Java

- Předpokládá se znalost práce s vlákny a Java concurrency
  - Viz KIV/PGS
  - V PPR jenom to, co nebylo jinde
- JVM bylo původně navrženo pro Javu, dnes lze však i z jiných jazyků produkovat bytecode JVM
  - Nicméně, další text se bude primárně zaobírat konstrukcemi Javy

Synchronizace v Javě

- Plánovač OS vs. plánovač JVM
  - Java je vysokoúrovňový prostředek
  - JVM může buď 1:1 namapovat svoje vlákna na vlákna poskytovaná OS, anebo může, teoreticky, zvolit i jiný přístup, kdy jedno vlákno poskytované OS může vykonávat několik vláken JVM
    - Uživatel JVM je od toho odstíněn
    - Není jenom jedno JVM
- V Javě (bez java.util.concurrency) se vše děje s pomocí monitoru (intrinsic lock/monitor lock)
- monitory jsou reentrantní – vlákno má povolen opakovaný exkluzivní přístup do té samé kritické sekce, aniž by se ho muselo nejprve vzdát

- jakékoliv jiné synchronizační primitiva musí být realizována s pomocí monitorů
  - monitory se v bytecodu vytvářejí pomocí instrukcí monitorenter a monitorexit, nebo pomocí flagu acc_synchronized
  - teprve až v nativním kódu lze dále optimalizovat

- blok kódu chráněný javovským monitorem je zabezpečená kritická sekce
  - EnterCriticalSection je vstup do bloku
    - monitorenter
  - LeaveCriticalSection je opuštění bloku
    - monitorexit

- aplikačně specifický monitor je třída, která používá klíčové slovo Javy synchronized
- synchronized u bloku kódu
  - klíčové slovo používané ke konstrukci bloku v těle metody synchronized (objectReferenceOrThis)

- volatile – takto deklarovanou proměnnou nemá JVM cachovat, zapisuje do ní několik vláken a pokud by byla uchovávána například v registru, který by byl součástí kontextu threadu, pracovalo by se s neaktuální hodnotou
  - tj. pokud poběží několik threadů podle jednoho kódu, pak by každý thread mohl danou proměnnou cachovat v registru svého procesoru
  - tím pádem by každé vlákno vidělo svoji kopii proměnné, jejíž hodnota by se tak mohla lišit mezi jednotlivými vlákny
    - => taková proměnná se deklaruje jako volatile, aby JVM nevytvářelo právě popsaný scénář

Možnost deadlocku v Javě

- buď díky slabinám v návrhu multithreadingu Javy
  - některé metody třídy Thread jsou proto deprecated
  - java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html
  - Stop
    - Ukončí běh vlákna a způsobí uvolnění všech zámků, které v té době vlastnilo
    - Pokud jakákoliv data, která vlákno zrovna zpracovávalo, jsou v nekonzistentním stavu, zůstanou v něm
    - Možné následky použití dat v nekonzistentním stavu:
      - Deadlock
      - Livelock
      - Chybná činnost
      - Abnormal termination
      - Propagace chyby dál – efekt laviny
  - Vyjímka ThreadDeath
    - Musela by být zpracovávána všude
    - Její zpracování by muselo být vnořené – vyjímka ve vyjímce
    - Neúměrně rychle roste TCO (Total Costs of Ownership)

- Místo stop()
  - Špatně

```
private Thread blinker;

public void start() {
  blinker = new Thread(this);
  blinker.start();
}

public void stop() {
  blinker.stop();  // UNSAFE!
}

 public void run() {
   Thread thisThread = Thread.currentThread();
   while (true) {
     try {
       thisThread.sleep(interval);
     } catch (InterruptedException e){ }
       repaint();
     }
 }
```

  - Správně

```
private volatile Thread blinker;

public void stop() { blinker = null; }

public void run() {
  Thread thisThread = Thread.currentThread();
  while (blinker == thisThread) {
    try {
      thisThread.sleep(interval);
    } catch (InterruptedException e){}
    repaint();
  }
}
```

        - V závislosti na délce intervalu se zbytečně konzumuje čas procesoru => delší interval a použití metody interrupt
            - V některých případech, např. I/O operace, sokety, interrupt nemusí fungovat správně
            - Řešením je pak např. uzavření soketu, což ale vždy není, co potřebujeme
    - Suspend & Resume
        - Pokud vlákno vlastní zámky a je uspáno pomocí suspend, žádné jiné vlákno nemůže tyto zámky získat
        - Pokud se jakékoliv vlákno pokusí získat některý ze zámků vlastněných spícím procesem, předtím než se zavolá jeho resume, nastane deadlock vlákna – aka frozen process
            - Jediné řešení je, že vlastníka zámků někdo vzbudí – je-li ovšem kdo
        - Řešením je používat wait a notify

- Špatným programovým kódem
    - čekáním se na podmínku, která nikdy nenastane a program skončí v nekonečné smyčce není deadlock, ale livelock
        - livelock – thread uváznul, ale stále se vykonává nějaký kód (i tak někdy bývá označován jako deadlock)
        - deadlock – thread uváznul, nevykonává se, čeká na podmínku, která nikdy nenastane

- použijeme-li dva různé objekty pro poskytnutí zámku monitorům a zavoláme-li je ze dvou různých vláken, docílíme deadlocku

```
public class LockMeUp implements Runnable{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void liveLock() {
       while (1) {
         if ((c1<c2) & (c1>c2)) break;
          //test překladače, jestli odhalí
          //nesplnitelnou podmínku
         c1++;
         c2--;
       }
    }

    public void run() {
        synchronized(lock2) {
          synchronized(lock1) {
            //Nikdy se sem nedostane
            c1++;
          }
        }
    }

    public void deadLock() {
      Thread killer = new Thread(this);
        synchronized(lock1) {
           killer.start();
           suspend(); // neuvolnit zámek,
                      // aby nastal deadlock
        }
    }
}
```

Překlad synchronizovaného kódu do bytecode

- Mějme implementovat celočíselný semafor prostředky Javy

```
class Semaphore {
   private int count;
   public Semaphore(int n) {
      this.count = n;
   }

   public synchronized void acquire() {
      while(count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
            //keep trying
         }
      }
      count--;
   }

   public synchronized void release() {
      count++;
      notify(); //alert a thread that's
                //blocking on this semaphore
   }
}
```

http://www.ibm.com/developerworks/library/j-thread.html

- metoda release semaforu implementovaného v Javě
  - pouze klíčové slovo synchronized jsme přesunuli do kódu, aby vynikla celá idea

```
  public void release() {
    synchronized (this) {
      count++;
      notify(); }
}
```

- Všimněte si v následujícím bytecodu instrukcí monitorenter a monitorexit

```
public void release();
  Code:
   0:   aload_0
   1:   dup
   2:   astore_1
   3:   monitorenter
   4:   aload_0
   5:   dup
   6:   getfield#2; //Field count:I
   9:   iconst_1
   10:  iadd
   11:  putfield#2; //Field count:I
   14:  aload_0
   15:  invokevirtual    #5; //Method
java/lang/Object.notify:()V
   18:  aload_1
   19:  monitorexit
   20:  goto28
   23:  astore_2
   24:  aload_1
   25:  monitorexit
   26:  aload_2
   27:  athrow
   28:  return
```

- Instrukce monitorenter a monitorexit odpovídají funkcím EnterCriticalSection a LeaveCriticalSection
  - Které jsou v operačním systému implementovány pomocí semaforu
  - Takže v Javě bude semafor implementován pomocí semaforu?!
    - Ano

- Pokud by metoda měla příznak synchronized, v bytecode se instrukce monitorenter a monitorexit neobjeví
  - Místo toho bude nastaven příslušný flag, acc_synchronized, dané metody, takže monitorenter a monitorexit se provedou

- Důsledek?
  - Java Concurrency Utilities, tj. java.util.concurrent

  - Oficiální motivace
    - Se základními prostředky Javy se stále dá dosáhnout deadlocku
    - Základní prostředky synchronizace v Javě vedou k malé škálovatelnosti výkonu
    - Programátoři nemají k ruce vysokoúrovňové prostředky jako právě např. semafor

  - Motivace po průchodu „selským rozumem“
    - Monitor v Javě je tak high-level, že oproti prostému použití API daného OS nemůže být jiný než pomalý

- Takže bylo třeba zabalit do Java tříd synchronizační primitiva přímo od OS, či atomické operace poskytované procesorem

- A teprve nad tímhle se dá postavit skutečně něco high-level tak, aby to běželo rozumně rychle

- Kritické části runtime knihoven nejsou psané v Javě
  - Buď se nakonec volá kód napsaný v C++
  - Anebo dokonce ručně optimalizovaný strojový kód
  - Např. metoda compareAndSwap třídy java.util.concurrency.atomic.AtomicIntege r volá metodu compareAndSet třídy sun.misc.Unsafe
    - přičemž tato metoda je final a nativní – proto není v třídě AtomicInteger deklarovaná jako synchronized

    - ve finále se to na x86 dá zredukovat až na jedinou instrukci xchg
    - přičemž C++ překladače pro std::atomic to udělají rovnou při překladu

    - JVM to sice nakonec může udělat také, ale může to udělat teprve až za runtime programu
    - Za cenu další paměti a výpočetního výkonu spotřebovaného na tzv. Escape analýzu

Escape analýza

- Java z principu nebude nikdy tak rychlá jako C/C++ staticky zkompilovaný kód (a už vůbec ne na x86)
  - Pokud to tak v některém benechmarku vychází, pak:
    A) buď kompilační jednotka mezi židlí a klávesnicí nepoužila všechny dostupné optimalizace
    B) anebo se jedná o dva příliš různé programy

- Změna alokace z haldy na změnu alokace na zásobníku
  - Např. C++ umožňuje alokovat jednou jedinou instrukcí objekt na zásobníku a jednou jedinou instrukcí ho zase uvolní z paměti
  - Jsou-li pro to splněny podmínky, použití takové techniky je mnohonásobně rychlejší než alokace objektu z haldy
    - Proto se to JVM snaží napodobit, tj. snaží se detekovat, kdy jsou vytvářeny právě takové objekty
      - Ovšem na rozdíl od C++ programátora to nikdy neví jistě a proto věnuje část strojového času na to, aby se to pokusila zjistit za běhu aplikace (což ji zdržuje)

- Eliminace synchronizace
  - Např. 1B, či integer do velikosti strojového slova zarovnaný v paměti na adresu beze zbytku dělitelnou velikostí strojového slova, je na x86 garantován, že bude přečten i zapsán atomicky i bez prefixu lock

- V Javě by ovšem přístup k takové proměnné třídy musel být chráněn pomocí synchronized způsobující zbytečně velkou režii
- A tak se JVM opět snaží detekovat takové případy, aby za běhu nedocházelo ke zbytečnému volání monitorenter a monitorexit

Výhody multithreadingu Javy

- poskytuje vysokoúrovňový prostředek, jakým je monitor
- monitor či java.util.concurrency obvykle stačí
- programátor si nemusí dělat starosti se vstupem a opuštěním kritické sekce
- (diskutabilní výhoda) JVM se snaží myslet za programátora

Nevýhody multithreadingu Javy

- další synchronizační prostředky musí být vytvořeny s pomocí monitoru
- stále je možné docílit deadlocku, ale kde není? ;-)
- garbage collector – běží rychle, dokud je k dispozici několikanásobek požadované paměti
  - rychlost odezvy gc se zhoršuje s počtem alokovaných objektů
  - málo paměti znamená časté procházení objektů, jestli se dají uvolnit
  - a tohle všechno trvá a na nedeterministicky dlouho pozastavuje spuštěná vlákna
  - na desktopu je RAMky třeba i dost, ale v mobilu?

Safe-Point

- Místo kde lze zastavit běh vlákna
  - Stop-the-World garbage collector
  - Vkládání bytecodu přeloženého do nativního kódu
  - Profilování – viz slidy 2c
- Vkládá ho dynamicky JVM podle potřeby
  - Příliš mnoho safe-pointů je sice ideální, ale pak už by to bylo „zdržení větší než jen malé“
  - Takže se dává jenom někam
    - Aneb se může stát, že např. z n-vláken se jich n-1 pozastaví a čeká se na n-té vlákno až dokončí smyčku – opět to zdržuje
  - Ovšem malá metoda nakonec může přerůst mez velikosti kódu a nemusí být inlinována - zdržení
- Je třeba, aby vykonávání safe-pointu nezdržovalo (alespoň v situaci, kdy není třeba zastavit vlákno)
- Safe-point lze např. na x86-64 implementovat instrukcí test rax, [memory]
  - Instrukce nic nemění, kromě toho, že nastaví některé bity v registru rflags
  - [memory] může být v cache procesoru, takže to normálně moc nezdržuje
    - Respektive záleží, kam to JVM umístí – např. ve smyčce s násobením dvou vektorů by to zdržovalo celkem dost
  - Jakmile je třeba vlákno zastavit, JVM odmapuje dedikovanou stránku [memory] z paměti, a vlákno se zastaví na vyjímce PageFault
    - Takže je to svým způsobem sice elegantní, ale na druhou stranu také drahé, když se zastaví
    - Odmapování viz např. unmap_mapping_range v Linuxu

Spurious Wakeup

- Ačkoliv má být JVM vysokoúrovňovou abstrakcí od OS, např. Spurious Wakeup je takový scénář, kdy před OS „nelze utéct“
- Jednodušš řečeno jde o to, že spící vlákno se může probudit, aniž by byla splněna programátorem zapsaná podmínka pro jeho probuzení
  - Ještě jednodušeji: když zavoláte wait(), tak se vlákno může probudit, aniž by někdo zavolal notify()

- Aneb jak napsat bariéru špatně a správně

```
class Bariera {
    private int pocetVlaken;
    private int citac;
    private boolean muzeVstat;

Bariera (int pocetVlaken) {
    this.citac = 0;
    this.muzeVstat = false;
    this.pocetVlaken = pocetVlaken;
}

public synchronized void synchronizuj_spatne()
            throws InterruptedException {

    citac++;

    if (citac < pocetVlaken) { wait(); }
     else { citac = 0; notifyAll();}

    //Když se jedno vlákno vzbudí předčasně, tak
    //také bariéru opustí předčasně => je třeba
    //zavést další podmínky pro čekání vlákna.
}
```

```
public synchronized void synchronizuj_dobre()
                throws InterruptedException {

    while (muzeVstat == true) {wait();}
    citac++;

    if (citac == pocetVlaken) {
        muzeVstat = true;
        notifyAll();
    }

    while (muzeVstat == false) {wait();}
    citac--;

    if (citac == 0) {
        muzeVstat = false;
        notifyAll();
    }
}
} //class Bariera
```

- Uvedený kód demonstruje SpuriousWakeup
  - Jinak se dá použí třída CyclicBarrier

- Linux
  - Mapuje-li JVM svá vlákna 1:1 na vlákna pthreads, pak wait odpovídá funkci pthread_cond_wait(), která je implementovaná pomocí futexu – tj. pomocí systémového volání
  - Jenomže, blokující systémové volání je „násilně“ přerušeno, EINTR, obdrží-li proces signál
  - pthread_cond_wait nelze restartovat, protože za tu dobu, co byl proces „vzhůru“, mohl nastat skutečný wakeup a ten by se mohl restartování „promeškat“
    - => spuriouswakeup, a ať si to hlídá programátor, protože jinak by to bylo příliš komplikované na úrovni jádra

- Futex == Fast User Space Mutex
  - Fronta čekajících procesů je na úrovni jádra
  - Ale počítadlo je v uživatelském adresovém prostoru
    - Vlákna se jeho použitím pokoušejí vyhnout drahým systémovým voláním
    - Ve Windows se takhle chovají slim locks a kritická sekce – viz spincount

- Vedle Spurious Wakeup ještě existuje tzv. Stolen Wakeup
  - Jiné vlákno se spustí dříve než vzbuzené vlákno