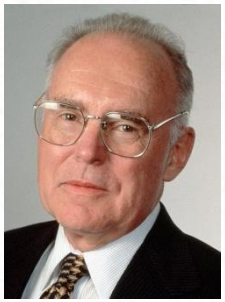
1. prednáška (14.2.2011)

Úvod do PDS

Prečo PDS? (1)



- Moorov zákon (pozorovanie 1958-1965):
  - zložitosť čipov sa zdvojnásobí každé 2 roky pri zachovaní ceny
  - pozorovanie platí do dnes
  - G.E. Moore – spoluzakladateľ Intelu
- Fyzikálne limity súčasných technológií
  - napr. rýchlosť prenosu informácie je obmedzená rýchlosťou svetla, energetické nároky rastúce s výkonom procesora

Prečo PDS? (2)

- **Moorov zákon** vs. **fyzikálne limity**
- Úlohy, ktoré dokážu „použiť" všetok **disponibilný výpočtový výkon** (simulácie, vedecké výpočty, grafika, ...)
- Potreba spracovávať čoraz **viac údajov**
  - odhaduje sa 10^9 GB vyprodukovaných dát ročne

Riešenie:

Paralelizmus = využitie viacerých súčasne pracujúcich „počítačov" na riešenie úlohy

PDS dnes

- Viacjadrové CPU (dnes 2 jadrá sú minimum)
- GPU
  - GeForce GTX 470, marec 2010, 448 jadier
- Google Server Farm: 450000 počítačov
- BOINC
  - Berkeley Open Infrastructure for Network Computing, 460157 aktívnych CPU v Folding@home
- BitTorrent, distribuované databázy
- Cloud computing

Kde je problém?

Veľa procesorov = rýchlejší výsledok?

- Príklady:
  - 1 robotník prehádže kopu piesku za 10 hodín
    - za aký čas prehádže kopu piesku 10 robotníkov?
    - za aký čas prehádže kopu piesku 1000 robotníkov?
  - 1 žena vynosí dieťa za 9 mesiacov
    - za aký čas vynosí 1 dieťa 9 žien?

O čo ide?

- **Nájsť paralelizmus**
  - ako nájsť časti, ktoré vieme riešiť paralelne?
- **Vybalancovať rozdelenie práce** (workload)
  - ako rozdeliť jednotlivé podúlohy procesorom, tak aby to celkovo bolo čo najefektívnejšie?
- **Minimalizovať/manažovať komunikáciu**
  - ako minimalizovať komunikáciu či synchronizáciu medzi procesmi?
- **Implementovať riešenie**...

O čom bude predmet PDS? (1)

- V súčasnosti existuje veľa rôznych prístupov k riešeniu paralelných a distribuovaných výpočtov:
  - rôzne HW architektúry a rôzna podpora HW
  - rôzne systémy a frameworky: Hadoop, Pregel, MPI implementácie, OpenMP, CUDA, OpenCL, BOINC, ...
- V predmete PDS chceme:
  - ukázať niektoré **základné problémy** a ich možné riešenia abstrahujúc od technických a implementačných detailov (zdanlivo teoretické veci)
  - praktické ukážky počas tutoriálov (2-3 za semester)

O čom bude predmet PDS? (2)

- Oblasť paralelných a distribuovaných výpočtov je veľmi **rozsiahlou podoblasťou informatiky**
  - aktívny výskum (základný aj aplikovaný)
  - početné aplikácie v praxi
  - paradigma, ktorá nahradí mnohé súčasne používané prístupy (?)

Prečo je paradigma PDS náročná?

- Pre ľudí je **sekvenčné** rozmýšľanie **prirodzené**
  - paralelné programovanie vyžaduje oveľa väčšie pochopenie podstaty problému
  - manažérsky prístup – treba manažovať prácu procesorov
- Treba myslieť na **širšom kontexte** a na **viac prípadov** – práca procesorov sa môže ovplyvňovať, mnohokrát aj „nepredvídateľne"
  - pri sekvenčnom programovaní vieme na základe aktuálneho stavu programu povedať, čo sa stane

Základné rozdelenie PDS

- **Konkurentné výpočty** (programovanie)
- **Paralelné výpočty** (programovanie)
  - so zdieľanou pamäťou (multiprocesory)
  - s distribuovanou pamäťou (multipočítače)
- **Distribuované výpočty** (programovanie)

- jednotlivé „oblasti" sa navzájom prelínajú
  - je ťažko robiť jedno bez znalosti iného

Proces a procesor

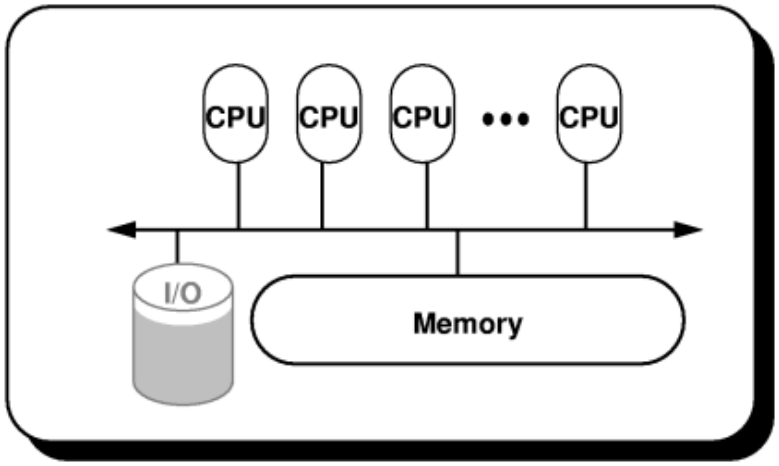
- **Proces**
  - sekvencia inštrukcií, ktorá môže byť vykonaná jedným fyzickým procesorom
  - „podprocedúra", softvérový prvok
  - z pohľadu PDS aj vlákna sú procesy
  - pozor, nezamieňať si proces v OS s procesom ako základným stavebným a konceptuálnym prvkom paralelných výpočtov
- **Procesor**
  - fyzické zariadenie vykonávajúce proces

Konkurentné programovanie

- Oblasť, ktorej sa žiaden programátor nevyhne
  - Vlákna a potreba ich kooperácie je nevyhnutná v každom rozsiahlejšom programe
- Základný problém:
  - **koordinácia procesov** (z pohľadu OS vlákien i procesov) pri pristupovaní k zdieľaným zdrojom (napr. zdieľaná premenná – potenciálne každá premenná v heape)

Paralelné výpočty so zdieľanou pamäťou

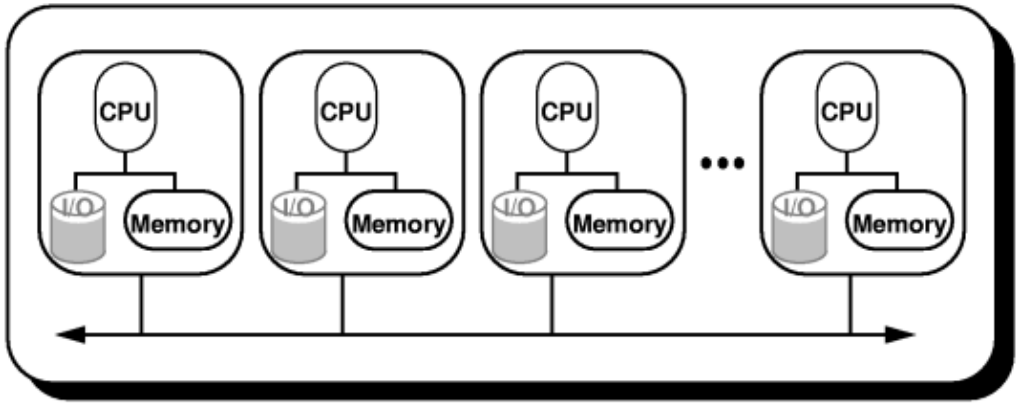
- Centrálna a spoločná **zdieľaná pamäť**
  - zbernica obsluhuje požiadavky CPU na prístup do pamäte a zabezpečuje spravodlivé obslúženie s minimálnym oneskorením



- Procesory sú typicky blízko seba (napr. viacjadrové CPU, GPU, „vlákna v procese")
- **Problémy:** cache coherence, memory contention

Paralelné výpočty so distribuovanou pamäťou

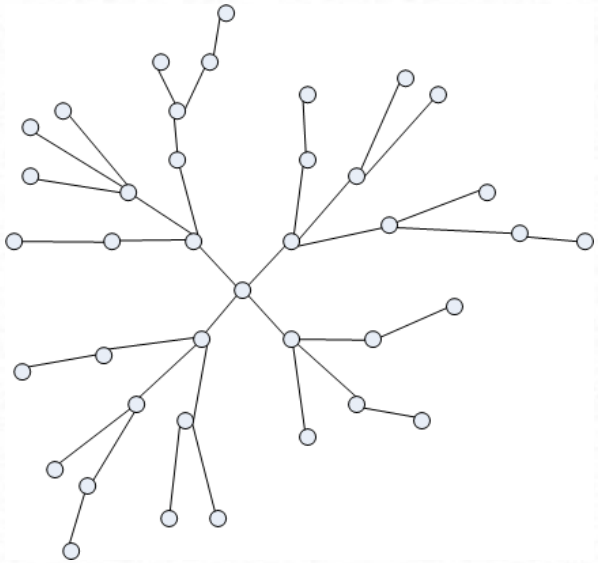
- Autonómne procesory s vlastnou pamäťou prepojené komunikačnou sieťou



- Komunikácia realizovaná posielaním správ
- Procesory sú typicky od seba fyzicky vzdialené (clustre, BOINC, „superpočítače", ...)
- **Problém:** vplyv komunikačnej siete na výpočet

Distribuované výpočty

- = paralelné výpočty
  s distribuovanou pamäťou
- Dôraz na komunikáciu
  (nie paralelné riešenie
  výpočtového problému)
- Cieľom sú algoritmy
  na koordináciu procesov (uzlov) siete:
  - routovanie, broadcasting, nájdenie minimálnej kostry, odolnosť voči chybám (fault-tolerance), voľba šéfa, clustering, ...



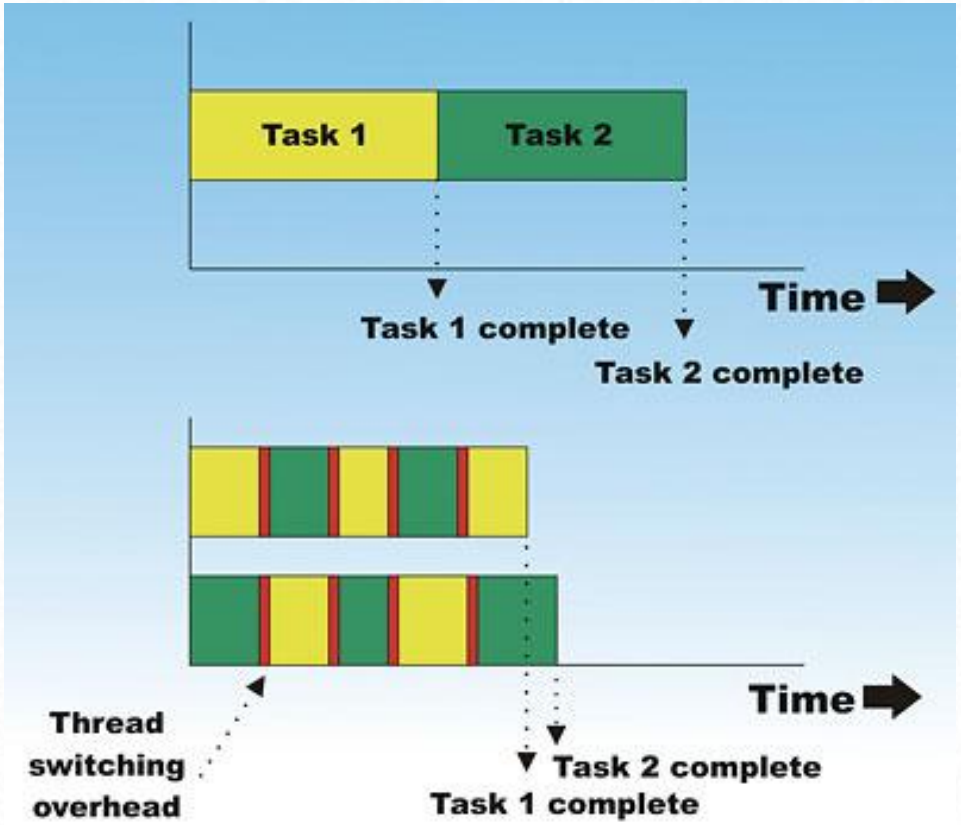
Konkurentné programovanie

Konkurentné programovanie

- Koordinácia procesov pri pristupovaní k zdieľaným zdrojom
  - väčšinu základných riešení poznáte z Operačných systémov
  - z OS:
    - vzájomné vylúčenie, deadlock, livelock, mutex, semafor
  - dôležité aj pri paralelnom programovaní so zdieľanou pamäťou
    - zdieľaným prostriedkom je pamäťové miesto

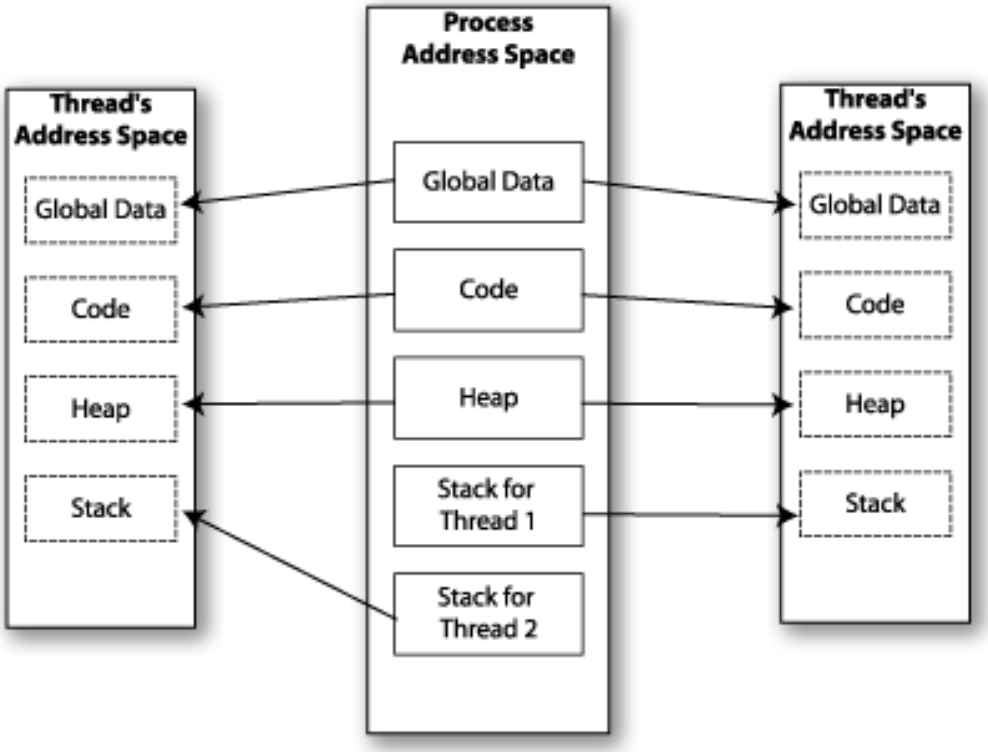
Paralelizmus vs. pseudoparalelizmus

- **Pseudoparalelizmus** vzniká, keď niekoľko procesov beží „súbežne" na jednom procesore, pričom na vytvorenie súbežnosti sa používa „time-slicing".



Vlákno (Thread)

- Najbežnejšia realizácia konkuretného procesu



- Každé vlákno má vlastný stack
- Heap je zdieľaný

Interakcia procesov (pohľad OS)

- Procesy si neuvedomujú iné procesy
  - súťaženie o zdieľané prostriedky
- Procesy si nepriamo uvedomujú iné procesy
  - kooperácia zdieľaním
- Procesy si uvedomujú iné procesy
  - kooperácia komunikáciou

- Pri paralelnom programovaní si procesy uvedomujú iné procesy – kooperujú na výsledku

Vzájomné vylúčenie

- **Kritická sekcia**
  - časť kódu procesu, ktorá vyžaduje prístup k zdieľaným prostriedkom a počas vykonávania ktorej žiadne iné procesy nepristupujú k týmto prostriedkom
- **Vzájomné vylúčenie**
  - požiadavka, že ak nejaký proces je v kritickej sekcii pristupujúcej k zdieľaným prostriedkom, potom žiaden iný proces nie je v kritickej sekcii pristupujúcej k niektorému z týchto zdieľaných prostriedkov

Podmienky vzájomného vylúčenia

- V skratke (z OS):
  - procesu sa nebráni vstúpiť do kritickej sekcie, ak v nej nie je iný proces
  - žiadne predpoklady o rýchlosti procesov
  - proces je v kritickej sekcii len konečný čas

Race condition

- Situácia, kedy výsledok procesu je kriticky závislý na postupnosti alebo načasovaní iných udalostí.
- Premenná A zdieľaná 2 totožnými procesmi:

`A = 0;`
`A++;`
 	
`if (A > 0) print(A);`
	
	`A = 0;`
	
	`A++;`
	`if (A > 0) print(A);`

Vzájomné vylúčenie s podporou HW

- Zakázanie prerušenia
  - funguje pri pseudoparalelizme, nie pri multiprocesoroch
- Atomické inštrukcie (vďaka podpore pamäte a procesora)
  - TEST-AND-SET
    - napr. Dual-Port RAM
  - COMPARE-AND-SWAP
    - 3 parametre: pamäťové miesto, pôvodná a nová hodnota
    - pamäťové miesto nastaví na novú hodnotu, len ak je tam aktuálne uložená pôvodná hodnota

TEST-AND-SET

- Pseudokód atomickej operácie test-and-set (nastaví novú hodnotu a vráti pôvodnú):

```
int test_and_set(int* reg, int newval){
        int oldval = *reg;
        *reg = newval;
        return oldval;
}
```

Činné čakanie „busy-waiting“

Zdieľaný zámok

```
volatile int lock = 0;
void Critical() {
  while (test_and_set(&lock, 1) == 1);
  kritická sekcia
  lock = 0;
}
```

Uvoľnenie zámku

Vzájomné vylúčenie cez algoritmus

- Dekkerov algoritmus (2 procesy)
- Petersonov algoritmus (2 procesy)
- Lamportov algoritmus pekára (n procesov)
- Eisenberg & McGuirov algoritmus (n procesov)
- Szymanského algoritmus (n procesov)

Lamportov algoritmus pekára

```
bool entering[NUM_THREADS] = {false, …, false};
int number[NUM_THREADS] = {0, …, 0};

void lock(int i) {
  entering[i] = true;
  number[i] = 1 + max(number[0], ..., number[NUM_THREADS-1]);
  entering[i] = false;

  for (int j = 0; j < NUM_THREADS; j++) {
    while (entering[j]);
    while ((number[j] != 0) && ((number[j], j) < (number[i], i)));
  }
}

void unlock(int i) {
  number[i] = 0;
}
```

```
void thread(int i) {
  while (true) {
    lock(i);
    // kritická sekcia
    unlock(i);
    // nekritická sekcia
  }
}
```

Treba tieto algoritmy?

- CUDA nemá mechanizmus mutexov
  - ako pomocou atomických read-write operácií naprogramovať vlastný mutex?

Deadlock, livelock, starvation

- **Deadlock**
  - situácia kedy 2 alebo viac procesov nie sú schopné pokračovať, pretože každý z nich čaká na iný proces, aby niečo vykonal
- **Livelock**
  - situácia, kedy 2 alebo viac procesov sústavne mení svoj stav reagujúc na zmenu stavu iného procesu bez toho, aby spravili užitočnú prácu
- **Starvation**
  - situácia, kedy proces pripravený na beh nie je spustený plánovačom

Pasívne vs. aktívne čakanie

- Predchádzajú príklady boli ukážky aktívneho čakanie (`while (...) ;`) - „busy waiting"
  - mrhá sa procesorovým časom
  - spinlock – aktívne čakanie sa niekedy využíva v jadrách OS, kedy sa predpokladá krátke čakanie (v porovnaní so context-switchingom a reschedulingom)
- Pasívne čakanie (zámky, monitory, semafory)
  - proces je zablokovaný, až kým nie je možnosť pokračovať alebo nenastane prerušenie procesu
  - vyžaduje sa podpora OS

Zámky a monitory

- Podpora pre vzáj. vylúčenie je často vo forme zámkov (C, C++, Java, ...) alebo monitorov (Java)
- **Zámok**
  - proces smie vstúpiť do kritickej sekcie, iba ak vlastní príslušný zámok (lock/acquire)
  - po skončení kritickej sekcie proces uvoľní zámok (unlock/release)
- **Monitor**
  - iba jeden proces môže byť v kritickej sekcii zviazanej s daným monitorom

Semafor

- Synchronizačný objekt s počítadlom
- Na začiatku inicializovaný na nejakú nenulovú hodnotu
- Základné operácie:
  - signal (V) – atomicky zvýši hodnotu počítadla o 1
  - wait (P) –
    - ak je počítadlo nulové, proces nepokračuje (čaká), kým je nulové
    - ak je počítadlo nenulové, počítadlo sa atomicky zníži o hodnotu 1

Rekurzívny vs. nerekurzívny zámok

- Rekurzívny (reentrant) zámok resp. monitor:
  - ak proces vlastní zámok, opätovná požiadavka na jeho získanie (lock/acquire) nie je blokovaná
  - počet lock a unlock musí byť rovnaký
  - so zámkom je asociovaná informácia, ktorý proces ho aktuálne vlastní
- Nerekurzívny zámok
  - opätovná požiadavka na už získaný zámok vedie k deadlocku
  - binárny semafor (počítadlo iniciálne nastavené na 0)

API pre konkurentné programovanie

- Java (java.util.concurrent, java.util.Thread)
  - monitory a zámky sú rekurzívne
- pthreads (POSIX threads)
  - pre zámky ide nastaviť, či sú rekurzívne alebo nie
- Win32 threads
- OpenMP (?)
  - Open Multi-Processing
  - štandardizované multiplatformové API pre paralelné programovanie so zdieľanou pamäťou

Klasické synchronizačné problémy

- Problém producenta a konzumenta
  - zdieľaný „sklad" s obmedzenou kapacitou, do ktorého producent pridáva položky a konzument ich vyberá
- Problém čitateľov a zapisovateľov
  - exkluzívny prístup zapisovateľov
  - súbežný (zdieľaný) prístup čitateľov
- Problém obedujúcich filozofov
  - synchronizácia pri použití viacerých zdrojov

Paralelizmus v Jave

- java.util.Thread
  - nastavenie priority
  - *ThreadGroup* (skupinový manažment)
  - používateľské vs. daemon vlákna
  - *Thread.sleep*
  - prerušenie cez metódu *interrupt()*, kontrolovanie prerušenie cez *isInterrupted()*
  - čakanie za skončením vlákna – metóda *join()*

Monitory

- Monitory zabezpečujú pre vlákna:
  - vzájomné vylúčenie (mutual exclusion)
  - kooperáciu
- S každým objektom je asociovaný monitor
- Monitor kontroluje vykonávanie istých častí kódu: kritických sekcií asociovaných s monitorom
- V každom okamihu môže vlastniť monitor (vykonávať kód kritickej sekcie monitora) nanajvýš jedno vlákno

Monitory v kóde

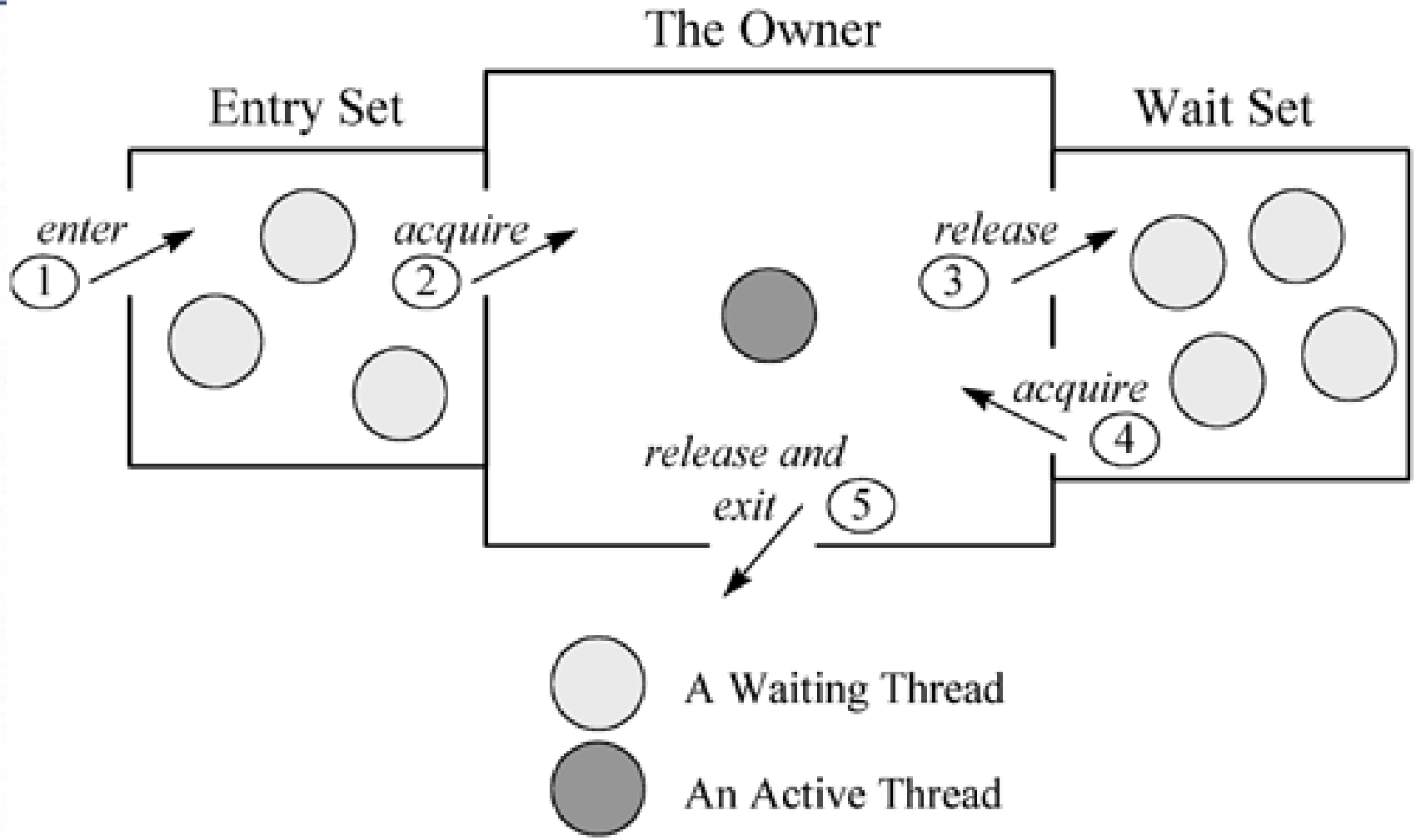
```
synchronized (objekt) {
        .... kód kritickej sekcie monitora
        asociovaného s objektom objekt ...
}
```

- Ak chceme synchronizovať celý kód metódy voči monitoru asociovanému objektu *this*, stačí pridať „*synchronized*" ku hlavičke inštančnej metódy

Kooperácia vlákien

- Podpora kooperácie (eliminácia činného čakania na splnenie podmienky):
  - *wait()* – vlákno v kritickej sekcii sa vzdá monitora dovtedy, kým od iného vlákna neprijme notifikáciu
  - *notify(), notifyAll()* – vlákno v kritickej sekcii oznámi nejakému čakajúcemu vláknu, resp. čakajúcim vláknam, že môžu v kritickej sekcii pokračovať
  - *wait(), notify(), notifyAll()* sú inštančné metódy objektu, s ktorým je asociovaný monitor

Schéma fungovania monitorov



Volatile premenné

- volatile označuje premenné, ktoré môžu byť modifikované viacerými vláknami a inštruuje procesor, aby vždy použil „aktuálnu" hodnotu namiesto hodnoty cachovanej v registroch

```
public void doWork()
{
    while (!shutdownRequested)
            processing();
}
```

Ďalšie možnosti synchronizácie

- Problém: systém monitorov je obmedzený, dovoľuje len hierarchické získavanie monitorov (zámkov)
- Riešenie:
  - *Semaphore: acquire/release*
  - *java.util.locks: Lock, Condition, ReadWriteLock, ReentrantLock, ReentrantReadWriteLock*
  - *Lock: lock/unlock*
- Synchronizované kolekcie: *ArrayBlockingQueue*
- Atomické premenné

Ďalšie možnosti synchronizácie

- *CountDownLatch*: umožňuje pozastaviť vlákna, kým sa nevykoná určitá množina operácií (countDown, await)
- *CyclicBarrier*: umožnuje pozastaviť vlákna, kým zadaný počet vlákien nedosiahne definovanú bariéru (await)

- Analogické mechanizmy sú aj v pthreads
  - monitor (Java) = condition variable (pthreads)

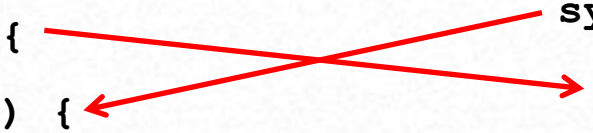
Príklad (1)

- Chceme implementovať bankový systém
  - podpora pre súbežné spracovanie transakcií
  - cieľ maximalizovať paralelizmus
    - intuícia: snažíme sa uzamykať „čo najmenšie časti"
    - fine-grained uzamykanie
- Základná operácia:
  - transfer(A, B, amount)
    - presunie sumu amount z účtu A na účet B

Príklad (2)

```
void transfer(A, B, amount) {
  synchronized(A) {
    synchronized(B) {
      withdraw(A, amount);
      deposit(B, amount);
    }
  }
}
```

```
void transfer(B, A, amount) {
  synchronized(B) {
    synchronized(A) {
      withdraw(B, amount);
      deposit(A, amount);
    }
  }
}
```



- Fine-grained zamýkanie môže viesť k deadlocku
- Treba zaviesť „globálnu" disciplínu

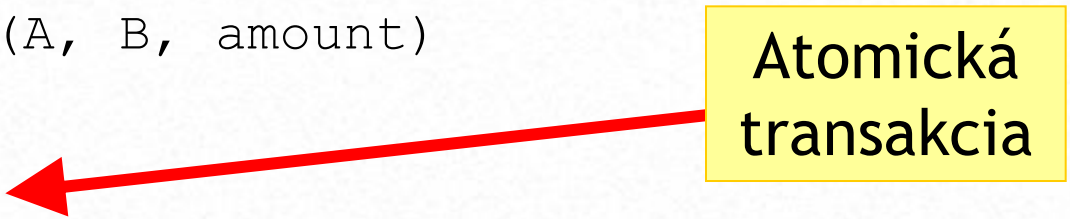
Príklad (3)

```
void transfer(A, B, amount) {
  synchronized(bank) {
    withdraw(A, amount);
    deposit(B, amount);
  }
}
```

- Coarse-grained zamykanie znižuje konkurentnosť a paralelizmus ...

Transakcie ako v databázach?

```
void transfer(A, B, amount)
{
  atomic {
    withdraw(A, amount);
    deposit(B, amount);
  }
}
```



- Transakcia
  - systém garantuje, že výsledok bude rovnaký, ako v prípade, keď sa postuposť príkazov vykoná ako atomická (neprerušiteľná) operácia

Transakčná pamäť (1)

- **Pamäťová transakcia**
  - atomická a izolovaná postupnosť prístupov do pamäte
- **Transakčná pamäť**
  - poskytuje pamäťové transakcie pre procesy pristupojúce k zdieľanej pamäti
  - inšpirácia z databáz

Transakčná pamäť (2)

- **Atomickosť**
  - pri commite sa všetky pamäťové zmeny vykonajú akoby naraz
- **Izolovateľnosť**
  - iný kód nemôže pozorovať zmeny pred commitom
- **Serializovateľnosť**
  - výsledok konkurentného vykonávania transakcií musí byť rovnaký, ako keby boli transakcie vykonávané v sérii postupne za sebou

Transakčná pamäť (3)

- O transakcie sa stará transakčný manažer:
  - Softvérová transakčná pamäť (STM)
    - asi 2-8 krát pomalšia ako sekvenčné spracovanie
  - Hardvérová akcelerovaná sotvérová transakčná pamäť (HASTM)
    - 1.8 – 5.6 krát pomalšia ako sekvenčné spracovanie
- Konkurentné programovanie bez zámkov (paralelné programovanie pre „masy")
- Intenzívne a aktuálne skúmaná oblasť (výskum)
- Implementácie STM pre mnohé prog. jazyky

Ďakujem za pozornosť !