Stručný úvod do síťování

## Síťová API

Soket

Třídy UDP a TCP soketů

Ošetření výjimečných stavů

## Příklad - Klient / Server

Popis činnosti

Komunikační protokol

Implementace



# Soket

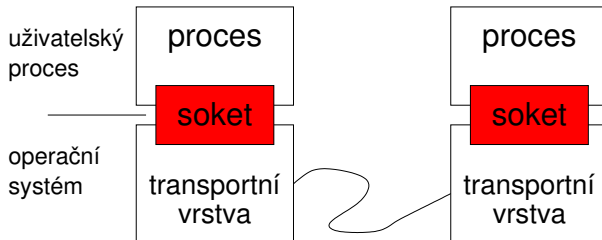
Soket je objekt, který propojuje aplikaci s nějakým „sítovým“ protokolem.

- 1981 BSD4.1 Unix.
- Soket je softwarová komponenta.
- Soket je obecný objekt komunikace mezi dvěma procesy.  
*Není omezen pouze na TCP/IP.*
- Soket API konvertuje obecnou aplikační vrstvu na specifický protokol transportní vrstvy.
- Soket API definuje operace nad soketem (primitiva).
- Soket reprezentuje koncový bod komunikace.



## Soket - aplikace a OS

Sítové rozhraní patří mezi sdílené prostředky, proto přístup k němu řídí OS.



# Sokety v Javě

## Lesson: All About Sockets

<http://docs.oracle.com/javase/tutorial/networking/sockets/index.html>

- UDP soket `java.net.DatagramSocket`
- TCP sokety:
  - `java.net.ServerSocket`
  - `java.net.Socket`
- Adresa
  - `String` `host`, `int` `port`,
  - `java.net.InetAddress`.
  - `java.net.SocketAddress`.



# UDP soket

- Datová jednotka `java.net.DatagramPacket`.
  - `DatagramPacket(byte[] buf, int length)`
  - `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
  - `byte[] getData()`
- Primitiva
  - `connect(InetAddress address, int port)`
  - `bind(SocketAddress addr)`
  - `disconnect()`
  - `close()`
  - `receive(DatagramPacket p)`
  - `send(DatagramPacket p)`  
*Cílová adresa je součástí datagramu.*



# TCP soket

## ■ Server soket primitiva

- `bind(SocketAddress endpoint)`
- `Socket accept()`
- `close()`

## ■ Soket (klientský) primitiva

- `connect(SocketAddress endpoint)`
- `connect(SocketAddress endpoint, int timeout)`
- `bind(SocketAddress bindpoint)`  
*Jaké rozhraní a jaký port chceme použít pro spojení (`null`).*
- `close()`
- Zápis a čtení je realizováno proudy.
  - `OutputStream getOutputStream()`
  - `InputStream getInputStream()`



## Ošetření výjimečných stavů

Mechanismem výjimek `java.net.SocketException`, resp. `java.io.IOException`.

- `BindException`
- `ConnectException`
- `NoRouteToHostException`
- `ProtocolException`
- `SocketException`
- `SocketTimeoutException`
- `UnknownHostException`





# Obsah

## Základní informace

Stručný úvod do síťování

## Síťová API

Soket

Třídy UDP a TCP soketů

Ošetření výjimečných stavů

## Příklad - Klient / Server

Popis činnosti

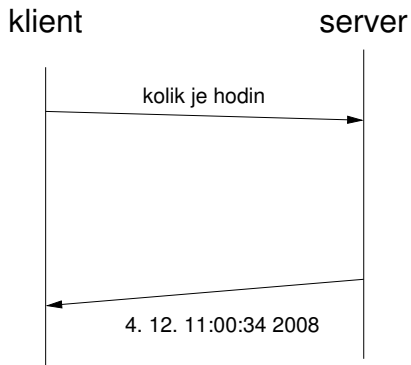
Komunikační protokol

Implementace



# Aplikace Klient / Server

Časový server



## Popis činnosti

- Jednoduchý *telnet* server s dvěma příkazy.
  - `time` pošle aktuální čas serveru.
  - `bye` ukončení sezení.
- Textově orientované spojení.
- Po navázání spojení (TCP) musí klient poslat uživatelské jméno a heslo.



## Definice protokolu

1. Po navázání spojení server posílá výzvu 'Username:'.
2. Klient odpovídá posláním uživatelského jména zakončeného znakem '\n'.
3. Server posílá výzvu 'Password:'.
4. Klient odpovídá posláním hesla zakončeného znakem '\n'.
5. Server odpovídá zprávou 'Welcome\n'.
6. Klient může posílat serveru příkazy v libovolném pořadí.



## Definice protokolu - příkazy

- Příkaz se skládá ze jména příkazu a znaku konce řádky `'\n'`.
- Server odpovídá textovou zprávou závislou na příkazu, ukončenou `'\n'`.
- Příkazy:
  - Žádost o zaslání aktuálního času.
    1. Klient: `'time\n'`.
    2. Server: posílá aktuální čas ve formátu `'time is: dow mon dd hh:mm:ss zzz yyyy\n'`.
  - Ukončení spojení.
    1. Klient: `'bye\n'`.
    2. Server: posílá konec proudu a zavírá soket.



# Implementace

Implementaci rozdělíme na třídy:

- `ParseMessage` - realizuje čtení a zápis textové zprávy z/do proudu.
  - Obsah textové zprávy může začínat a nebo končit zadanou sekvencí znaků.
- `Server` - otevírá serverový soket na zadaném portu, po přijetí klienta vytváří ovladač klientského spojení.
- `ClientHandler` - realizuje obsluhu klientského spojení v samostatném vlákně.
- `Client` - testovací klient, který se připojí k serveru na zadané adrese a portu.  
*Klient pošle uživatelské jméno, heslo a žádost o aktuální čas, který vypíše na obrazovku (pouze časový údaj) a skončí.*  
*Vše proběhne bez interakce uživatele.*



## ParseMessage

```
1 class ParseMessage {
2     void write(String msg) throws IOException {
3         out.write(msg.getBytes());
4     }
5     String read(String startStr, String endStr) throws
        IOException {
6         byte[] start = startStr.getBytes();
7         byte[] end = endStr.getBytes();
8         int sI = 0; int eI = 0; byte r; int count = 0;
9         while((sI < start.length)
10             && ((r = (byte)in.read()) != -1)) {
11             sI = (r == start[sI]) ? sI+1 : 0;
12         }
13         while ((eI < end.length) && (count < BUFSIZE)
14             && ((r = (byte)in.read()) != -1)) {
15             buffer[count++] = r;
16             eI = (r == end[eI]) ? eI+1 : 0;
17         }
18         return new String(buffer, 0,
19             count > end.length ? count-end.length : 0);
20     }
21 }
```



# ClientHandler 1/3

```
1 class ClientHandler extends ParseMessage implements
    Runnable {
2     static final int UNKNOWN = -1;
3     static final int TIME = 0;
4     static final int BYE = 1;
5     static final int NUMBER = 2;
6     static final String[] STRCMD = {"time", "bye"};
7
8     static int parseCmd(String str) {
9         int ret = UNKNOWN;
10        for (int i = 0; i < NUMBER; i++) {
11            if (str.compareTo(STRCMD[i]) == 0) {
12                ret = i;
13                break;
14            }
15        }
16        return ret;
17    }
18
19    Socket sock; //klientský soket
20    int id; //číslo klientu
```





## ClientHandler 2/3

```
1 ClientHandler(Socket iSocket, int iID) throws IOException
    {
2     sock = iSocket;
3     id = iID;
4     out = sock.getOutputStream();
5     in = sock.getInputStream();
6 }
7 public void start() { new Thread(this).start(); }
8 void log(String str) {System.out.println(str)};
9
10 public void run() {
11     String cID = "client["+id+"] ";
12     try {
13         log(cID + "Accepted");
14         write("Login:");
15         log(cID + "Username:" + read("", "\n"));
16         write("Password:");
17         log(cID + "Password:" + read("", "\n"));
18         write("Welcome\n");
```



## ClientHandler 3/3

```
1  ... //run pokračování
2  boolean quit = false;
3  while (!quit) {
4      switch(parseCmd(read("", "\n"))) {
5          case TIME:
6              write("time is:" + new Date().toString() + "\n");
7              break;
8          case BYE:
9              log(cID + "Client sends bye");
10             quit = true;
11             break;
12         default:
13             log(cID + "Unknown message, disconnect");
14             quit = true;
15             break;
16     } }
17 sock.shutdownOutput(); sock.close();
18 } catch (Exception e) {
19     log(cID + "Exception:" + e.getMessage());
20     e.printStackTrace();
21 } } }
```



# Server

```
1 public class Server {
2     public Server(int port) throws IOException {
3         int i = 0;
4         ServerSocket servsock = new ServerSocket(port);
5         while (true) {
6             try {
7                 new ClientHandler(servsock.accept(), i++);
8             } catch (IOException e) {
9                 System.out.println("IO error in new client");
10            } }
11 } // Server()
12
13 public static void main(String[] args) {
14     try {
15         new Server(args.length > 0 ?
16             Integer.parseInt(args[0]) : 9000);
17     } catch (Exception e) {
18         e.printStackTrace();
19     }
20 }
21 }
```



# Client 1/2

```
1 public class Client extends ParseMessage {
2     Socket sock;
3     public static void main(String[] args) {
4         Client c = new Client(
5             args.length > 0 ? args[0] : "localhost",
6             args.length > 1 ? Integer.parseInt(args[1]) : 9000
7         );
8     }
9
10    Client(String host, int port) {
11        try {
12            sock = new Socket();
13            sock.connect(new InetSocketAddress(host, port));
14            out = sock.getOutputStream();
15            in = sock.getInputStream();
```



## Client 2/2

```
1 //Client konstruktor pokračování
2 write("user\n");
3 read("", "Password:");
4 System.out.println("Password prompt readed");
5 write("heslo\n");
6 read("", "Welcome\n");
7 write("time\n");
8 out.flush();
9 System.out.println("Time on server is " + read("time
10 is:", "\n"));
11 write("bye\n");
12 sock.shutdownOutput();
13 sock.close();
14 System.out.println("Communication END");
15 } catch (Exception e) {
16     System.out.println("Exception:" + e.getMessage());
17 }
18 }
```



## Ukázka činnosti

### Příklad – Telnet

```
1 oredre$ java Telnet
2 Login:telnet
3 Password:tel
4 Welcome
5 time
6 time is:Tue Nov 28 09:56:49
   CET 2006
7 time
8 time is:Tue Nov 28 09:56:50
   CET 2006
9 bye
```

### Příklad – Server

```
1 oredre$ java Server
2 client[0] Accepted
3 client[0] Username:telnet
4 client[1] Accepted
5 client[1] Username:user
6 client[1] Password:heslo
7 client[1] Client sends bye
8 client[0] Password:tel
9 client[0] Client sends bye
```

### Příklad – Klient

```
1 oredre$ java Client
2 Password prompt readed
3 Time on server is Tue Nov 28 09:56:40 CET 2006
4 Communication END
```

lec06/socket

*Demonstrační příklad se zdrojovými soubory z roku 2006, kterému odpovídá kódovací styl*



# Část IV

## Část 4 – Modely vícevláknových aplikací



# Obsah

Modely více-vláknových aplikací

Prostředky ladění





## Kdy použít vlákna?

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

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



## Typické aplikace

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



# Modely vícevláknových aplikací

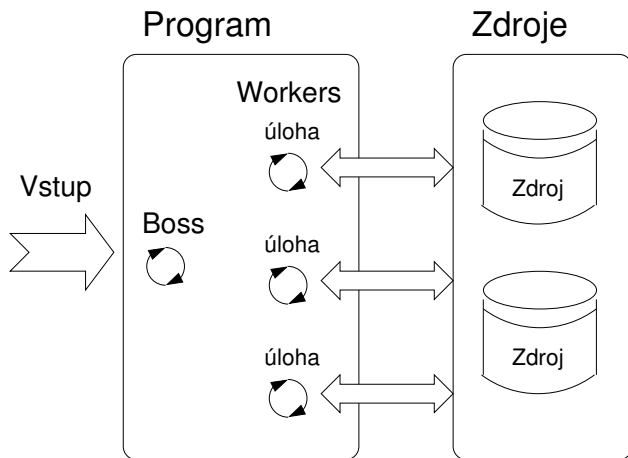
Modely řeší způsob vytváření a rozdělování práce mezi vlákna.

- **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 model



## Boss/Worker rozdělení činnosti

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



# Boss/Worker příklad

## Příklad Boss/Worker model

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

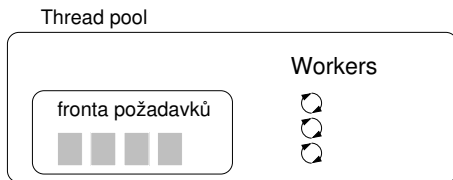
```
1 //Worker
2 taskX() {
3     řešení úlohy,
4     synchronizace v
5     případě sdílených
6     zdrojů;
7     done;
8 }
9 taskY() {
10    řešení úlohy,
11    synchronizace v
12    případě sdílených
13    zdrojů;
14    done;
15 }
```

C style



# Thread Pool

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



## Thread Pool - vlastnosti

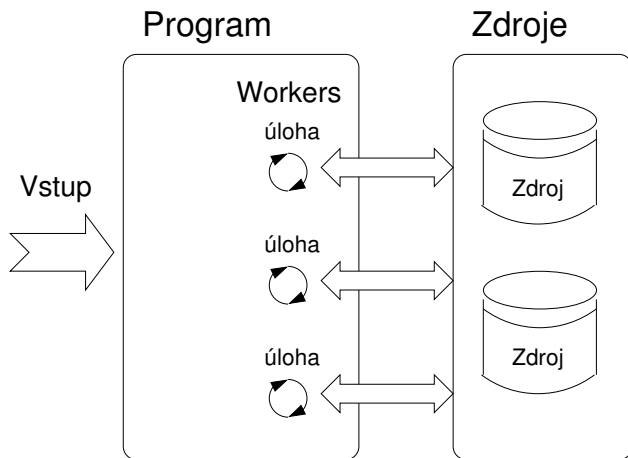
- Počet vytvořených vláken.
- Maximální počet požadavků ve frontě požadavků.
- Definice chování v případě plné fronty požadavků a žádného volného vlákna.

*Například blokování příchozích požadavků.*





## 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.



# Peer model - příklad

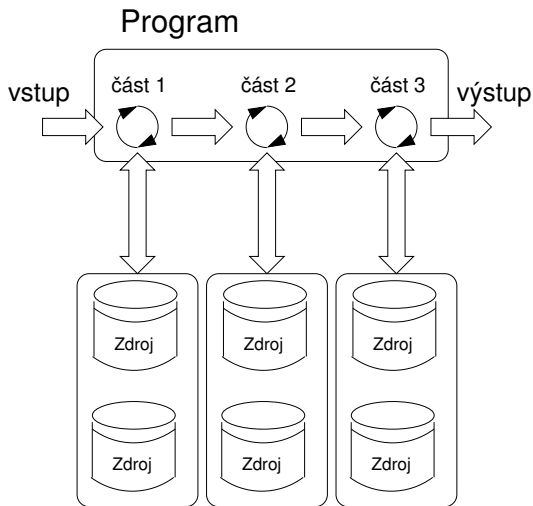
## Příklad Peer model

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

```
1 //Worker
2 task1() {
3     čekáná na spuštění;
4     řešení úlohy,
5     synchronizace v
6     případě sdílených
7     zdrojů;
8     done;
9 }
10 task2() {
11     čekáná na spuštění;
12     řešení úlohy,
13     synchronizace v
14     případě sdílených
15     zdrojů;
16     done;
17 }
```



# Zpracování proudu dat - Pipeline



# Pipeline

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



# Pipeline model - příklad

## Příklad Pipeline model

```
1  main() {
2    create_thread(stage1);
3    create_thread(stage2);
4    :
5    :
6    wait for all pipeline;
7  }
8  stage1() {
9    while(input) {
10     get next program
11     input;
12     process input;
13     pass result to next
14     stage;
15  }
```

```
1  stage2() {
2    while(input) {
3     get next input from
4     thread;
5     process input;
6     pass result to next
7     stage;
8  }
9  stageN() {
10   while(input) {
11    get next input from
12    thread;
13    process input;
14    pass result to output;
15  }
```

C style

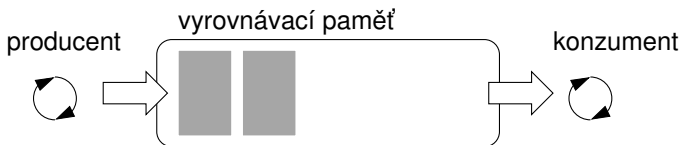


## Producent a konzument

Předávání dat mezi vlákny je realizováno vyrovnávací pamětí bufferem.

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

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



# Obsah

Modely více-vláknových aplikací

Prostředky ladění





## Funkce a paralelismus

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

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

*Např. vnořená obsluha přerušení*

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

Dosažení těchto vlastností:

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



# Vícevláknové aplikace a ladění

Hlavní problémy vícevláknových aplikací souvisí se synchronizací:

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



# Prostředky ladění

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



# Prostředky ladění

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



# Prostředky ladění

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



## Podpora ladění

### Debugger:

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

### Logování:

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



## Poznámky - „problémy souběhu“

Problémy souběhu jsou typicky způsobeny nedostatkem synchronizace.

- Vlákna jsou **asynchronní**.

*Nespoléhat na to, že na jednoprocessorovém 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 (zámků/monitorů).
- Mutex deadlock nemůže nastat, pokud má každé vlákno uzamčený pouze jeden mutex (*chce uzamknout*).
- Není dobré volat funkce s uzamčeným mutexem, obzvláště zamyká-li volaná funkce jiný mutex.
- Je dobré zamykat mutex na co možná nejkratší dobu.

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

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





# Shrnutí přednášky



# Diskutovaná témata

- Modely vícevláknových aplikací
- Paralelní programování a ladění
  - Problém uváznutí a problém souběhu
- Spuštění externího programu v rámci Java programu
- Sokety v Javě
- Příklady vícevláknových aplikací
  - GUI plátno – simulátor a kreslení do canvasu
  
- **Příště: Test - 7.4.2016!**

