KONKURENTNÉ PROGRAMOVANIE

Prvé cvičenie: úvod

Hodnotenie predmetu

- 2 projekty – jeden s viacvláknovým programom, druhý s programom využívajúcim vzdialené volania metód
- ... a to je všetko (Ďakujem za pozornosť)

Aký je rozdiel medzi paralelným a distribuovaným programom?

- Implementačný pohľad:
  - Paralelný: Program beží na v jednom počítači vo viacerých procesoch/vláknach
  - Distribuovaný: Program beží vo viacerých procesoch komunikujúcich po sieti
- Výkonnostný pohľad:
  - Paralelný: nezáleží mi na tom, na ktorom jadre/procesore/počítači spustím paralelnú metódu, lebo od všetkých mám rovnako rýchlu odozvu
  - Distribuovaný: musím rátať s latenciou (druhý počítač je ďaleko)

Viacvláknové programy

- Varenie kávy (sekvenčné)
  - Vyberiem hrnček a kávu
  - Nasypem kávu do hrnčeka
  - Pozriem koľko vody je v konvičke
    - Ak je jej málo, dolejem vodu
    - Ak je jej dosť, vodu nedolievam
  - Dám zovrieť vodu
  - Pozerám, kým sa zohreje (nič iné nerobím)
  - Keď zovrie, vypnem konvičku a zalejem kávu

Viacvláknové programy

- Varenie kávy (paralelné)
  - Vyberiem hrnček a kávu
  - Nasypem kávu do hrnčeka
  - Pozriem koľko vody je v konvičke
    - Ak je jej málo, dolejem vodu
    - Ak je jej dosť, vodu nedolievam
  - Dám zovrieť vodu
  - **Robím niečo iné, kým sa konvička nevypne**
  - Zalejem kávu

Viem súčasne variť kávu aj natierať chlieb s maslom!

Prečo robiť viacvláknové programy

- Máme veľa jadier v počítači, jednovláknový program nevyužije úplnú výpočtovú silu počítača
- Ak jedno vlákno čaká na I/O, iné môže bežať
- Vlákna komunikujú cez spoločnú pamäť svojho procesu (medzivýsledky netreba posielať hore-dole)
- GUI nevytuhne, keď sa deje nejaká dlhšia úloha

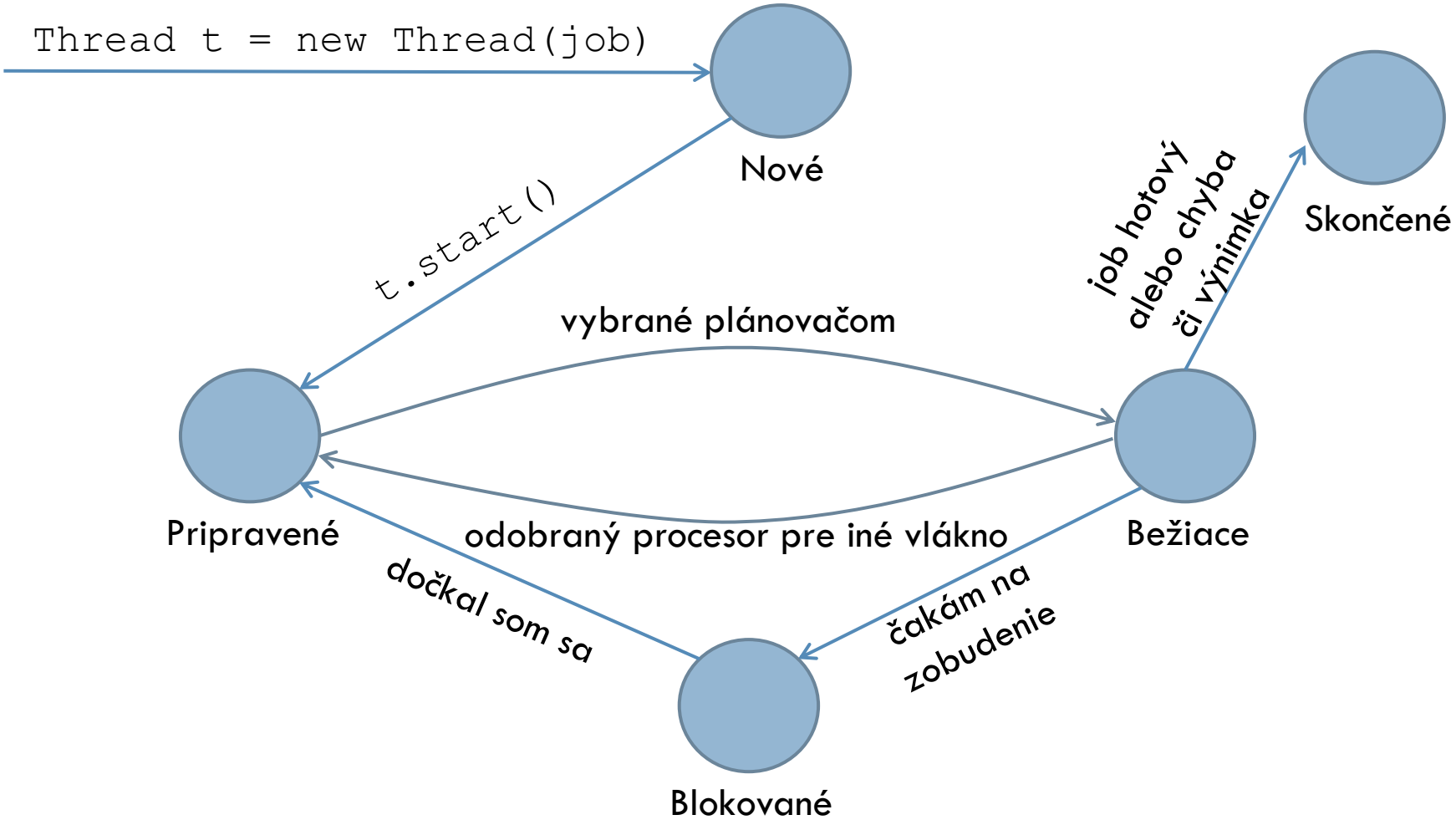
Nové vlákno: logicky, trieda Thread

- Vlákno potrebuje úlohu (job), ktorú by vykonalo
- Úloha pre vlákno implementuje rozhranie **`Runnable`**
- Spravíme inštanciu úlohy
  - `Runnable job = new MyJob();`
- Spravíme inštanciu vlákna, ktorému odovzdáme úlohu
  - `Thread thread = new Thread(job);`
- Spustíme vlákno
  - `thread.start();`
- Vlákno vykoná **samostatne** úlohu a môžeme ho zahodiť

Príklad 1

- Vytvorte program, ktorý v novom vlákne spustí nasledovnú úlohu: vypíše postupne čísla od 1 po 10
- Po spustení tohto nového vlákna, nech hlavný program vypíše postupne čísla od 1 po 10 a skončí

Stavy vlákien z pohľadu plánovača



Príklad 2 – 2 vlákna, 1 job

- Vytvorte úlohu: vypíše postupne meno vlákna, na ktorom beží a čísla od 1 po 10
- Túto úlohu zadajte dvom vláknam
- V hlavnom programe (hlavnom vlákne) tieto vlákna spustite

Príklad 3 - Spolupracujúce vlákna

- Vytvorte triedu Počítadlo, ktorá si bude v inštančnej premennej pamätať, koľkokrát bola zavolaná metóda getNext(), ktorá aj vráti jej hodnotu
- Vypisujte v dvoch vláknach 100000x hodnotu, ktorú vráti táto metóda

- Aké číslo vypíše posledný výpis?

Niekedy to nefunguje ☹

```
@NotThreadSafe
public class Počítadlo {
  private int value;

  /** vráti jedinečnú hodnotu */
  public int getNext() {
       return ++value;
  }
}
```

Niekedy to nefunguje ☹

Neatomická operácia:
(read-modify-write)

value → x
x+1 → y
value = y

```
@NotThreadSafe
public class Počítadlo {
  private int value;

  /** vráti jedinečnú hodnotu */
  public int getNext() {
      return ++value;
  }
}
```

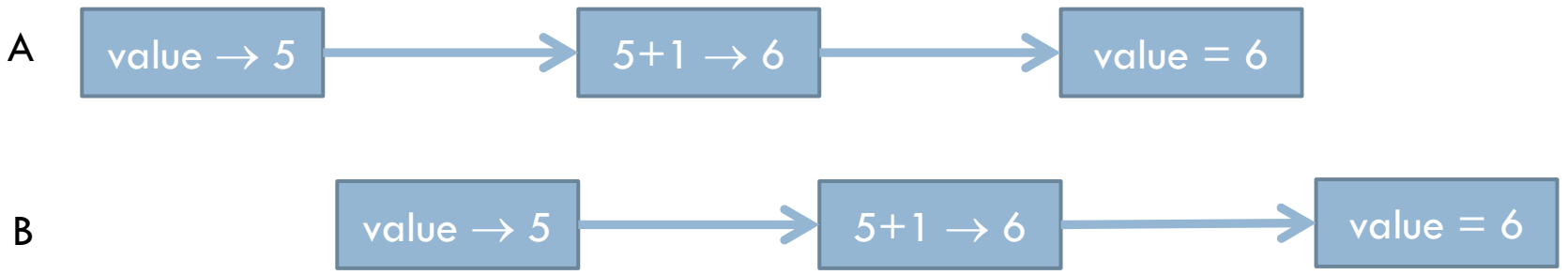
Niekedy to nefunguje ☹

Neatomická operácia:
(read-modify-write)
value → x
x+1 → y
value = y

```
@NotThreadSafe
public class Počítadlo {
  private int value;

  /** vráti jedinečnú hodnotu */
  public int getNext() {
    return ++value;
  }
}
```

Race condition:



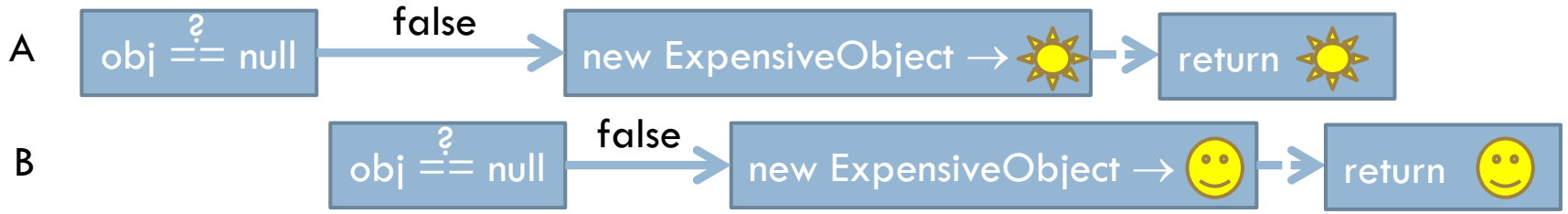
Z toho istého súdka

- Lenivé vytváranie singletonu:

Neatomická operácia:
(check-then-act)

```
@NotThreadSafe
public class LazyInitRace {
        private ExpensiveObject obj = null;
        public ExpensiveObject getInstance() {
                if (obj == null)
                        obj = new ExpensiveObject();
                return obj;
        }
}
```

Race condition:



Stráženie zámkom

- Ako zámok môžeme použiť ľubovoľný objekt alebo triedu
- Princíp:
  - Ak vlákno narazí na blok chránený zámkom X (kritickú sekciu) poprosí objekt X o jeho kľúč
    - Ak objekt X vlastní svoj kľúč, odovzdá ho vláknu a to môže vstúpiť do kritickej sekcie
      - Vlákno vykoná kritickú sekciu a potom vráti kľúč objektu X
    - Ak objekt X nevlastní svoj kľúč, vlákno sa uspí, pokiaľ objektu X niekto nevráti jeho kľúč
      - Ak už objekt X opätovne vlastní svoj kľúč, odovzdá ho tomu vláknu, ktoré najdlhšie čakalo na vstup do kritickej sekcie zamknutej zámkom X

Stráženie zámkom

Použi kľúč A

Objekt A

Použi kľúč B

Objekt B

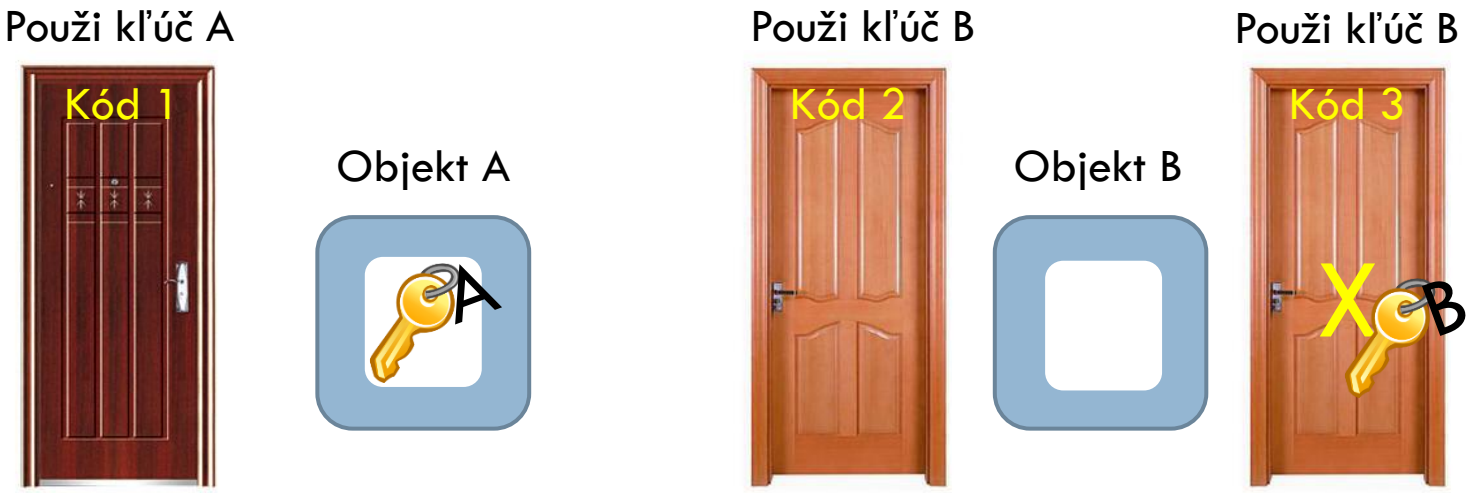
Použi kľúč B

Stráženie zámkom

Situácia:

- Vlákno X ide vykonávať kód 3 a narazí na dvere
- Poprosí objekt B o kľúč
- Objekt B odovzdá svoj kľúč vláknu X
- Vlákno X vstúpi do dverí „kód 3“

Stráženie zámkom



Situácia:

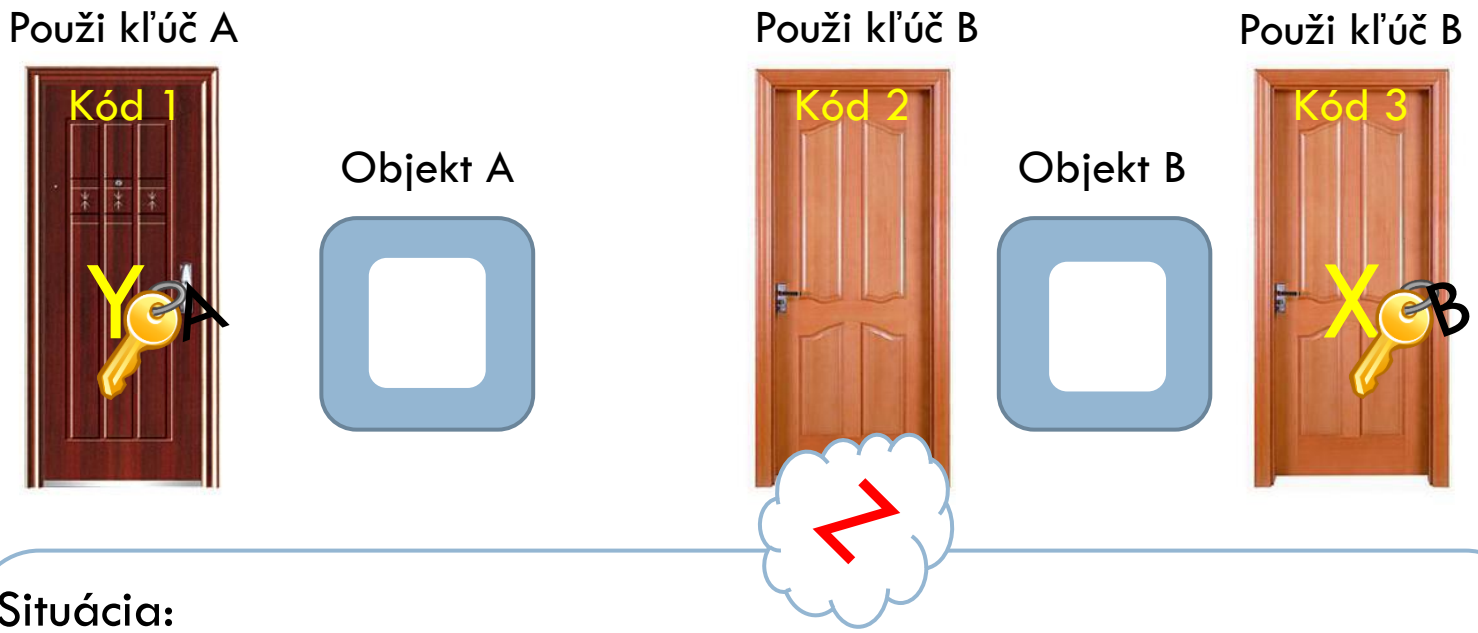
- Vlákno Y ide vykonávať kód 1 a narazí na dvere
- Poprosí objekt A o kľúč
- Objekt A odovzdá svoj kľúč vláknu Y
- Vlákno Y vstúpi do dverí „kód 1“

Stráženie zámkom

Situácia:

- Vlákno Z ide vykonávať kód 2 a narazí na dvere
- Poprosí objekt B o kľúč
- Objekt B kľúč nemá a tak vlákno Z musí spať pred dverami „kód 2“

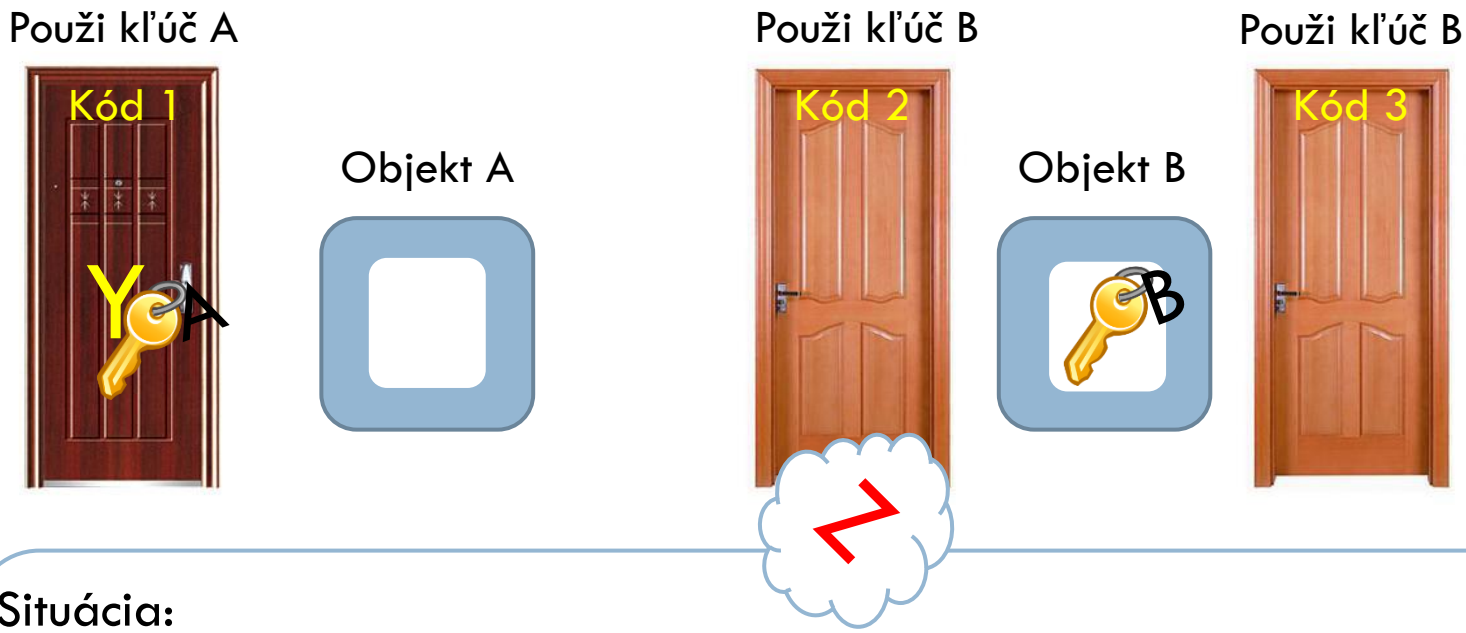
Stráženie zámkom



Situácia:

- Vlákno X skončí svoju robotu s vykonávaním kódu 3
- Vráti kľúč B objektu B

Stráženie zámkom



Situácia:

- Vlákno Z sa zobudí
- Objekt B odovzdá svoj kľúč vláknu Z
- Vlákno Z vstúpi do dverí „kód 2“

Stráženie zámkom

Situácia:

- Vlákno Z sa zobudí
- Objekt B odovzdá svoj kľúč vláknu Z
- Vlákno Z vstúpi do dverí „kód 2“

Stráženie zámkom

- Kritická sekcia **nie je** stena okolo kusu kódu, ktorý môže naraz vykonávať iba jedno vlákno
  - Nie je to WC, ktoré môže súčasne používať iba jeden človek
- Určujúci na vstup do kritickej sekcie sú:
  - aký zámok je na vstupných dverách a
  - prítomnosť kľúča od zámku u strážcu
- Viac kritických sekcií môže byť zamknutých tým istým zámkom

Stráženie: Atomická operácia

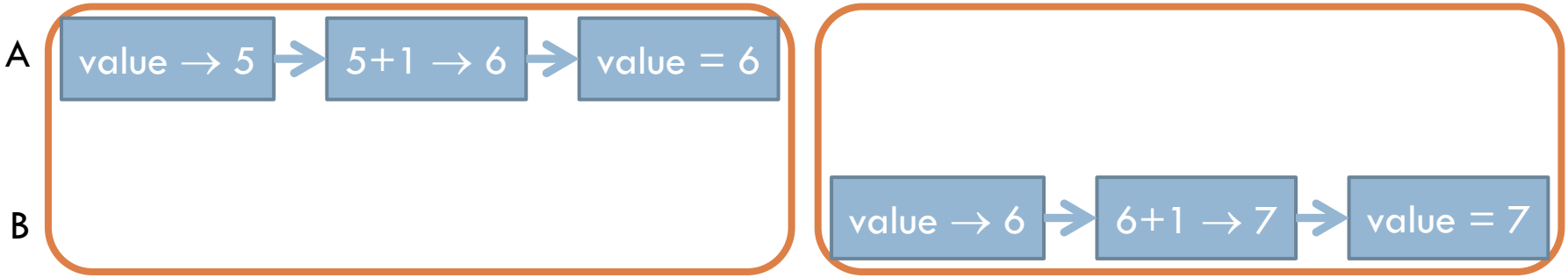
```
@ThreadSafe
public class Počítadlo {
  private int value;

  /** vráti jedninečnú hodnotu */
  public int getNext() {
    synchronized(this) {
      return ++value;
    }
  }
}
```

Strážená premenná zámkom `this`

zámok

Kritická sekcia = Atomická operácia!



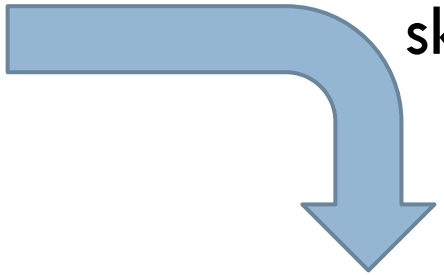
Príklady 4, 5 – čo bude atomické?

- Zmeňte príklad 3 tak, aby vo výpise neboli čísla poprehadzované, ale vypisované postupne od 1 po 200000

- Zmeňte príklad 3 tak, aby jedno vlákno spravilo výpis 100000 čísiel a až potom spravilo tento výpis druhé vlákno a to iba zmenou rozsahu atomickosti operácií

Synchronized metódy=skratky

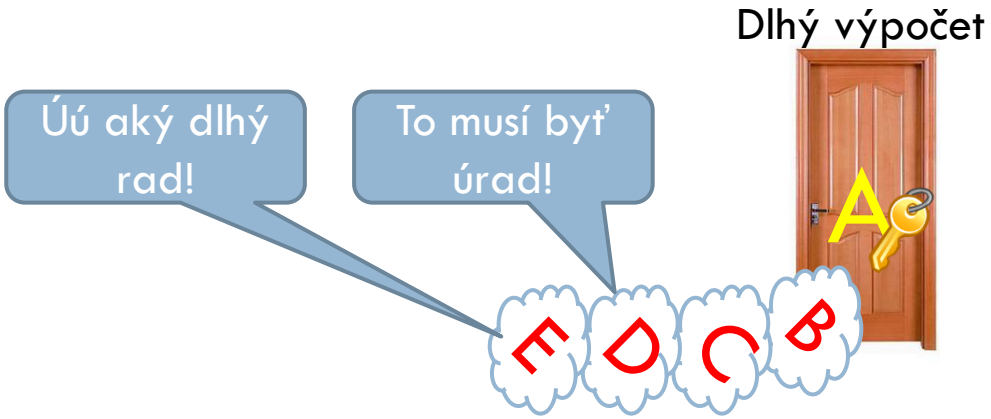
```
public class A {
…
   public int metóda() {
      synchronized(this) {
        dôležitáMetóda();
        kritickáMetóda();
      }
   }
…
}
```




```
public class A {
…
   public synchronized int metóda() {
      dôležitáMetóda();
      kritickáMetóda();
   }
…
}
```

Zamykať treba s mierou

- Ak sú príliš veľké kusy kódu zamknuté do kritických sekcií, môže to spôsobiť výrazné spomalenie výpočtu
- Kým nemusíte, nezamykajte
  - Dlhé výpočty
  - Blokované operácie (I/O na disk, z klávesnice, zo siete)



Zámok smrti (deadlock)

Situácia:

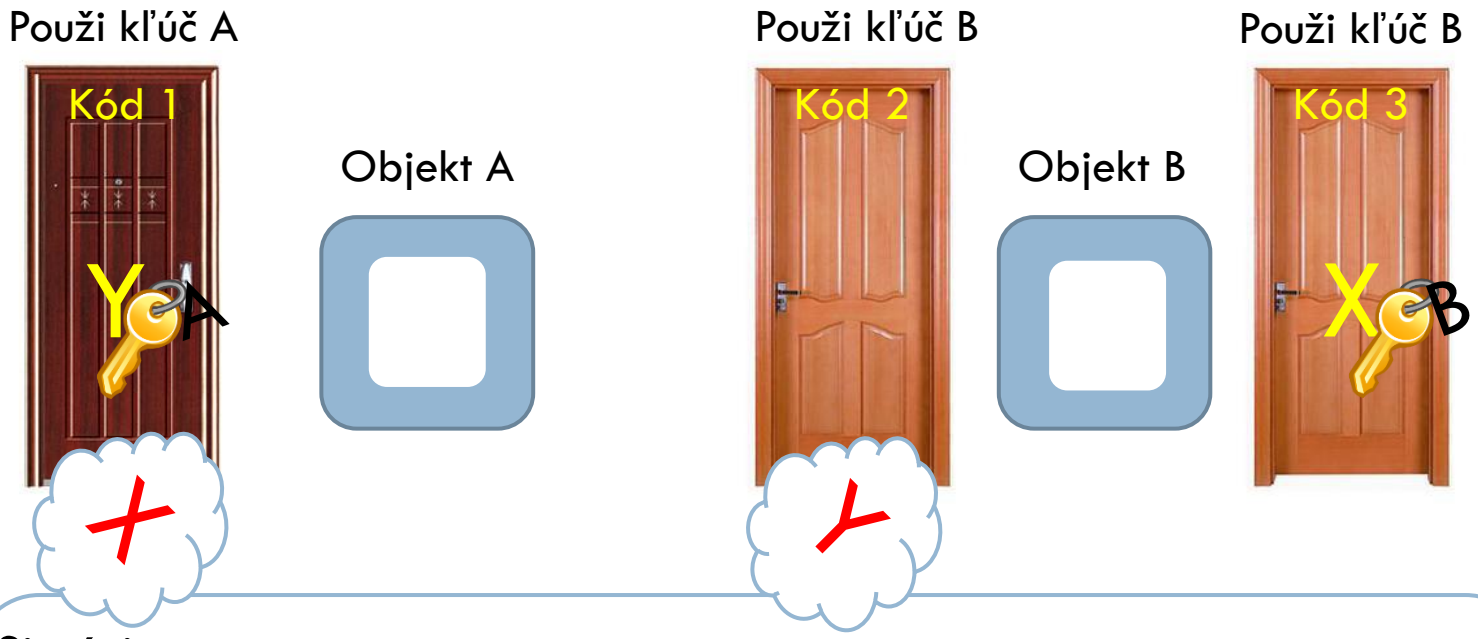
- Vlákno X vykonáva kód 3, z ktorého ide volať kód 1
- Poprosí objekt A o kľúč
- Objekt A kľúč nemá a tak vlákno X musí spať pred dverami „kód 1“

Zámok smrti (deadlock)

Situácia:

- Vlákno Y vykonáva kód 1, z ktorého ide volať kód 2
- Poprosí objekt B o kľúč
- Objekt B kľúč nemá a tak vlákno Y musí spať pred dverami „kód 2“

Zámok smrti (deadlock)



Situácia:

- Vlákno Y vykonáva kód 1, z ktorého ide volať kód 2
- Poprosí objekt B o kľúč
- Objekt B kľúč nemá a tak vlákno Y musí spať pred dverami „kód 2“
- DEADLOCK!

Prečo sa **naučiť** robiť viacvláknové programy

- Ak mám iba jednu úlohu, alebo veľa rovnakých tak ich proste beriem za radom, až kým ich nevyčerpám (alebo seba)
  - Jednovláknové programy sa programujú ľahšie
- Ak mám veľa rôznych úloh, prepínam medzi nimi začínam pociťovať potrebu organizácie času
  - Viacvláknové programy treba dobre premyslieť, dobre zdokumentovať a udržiavať s rozumom
- Vlákna sú (nielen) v Jave vždy: garbage collection, finalization, Swing, Timer, servlety, RMI

Viacvláknové problémy

- Pri zlom návrhu môže kód bezchybne fungovať, prejsť testami a nasadiť sa u zákazníka a funguje aj uňho až na 0,1% „záhadných zlyhaní“
  - Týmto problémom môže trpieť aj jednovláknový program, ale to je závislé iba od vstupu
    - vieme ľahko zopakovať stav zlyhania
  - Pri viacvláknových programoch môže byť problém
    - vstup,
    - náhoda poradia vykonania príkazov v rôznych vláknach
      - race condition, deadlock, starvation, livelock,...

Ďakujem za pozornosť

- Web predmetu:
  - https://kopr.ics.upjs.sk/