

# Vlákná a konkurentné výpočty

## (pokračovanie)

dnes bude:

- komunikácia cez rúry (pipes),
- synchronizácia a kritická sekcia (semafóry),
- deadlock

literatúra:

- [Thinking in Java, 3rd Edition](#), 13.kapitola,
- [Concurrency Lesson](#), resp. [Lekcia Súbežnosť](#),
- [Java Threads Tutorial](#),
- [Introduction to Java threads](#)

Cvičenia:

- Simulácie grafické, javafx (ak treba, použiť existujúci kód),
- napr. iné triedenie, iné guľičky, plavecký bazén, lienky na priamke, ...



# Pozastavenie/uspanie vlákna

- zaťaženie vlákna (nezmyselným výpočtom) vyčerpáva procesor, potrebujeme jemnejšiu techniku,
- nasledujúci príklad ukáže, ako uspíme vlákno bez toho aby sme zaťažovali procesor nepotrebným výpočtom,
- vlákno uspíme na čas v milisekundách metódou `Thread.sleep(long millis)` throws `InterruptedException`,
- spánok vlákna môže byť prerušený metódou `Thread.interrupt()`, preto pre `sleep` musíme ošetriť výnimku `InterruptedException`,
- ak chceme počkať, kým výpočet vlákna prirodzene dobehne (umrie), použijeme metódu `Thread.join()`
- ak chceme testovať, či život vlákna bol prerušený, použijeme metódu `boolean isInterrupted()`, resp. `Thread.interrupted()`.

# Uspatie vlákna

```
public class SleepingThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SleepingThread() { ... .start(); }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0) return;
            try {
                sleep(100); // uspi na 0.1 sek.
            } catch (InterruptedException e) { // výnimku musíme ochytiť
                throw new RuntimeException(e); // spánok bol prerušený
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        for(int i = 0; i < 5; i++) {
            new SleepingThread().join(); // počkaj kým dobehne
            System.out.println("--");
        }
    }
}
```

#1: 5  
#1: 4  
#1: 3  
#1: 2  
#1: 1  
--  
#2: 5  
#2: 4  
#2: 3  
#2: 2  
#2: 1  
--  
#3: 5  
#3: 4  
#3: 3  
#3: 2  
#3: 1  
--  
#4: 5  
#4: 4  
#4: 3  
#4: 2  
#4: 1  
--  
#5: 5  
#5: 4  
#5: 3  
#5: 2  
#5: 1  
--

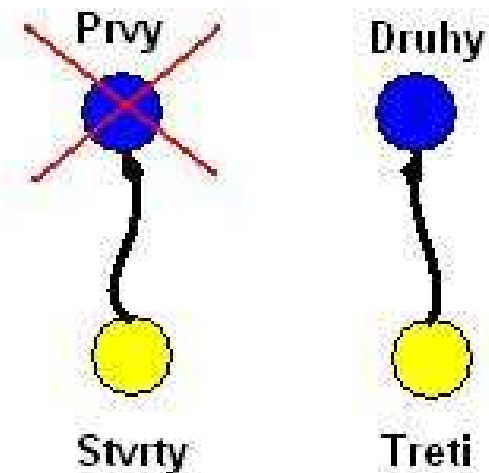
Súbor: [SleepingThread.java](#)

# Čakanie na vlákno

- nasledujúci príklad vytvorí 4 vlákna,
- dva modré (Prvy, Druhy) triedy **Sleeper**, ktorý zaspia na 1.5 sek.
- ďalšie dva žlté (Treti, Stvrty) triedy **Joiner**, ktoré sa metódou **join()** pripoja na sleeperov a čakajú, kým dobehnú,
- aby vedelo vlákno triedy **Joiner**, na koho má čakať, konštruktor triedy **Joiner** dostane odkaz na vlákno (sleepera), na ktorého má čakať,
- medzičasom, výpočet vlákna Prvy násilne zastavíme v hlavnom vlákne metódou **interrupt()**.

// hlavný thread:

```
Sleeper prvy = new Sleeper("Prvy", 1500);  
Sleeper druhy = new Sleeper("Druhy", 1500);  
Joiner treti = new Joiner("Treti", druhy);  
Joiner stvrty = new Joiner("Stvrty", prvy);  
prvy.interrupt();
```

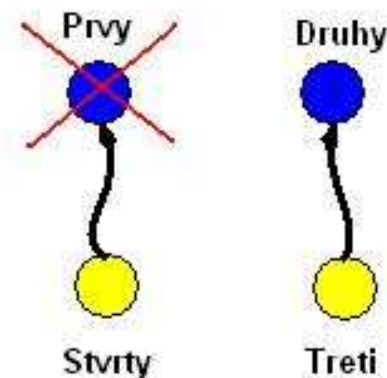


# Čakanie na vlákno - Sleeper

```
class Sleeper extends Thread {  
    private int duration;  
    public Sleeper( String name,  
                    int sleepTime) {  
        super(name);  
        duration = sleepTime;  
        start();  
    }  
    public void run() {  
        try {  
            sleep(duration);  
        } catch (InterruptedException e) {  
            System.out.println(getName() + " preruseny");  
            return;  
        }  
        System.out.println(getName() + " vyspaty");  
    }  
}
```

Súbor: **Sleeper.java**

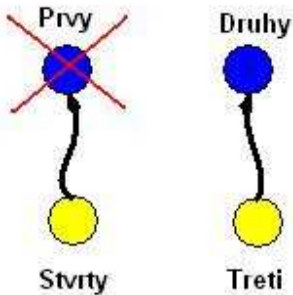
```
class Joiner extends Thread {  
    private Sleeper sleeper;  
    public Joiner(String name, Sleeper sleeper) {  
        super(name);  
        this.sleeper = sleeper;  
        start();  
    }  
    public void run() {  
        try {  
            sleeper.join();  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
        System.out.println(getName() + "dobehol");  
    }  
}
```



Súbor: [Sleeper.java](#)

# Čakanie na vlákno - Joiner

```
class Sleeper extends Thread {  
    private int duration;  
    public Sleeper(String name, int sleepTime) {  
        super(name);  
        duration = sleepTime;  
        start();  
    }  
    public void run() {  
        try {  
            sleep(duration);  
        } catch (InterruptedException e) {  
            System.out.println(getName() + " preruseny");  
            return;  
        }  
        System.out.println(getName() + " vyspaty");  
    }  
}
```



Prvy preruseny  
Stvrty dobehol  
Druhy vyspaty  
Treti dobehol

```
class Joiner extends Thread {  
    private Sleeper sleeper;  
    public Joiner(String name,  
        Sleeper sleeper) {  
        super(name);  
        this.sleeper = sleeper;  
        start();  
    }  
    public void run() {  
        try {  
            sleeper.join();  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
        System.out.println(getName() + "  
        dobehol");  
    }  
}
```

# Komunikácia medzi vláknami

- doteraz sme mali príklady vlákien, ktoré medzi sebou (počas ich behu...) nekomunikovali (ak teda nerátame za komunikáciu, že sa zabíjali),
- ak chceme, aby si vlákna vymieňali dáta, vytvoríme medzi nimi rúru (pipe),
- rúra pozostáva z jednosmerne orientovaného streamu, ktorý sa na strane zapisovača (producenta, Sender) tvári ako PipedWriter, a na strane čítača (konzumenta, Reader) ako PipedReader,
- aby čítač čítal z rúry, ktorú zapisovač pre neho vytvoril, musíme mu poslať odkaz na vytvorenú rúru PipedWriter, inak máme dve rúry...
- do rúry môžeme písať bajty, znaky, reťazce, objekty, v závislosti, ako si rúru *zabalíme* (viď techniky z I/O prednášky),
- vytvoríme objekt Sender (producent), ktorý do rúry zapíše znaky A, B, ..., z
- objekt Reader (konzument), ktorý číta znaky z rúry a vypíše A, B, ..., z

```
public class SenderReceiver {           // hlavný program
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        sender.start(); receiver.start();
    }
}
```

Súbor: [SenderReceiver.java](#)

# Výstupná rúra

```
class Sender extends Thread {  
    private Random rand = new Random();  
  
    private PipedWriter out =  
        new PipedWriter();    // vytvor rúru na zápis, rúra je ukrytá, private  
  
    public PipedWriter getPipedWriter() {  
        return out;    // daj rúru, bude ju potrebovať Reader na nadviazanie spojenia  
    }  
    public void run() {  
        while(true) {  
  
            for(char c = 'A'; c <= 'z'; c++) {  
                try {  
                    out.write(c);  
                    sleep(rand.nextInt(500));  
                } catch(Exception e) {  
                    throw new RuntimeException(e);  
                }  
            }  
        }  
    }  
}
```

// vypíš znaky abecedy do rúry  
// a za každým počkaj max. 1/2 sek.



# Vstupná rúra

```
class Receiver extends Thread {  
    private PipedReader in;  
  
    public Receiver(Sender sender) throws IOException {  
        in = new PipedReader(sender.getPipedWriter());  
    }  
    public void run() {  
        try {  
            while(true)  
                System.out.println("Read: " + (char)in.read());  
        } catch(IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

// vytvor vstupnú  
// rúru napojenú na výstupnú rúru Sendera

// čítaj zo vstupnej rúry a píš na konzolu

Read: A  
Read: B  
Read: C  
Read: D  
Read: E  
Read: F  
Read: G  
Read: H  
Read: I  
Read: J  
Read: K  
Read: L  
Read: M  
Read: N  
Read: O  
Read: P  
Read: Q  
Read: R

Súbor: [Receiver.java](#)

# Synchronizácia

- v prípade, ak dve vlákna zdieľajú nejaký zdroj, môže dôjsť k nepredvídateľnej interakcii vlákien (napr. jeden číta, druhý píše),
- spôsob, akým sa riadi prístup k zdieľaným zdrojom (synchronizácia) sa volá:
  - kritická sekcia,
  - semafor, mutex, PV operácie,
  - java monitor.
- skúsime si sami naprogramovať semafor, aby sme pochopili, prečo táto vlastnosť musí byť súčasťou jazyka, a nie naprogramovaná v jazyku,
- semafor reprezentuje celočíselná premenná semaphore inicializovaná na 0,
- ak je zdieľaný zdroj voľný, semaphore == 0,
- záujem použiť zdroj vyjadrím pomocou acquire(),
- ak prestanem používať zdroj, uvoľním ho pomocou release().
- Naivná implementácia vedie k tomu, že dve vlákna sa v istom čase dozvedia, že zdroj je voľný, oba si ho zarezervujú, a dochádza ku kolízii

# Semafór

prvý pokus

```
public class Semaphore {  
  
    // neoptimalizuj !  
    private volatile int semaphore = 0;  
  
    // môžem vojsť ?  
    public boolean available() {  
        return semaphore == 0;  
    }  
  
    // idem dnu !  
    public void acquire() {  
        ++semaphore; }  
  
    // odchádzam...  
    public void release() {  
        --semaphore; }  
}
```

```
public class SemaphoreTester  
    extends Thread {  
  
    public void run() {  
        while(true) // stále chce dnu a von  
            if(semaphore.available()) {  
                yield(); // skôr to spadne ☺  
                semaphore.acquire();  
                yield();  
                semaphore.release();  
                yield();  
            }  
    }  
  
    public static void main(String[] args)  
        throws Exception {  
        // pustíme semafor a dva testery  
        Semaphore sem=new Semaphore() .start();  
        new SemaphoreTester(sem).start();  
        new SemaphoreTester(sem) .start();  
    }  
}
```

# Synchronizovaná metóda

Riešenie: Java ponúka konštrukciu synchronized:

- **synchronizovaná metóda** – nie je možné súčasne volať dve synchronizované metódy toho istého objektu (kým sa vykonáva jedna synchronizovaná, ostatné sú pozastavené do jej skončenia).

```
public class SynchronizedSemaphore extends Semaphore {  
    private volatile int semaphore = 0;  
    public synchronized boolean available() { return semaphore == 0; }  
    public synchronized void acquire() { ++semaphore; }  
    public synchronized void release() { --semaphore; }
```

... a teraz to už pojde ?

```
public void run() {  
    while(true)  
        if(semaphore.available()) {  
            semaphore.acquire();  
            semaphore.release();  
        }  
}
```



# Synchronizovaná (kritická) sekcia

## Atomická operácia:

- sú operácie, ktoré sú nedeliteľné pre plánovač vlákien, napr. nie je možné, aby jedno vlákno zapísalo len spodné 2 bajty do premennej int,
- čítanie a zápis do premenných primitívnych typov a premenných deklarovaných ako volatile je atomická operácia.

ale

- operácie nad zložitejšími štruktúrami nemusia byť synchronizované (napr. ArrayList, HashMap, LinkedList, ... (v dokumentácii nájdete **Note that this implementation is not synchronized**)).

Riešenie:

**synchronizovaná sekcia** – správa sa podobne ako synchronizovaná metóda, ale musí špecifikovať objekt, na ktorý sa synchronizácia vzťahuje.

```
while(true)
    synchronized(this) {
        if(semaphore.available()) {
            semaphore.acquire();
            semaphore.release();
        }
    }
```

# Nesynchronizovaný prístup

Iný, praktickejší príklad dátovej štruktúry, ku ktorej nesynchronizovane prístupujú (modifikujú ju) dve vlákna:

```
public class ArrayListNotSynchronized extends Thread {  
  
    ArrayList<Integer> al = new ArrayList<Integer>();           // štruktúra  
    int counter = 0;                                           // počítadlo  
  
    //not synchronized  
    public void add() {  
        System.out.println("add "+counter);  
        al.add(counter); counter++;           // pridaj prvok do štruktúry  
    }  
  
    //not synchronized  
    public void delete() {  
        if (al.indexOf(counter-1) != -1) {           // nachádza sa v štruktúre  
            System.out.println("delete "+(counter-1));  
            al.remove(counter-1); counter--;       // vyhod' zo štruktúry  
        }  
    }  
}
```

Súbor: [ArrayListNotSynchronized.java](#)

# Pokračovanie – dve vlákna

Vlákno t1 pridáva prvky, vlákno t2 maže zo štruktúry

```
public class ArrayListTester extends Thread {  
    boolean kind;  
    static ArrayListNotSynchronized a/ = new ArrayListNotSynchronized();  
    public ArrayListTester(boolean kind) { this.kind = kind; }  
  
    public void run() {  
        while (true) {  
            if (kind)  
                a/.add();  
            else  
                a/.delete();  
        }  
    }  
    public static void main(String[] args) {  
        new ArrayListTester(true).start();  
        new ArrayListTester(false).start();  
    }  
}
```

... a dostaneme (ked' zakomentujeme System.out.println):  
Exception in thread "Thread-2" [java.lang.IndexOutOfBoundsException](#):  
Index: 17435, Size: 17432  
at java.util.ArrayList.RangeCheck(Unknown Source)  
at java.util.ArrayList.remove(Unknown Source)  
at ArrayListNotSynchronized.delete([ArrayListNotSynchronized.java:12](#)):  
at ArrayListTester.run([ArrayListTester.java:12](#))

Súbor: [ArrayListTester.java](#)

# Synchronizovaná metóda vs. štruktúra

```
public class ArrayListNotSynchronized extends Thread {  
    ArrayList<Integer> al = new ArrayList<Integer>();  
    int counter = 0;  
    synchronized public void add() { al.add(counter); counter++; }  
    synchronized public void delete() {  
        if (al.indexOf(counter-1) != -1) { al.remove(counter-1); counter--; }  
    }  
}
```

```
public class ArrayListSynchronized extends Thread {  
    List al = Collections.synchronizedList(new ArrayList());  
    int counter = 0;  
    synchronized public void add() { al.add(counter); counter++; }  
    synchronized public void delete() {  
        if (al.indexOf(counter-1) != -1) { al.remove(counter-1); counter--; }  
    }  
}
```



# Monitor a čakacia listina

Každý objekt má **monitor**, ktorý obsahuje jediné vlákno v danom čase. Keď sa vstupuje do synchronizovanej sekcie/metódy viazanej na tento objekt, vlákno sa poznačí v monitore. Ak sa opäť pokúša vlákno dostať do synchronizovanej sekcie, monitor už obsahuje iné vlákno, preto je vstup do sekcie pozastavený, kým toto neopustí sekciu (a monitor sa uvoľní).

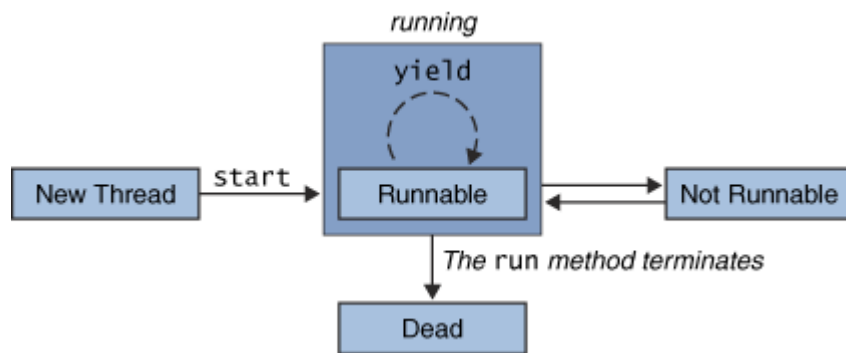
Každý objekt má **čakaciu listinu** – tá obsahuje vlákna uspané prostredníctvom volania objekt.**wait()**, ktoré čakajú, kým iné vlákno prebudí tento objekt prostredníctvom objekt.**notify()**.

```
public class Semaphore {  
    private int value;  
    public Semaphore(int val) {  
        value = val; }  
    public synchronized void release() {  
        ++value;  
        notify();    // this.notify();  
    }  
}
```

```
public synchronized void acquire() {  
    while (value == 0)  
        try {  
            wait();    // this.wait();  
        } catch (InterruptedException ie) { }  
    value--;  
}
```

**java.util.concurrent.Semaphor**

# Stavy vlákna



- new – nenašartovaný ešte,
- runnable – môže bežať, keď mu bude pridelený CPU,
- dead – keď skončí metóda run(), resp. po stop(),
- blocked – niečo mu bráni, aby bežal:
  - **sleep**(milliseconds) – počká daný čas, ak nie je interrupted...
  - **wait()**, resp. **wait**(milisec) čaká na správu **notify()** resp. **notifyAll()** ,
  - čaká na I/O,
  - pokúša sa zavolať **synchronized** metódu.

sleep vs. wait

keď vlákno volá wait(), výpočet je pozastavený, ale iné synchronizované metódy (tohto objektu) môžu byť volané

# Lopaty – cvičenie 8

## (pomocou `java.util.concurrent.Semaphore`)

```
Semaphore sem = new Semaphore(N, true);  
for (int i = 0; i < R; i++)  
    new Robotnik(i, sem).start();
```

```
class Robotnik extends Thread {  
    private int id; private Semaphore sem; private int odrobene = 0;  
  
    public void run() { // životný cyklus  
        while (odrobene < 10000) {  
            try { sleep(rnd.nextInt(1000)); } catch (InterruptedException e){} // spi  
            try { sem.acquire(); } catch (InterruptedException e1) {} // čaká lopatu  
            int cas = rnd.nextInt(1000);  
            odrobene += cas;  
            try { sleep(cas); } catch (InterruptedException e) { } // pracuje  
            sem.release();  
        }  
        System.out.println("Celkovy cas cakania "+id+ " "+celkovyCas);  
    }  
}
```

# Lopaty – cvičenie 8

## (bez semaforu)

```
lopaty = new LinkedList<Lopata>();           // vyrobíme si lopaty do zoznamu
for (int i = 0; i < N; i++) lopaty.add(new Lopata());
for (int i = 0; i < R; i++) {                // vyrobíme si robotníkov
    new Thread(Integer.toString(i)) {
        private Lopata moja;
        public void run() {
            while (true) {
                synchronized (lopaty) {      // čaká na lopatu
                    if (lopaty.size() > 0) {
                        moja = lopaty.removeFirst();
                    } else continue;
                }
                try { sleep(r.nextInt(1000)); } catch (Interrupt... e) { } //pracuje s moja
                synchronized (lopaty) {      // vráti na lopatu
                    lopaty.add(moja); moja=null;
                }
                try { sleep(r.nextInt(1000)); } catch (InterruptedException e){} //spí
            }
        }
    }.start();
}
```

? môže tu byť this ?



# Lopaty – cvičenie 8

## (pomocou wait-notify)

```
private int pocetLopatNaZemi = N;
```

```
public synchronized void zoberLopatu(){  
    if (pocetLopatNaZemi==0)  
        try { wait(); } catch (InterruptedException e) {}  
    pocetLopatNaZemi--;  
}
```

```
public synchronized void polozLopatu(){  
    notify(); pocetLopatNaZemi++;  
}
```

```
while (odrobene < 10000) {  
    try { sleep(rnd.nextInt(1000)); } catch (InterruptedException e) {} //spí  
    lopata.zoberLopatu(); // čaká  
    cas = rnd.nextInt(1000); // pracuje  
    odrobene += cas;  
    try { sleep(cas); } catch (InterruptedException e) { }  
    lopata.polozLopatu(); // položí  
}
```

```
System.out.println("Celkovy cas cakania "+id+ " "+celkovyCas);
```

# Lopaty – cvičenie 8

## (so skutočnými lopatami)

```
private int pocetLopatNaZemi = N;
```

```
public synchronized int zoberLopatu() throws InterruptedException {  
    if (pocetLopatNaZemi==0) wait();  
    pocetLopatNaZemi--;  
    return pocetLopatNaZemi;           // zlé riešenie, prečo ?  
}                                     // a čo je na ňom zlé ?
```

```
public synchronized void polozLopatu() throws InterruptedException {  
    notify(); pocetLopatNaZemi++;  
}
```

```
static ArrayList<Integer> lopaty = new ArrayList<Integer>(); // skutočné lopaty
```

```
public synchronized Integer zoberLopatu() throws InterruptedException {  
    if (lopaty.size() == 0) wait();  
    return lopaty.remove(0);           // zober prvú lopatu v zozname  
}
```

```
public synchronized void polozLopatu(Integer lopa) throws InterruptedException {  
    lopaty.add(lopa);                  // pridaj lopatu do zoznamu, na koniec ?, začiatok ?  
    notify();  
}
```

# Lopaty – cvičenie 8

## (s ascii lopatami)

```
final static char[] lopatyL = { '(', '[', '{', '<', '\\ ' }; // zober lopatu
final static char[] lopatyR = { ')', ']', '}', '>', '/ ' }; // polož lopatu
```

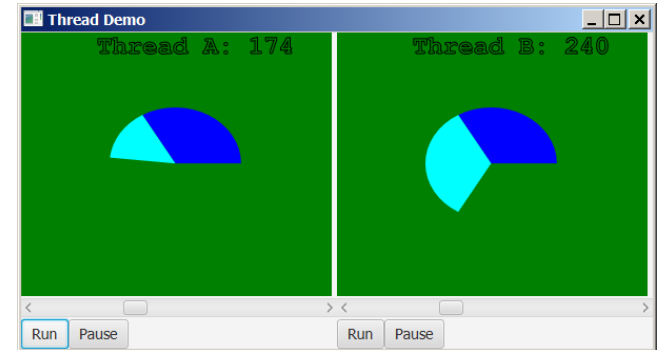
```
Integer lopata = cv.zoberLopatu();
System.out.println("Robotnik:" + id + " pracujem,zobral lopatu " + lopata );
System.out.print(RobotniciBezSemaforu.LopatyL[lopata]);
...
cv.položLopatu(lopata);
System.out.println("Robotnik:" + id + " položil lopatu " + lopata);
System.out.print(RobotniciBezSemaforu.LopatyR[lopata]);
```

```
([{}][{}]{[])([{}]{[]}[{}]{[])([{}]{[])({})({}{[]})}
([{}][[]]({})([{}][[]]{})({}){[]}([]){}{}{[]}{}){[]})
([{}]([]){})([{}]{[]}{})({}){[]}{[]}{[]}{[]}{[]}
```

# Thread demo

Simulujeme dve rovnako rýchlo bežiacie vlákna

- s možnosťou pozastavenia a opätovného spustenia,
- slajder ukazuje veľkosť kritickej oblasti,
- ale,
- **nesimulujeme žiaden monitor nad kritickou oblasťou, zatiaľ ...**



Štruktúra:

- ThreadPane je BorderPane a obsahuje panely:
  - TOP: GraphicCanvas typu Canvas, kreslí modrý pizza diagram na základe troch uhlov,
  - CENTER: Slider typu ScrollBar na nastavovanie veľkosti kritickej oblasti,
  - BOTTON: FlowPane obsahujúci gombíky Run a Pause

Ako pozastaviť animáciu:

- boolean suspended = false
- **aktívne čakanie** while (true) { ... if (suspended) sleep(chvíložku); ... }
- **pasívne čakanie**, pomocou wait & notify
- ~~CPU killer ... if (suspended) for (5000000x) Math.cos(...) ... ☹ ☹ ☹ ☹ ☹ ☹ ☹~~

Zdroj: pôvodná appletová verzia [http://www.doc.ic.ac.uk/~jnm/book/book\\_applets/concurrency.html](http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html)



# Neaktívne čakanie

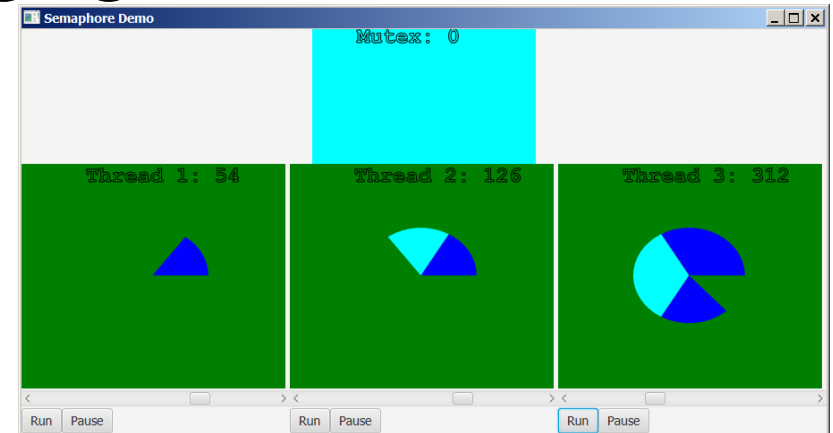
wait & notify

```
synchronized void waitIfSuspended() throws InterruptedException {  
    while (suspended)    // ak je vlákno suspended, tak sa zablokuje vo wait  
        wait();  
}  
  
void pauseThread() {    // reakcia na button Pause, treba suspendovať vlákno  
    if (!suspended) {  
        suspended = true;  
        display.setColor(Color.RED); // reakcia do GUI, premaľuj na RED  
    }  
}  
  
void restartThread() {    // reakcia na button Run, treba Odsuspendovať vlákno  
    if (suspended) {  
        suspended = false;  
        display.setColor(Color.GREEN); // reakcia do GUI, premaľuj na GREEN  
        synchronized (this) notify(); // tento notify odblokuje čakajúci wait  
    }  
}
```

Súbor: [ThreadDemo](#), [ThreadPanel.java](#)

# Semaphore loop

```
class SemaphoreLoop implements Runnable {  
    public void run() {  
        try {  
            while (true) {  
                while (!ThreadPanel.rotate())  
                    ;  
                semaphore.acquire();  
                while (ThreadPanel.rotate())  
                    ;  
                semaphore.release();  
            }  
        } catch (InterruptedException e) { }  
    }  
}
```



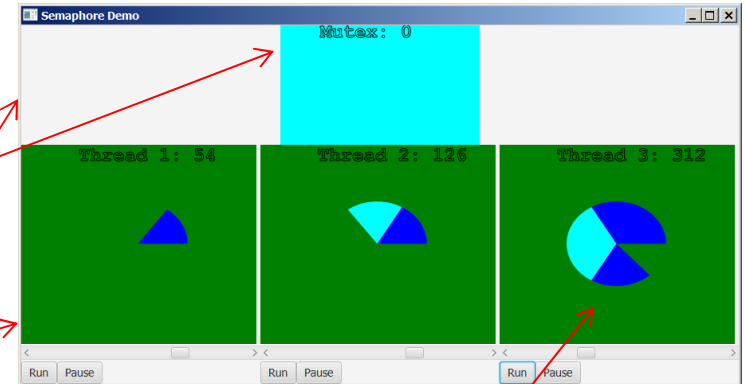
```
while (!ThreadPanel.rotate()) //false ak nie som v kritickej oblasti  
    ; // život mimo kritickej oblasti  
semaphore.acquire(); // vkroč do kritickej oblasti  
while (ThreadPanel.rotate()) // true ak som v kritickej oblasti  
    ; // som v kritickej oblasti  
semaphore.release(); // výstup z kritickej oblasti
```

Súbor: [SemaDemo.java](#)

Zdroj: pôvodná appletová verzia [http://www.doc.ic.ac.uk/~jnm/book/book\\_applets/concurrency.html](http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency.html)

# Semaphore main stage

```
public void start(Stage stage) throws Exception {  
    BorderPane bp = new BorderPane();  
    semaDisplay = new NumberCanvas("Mutex");  
    StackPane.setAlignment(semaDisplay, Pos.CENTER);  
    StackPane topPane = new StackPane(semaDisplay);  
    bp.setTop(topPane);  
    FlowPane pane = new FlowPane();  
    thread1 = new ThreadPanel("Thread 1", Color.BLUE, true);  
    thread2 = new ThreadPanel("Thread 2", Color.BLUE, true);  
    thread3 = new ThreadPanel("Thread 3", Color.BLUE, true);  
    Semaphore mutex = new DisplaySemaphore(semaDisplay, 1);  
    thread1.start(new SemaphoreLoop(mutex));  
    thread2.start(new SemaphoreLoop(mutex));  
    thread3.start(new SemaphoreLoop(mutex));  
    pane.getChildren().addAll(thread1, thread2, thread3);  
    bp.setBottom(pane);  
    Scene scene = new Scene(bp, 900, 450, Color.GREY);  
    stage.setScene(scene);  
    stage.setTitle("Semaphore Demo");  
    stage.show();  
}
```



# Ohraničený buffer

Príklad: producer-consumer:

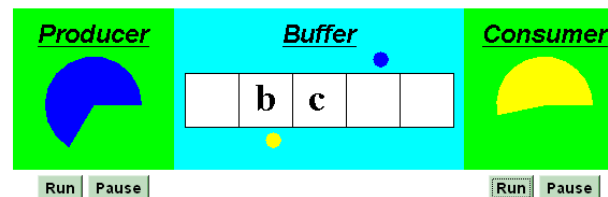
// zapíš objekt do buffra

```
public synchronized void put(Object o) throws InterruptedException {  
    while (count==size) wait(); // kým je buffer plný, čakaj...  
    buf[in] = o;  
    ++count;  
    in=(in+1) % size;  
    notify();  
}
```

// keď si zapísal, informuj čakajúceho

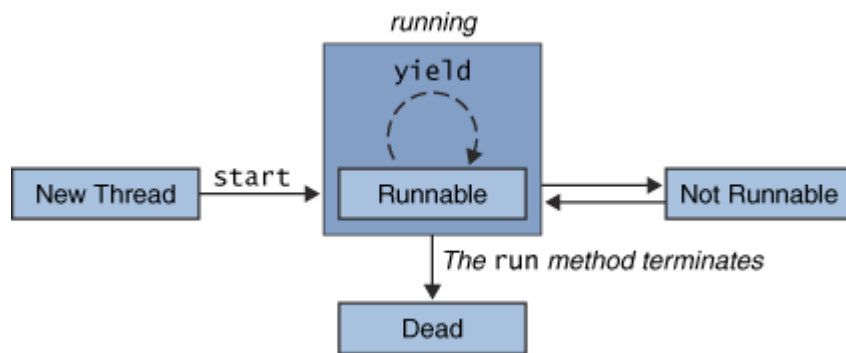
// vyber objekt do buffra

```
public synchronized Object get() throws InterruptedException {  
    while (count==0) wait(); // kým je buffer prázdny, čakaj...  
    Object o = buf[out];  
    buf[out]=null;  
    --count;  
    out=(out+1) % size;  
    notify();  
    return (o);  
}
```



// keď si vybral prvok, informuj ...

# Stavy vlákna



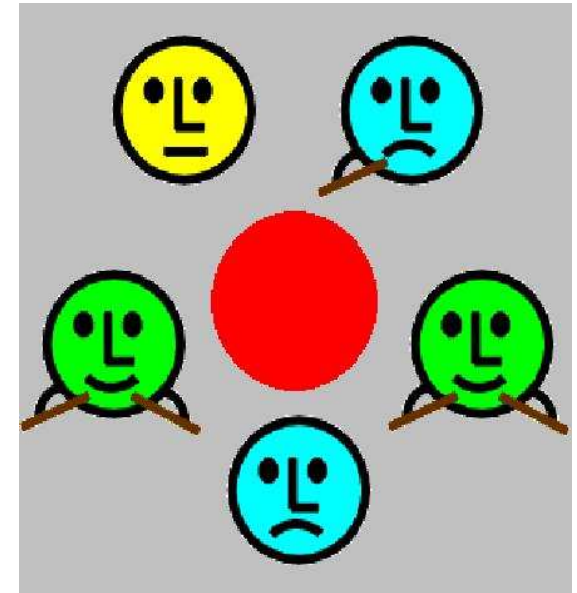
- new – nenašartovaný ešte,
- runnable – môže bežať, keď mu bude pridelený CPU,
- dead – keď skončí metóda run(), resp. po stop(),
- blocked – niečo mu bráni, aby bežal:
  - **sleep**(milliseconds) – počká daný čas, ak nie je interrupted...
  - **wait()**, resp. **wait**(milisec) čaká na správu **notify()** resp. **notifyAll()** ,
  - čaká na I/O,
  - pokúša sa zavolať **synchronized** metódu.

sleep vs. wait

keď vlákno volá wait(), výpočet je pozastavený, ale iné synchronizované metódy (tohto objektu) môžu byť volané

# Večerajúci filozofovia

```
class Fork {  
    private boolean taken=false;  
    private PhilCanvas display;  
    private int identity;  
  
    Fork(PhilCanvas disp, int id) {  
        display = disp; identity = id;}  
  
    synchronized void put() {  
        taken=false;  
        display.setFork(identity,taken);  
        notify();  
    }  
  
    synchronized void get() throws java.lang.InterruptedException {  
        while (taken) wait();  
        taken=true;  
        display.setFork(identity,taken);  
    }  
}
```

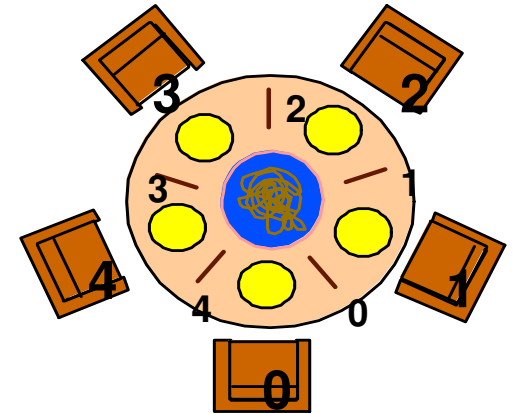


Zdroj: [http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency\\_applets/concurrency/diners/](http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency_applets/concurrency/diners/)

Súbor: [Fork.java](#)

# Večerajúci filozofovia

```
class Philosopher extends Thread {  
    private PhilCanvas view;  
    ....  
    public void run() {  
        try {  
            while (true) {  
                view.setPhil(identity,view.THINKING);  
                sleep(controller.sleepTime());  
                view.setPhil(identity,view.HUNGRY);  
                right.get();  
                view.setPhil(identity,view.GOTRIGHT);  
                sleep(500);  
                left.get();  
                view.setPhil(identity,view.EATING);  
                sleep(controller.eatTime());  
                right.put();  
                left.put();  
            }  
        } catch (java.lang.InterruptedException e){}  
    }  
}
```



*// thinking*

*// hungry*

*// gotright chopstick*

*// eating*

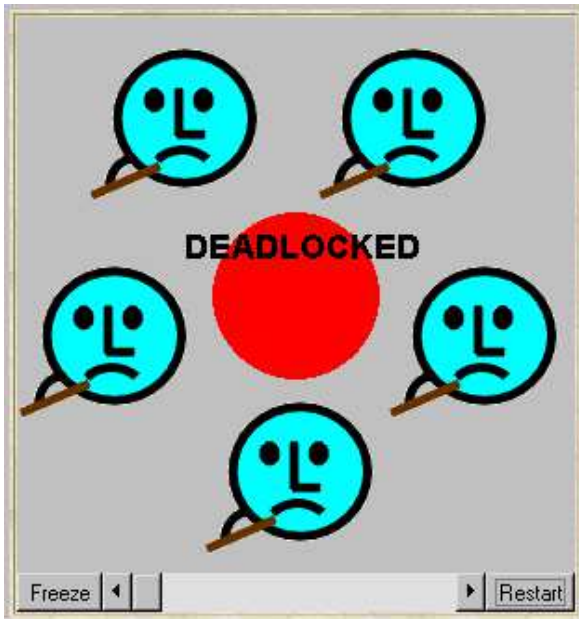
Zdroj: [http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency\\_applets/concurrency/diners/](http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency_applets/concurrency/diners/)

Súbor: **Philosopher.java**

[http://www.doc.ic.ac.uk/~jnm/book/book\\_applets/Diners.html](http://www.doc.ic.ac.uk/~jnm/book/book_applets/Diners.html)

# Večerajúci filozofovia

```
for (int i =0; i<N; ++i)
    fork[i] = new Fork(display,i);
for (int i =0; i<N; ++i){
    phil[i] = new Philosopher
        (this,i,fork[(i-1+N)%N],fork[i]);
    phil[i].start();
}
```



Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 0 eating  
Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 0 eating  
Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 0 eating  
Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 0 eating  
Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 0 eating  
Phil 0 thinking  
Phil 0 has Chopstick 0 Waiting for Chopstick 1  
Phil 1 thinking  
Phil 2 thinking  
Phil 3 thinking  
Phil 4 thinking  
Phil 1 has Chopstick 1 Waiting for Chopstick 2  
Phil 2 has Chopstick 2 Waiting for Chopstick 3  
Phil 3 has Chopstick 3 Waiting for Chopstick 4  
Phil 4 has Chopstick 4 Waiting for Chopstick 0



# Poučený večeraující filozof

```
class Philosopher extends Thread {
    private PhilCanvas view;
    ....
    public void run() {
        try {
            while (true) {
                view.setPhil(identity,view.THINKING);           // thinking
                sleep(controller.sleepTime());                  // hungry
                view.setPhil(identity,view.HUNGRY);
                if (identity%2 == 0) {
                    left.get();                                   // gotleft chopstick
                    view.setPhil(identity,view.GOTLEFT);
                } else {
                    right.get();                                  // gotright chopstick
                    view.setPhil(identity,view.GOTRIGHT);
                }
                sleep(500);
                if (identity%2 == 0)
                    right.get();                                  // eating
                else
                    left.get();                                   // eating
                view.setPhil(identity,view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (java.lang.InterruptedException e){}
    }
}
```

Zdroj: [http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency\\_applets/concurrency/diners/](http://www.cse.psu.edu/~catuscia/teaching/cg428/Concurrency_applets/concurrency/diners/)  
Súbor: [FixedPhilosopher.java](#)