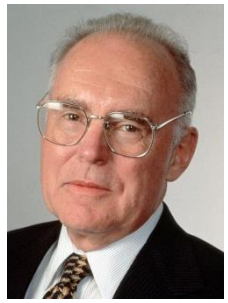
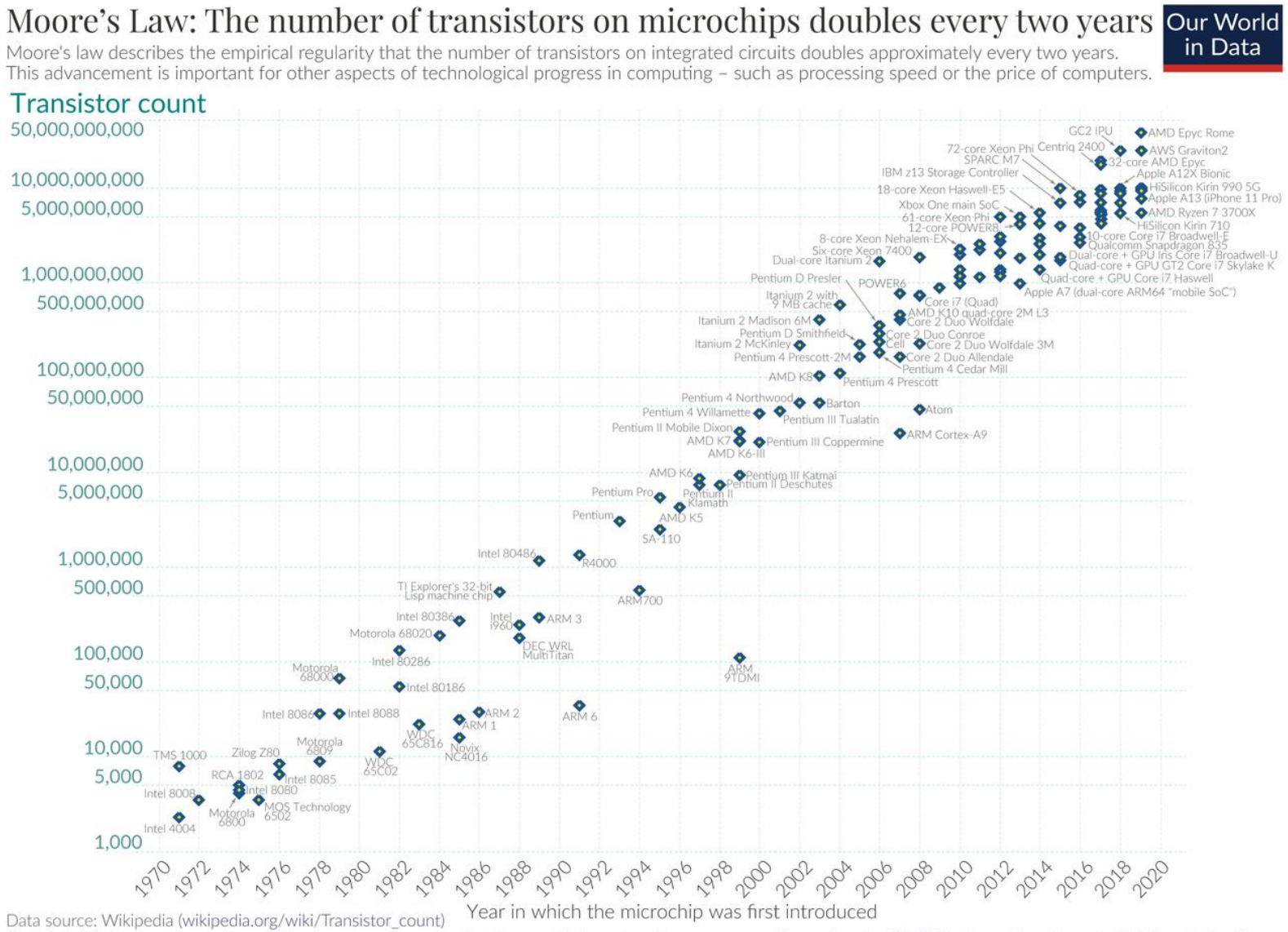
Úvod do PDS

limity počítačov von Neumannovho typu



- Moorov zákon (pozorovanie 1958-1965):
  - zložitosť čipov sa zdvojnásobí každé 2 roky pri zachovaní ceny
  - pozorovanie platí dodnes
  - Gordon E. Moore – spoluzakladateľ Intelu
- fyzikálne limity súčasných technológií
  - rýchlosť prenosu informácie je obmedzená rýchlosťou svetla (30 cm / ns)
  - hranice hrúbky vodivého kanála (3 nm – min. 1 nm (10 Å))
  - energetické nároky rastú s výkonom procesora

Moorov zákon – pozorovanie 1971-2020



prečo paralelizmus ?

- úlohy, ktoré dokážu „použiť“ všetok **disponibilný výpočtový výkon**
  - simulácie, vedecké výpočty, grafika, kryptomena ...
- potreba spracovávať čoraz **viac údajov** (Big data)
  - 59 ZB uložených údajov (IDC 2020) { k, M, G, T, P, E, Z, Y }
  - ≈ 10 EB prenesených údajov denne Internetom
    (za minútu – 3 mil. FB Posts, 4 mil. G searches, 50 tis. Instagram)
    https://www.internetlivestats.com/
  - LHC CERN – 30 PB raw data denne – 300 PB Spark DB

paralelizmus = využitie viacerých súčasne pracujúcich „počítačov“ na riešenie úlohy

paralelizmus dnes

- viacjadrové CPU (dnes 2 jadrá sú minimum)
- general purpose GPU stream processing
  - NVIDIA Tesla H100, 17000 jadier, 60 Tflops, 80 mld tr.

- Google Server Farm: 30x 450000 počítačov, 2500 MW
- BOINC
  - Berkeley Open Infrastructure for Network Computing, 300000 aktívnych CPU, 30 Pflops (denný priemer)
- 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?
- **Vyvážiť 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?

- 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, ...*
- Zámer predmetu PDS:
  - ukázať niektoré **základné problémy** a ich možné riešenia abstrahujúc od technických a implementačných detailov (zdanlivo teoretické veci)
  - praktické ukážky počas tutoriálov (3-4 za semester)

Prečo je paradigma PDS náročná?

- Pre ľudí je **sekvenčné** rozmýšľanie **prirodzené**
  - paralelné programovanie vyžaduje oveľa hlbšie pochopenie podstaty problému
  - manažérsky prístup – treba manažovať prácu procesorov
- Treba myslieť v **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 (beh závisí len od vstupu – opakovateľnosť)
  - teoretická analýza môže predísť nejednoznačnostiam

základné typy výpočtových modelov

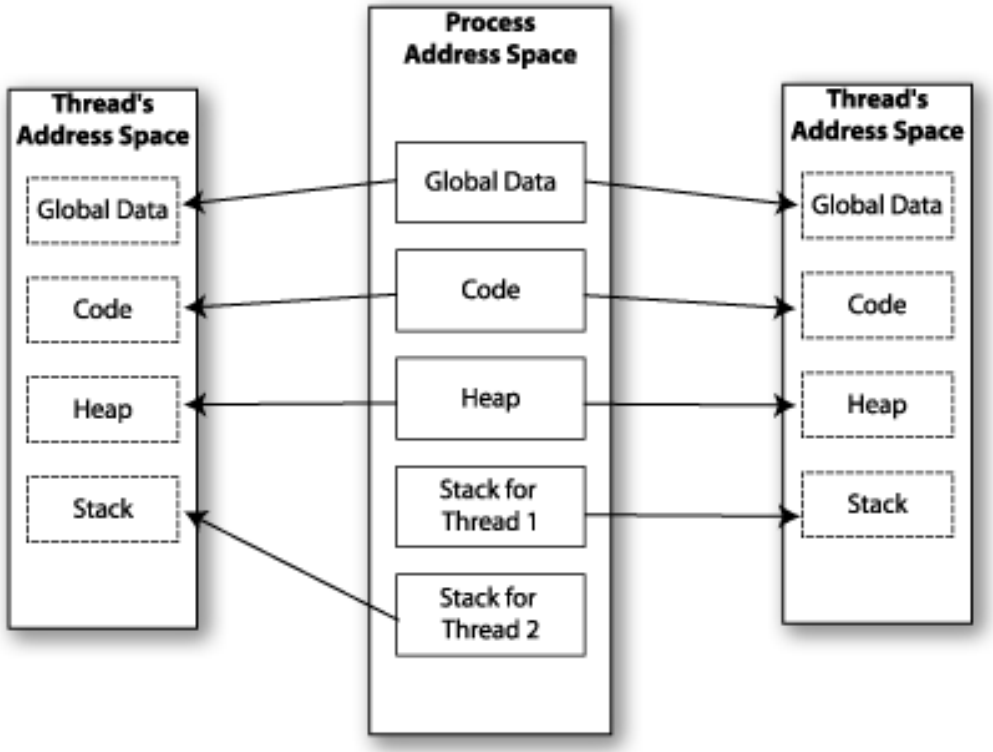
- **Konkurentné výpočty** (programovanie)
  jednoprocesorové systémy
- **Paralelné výpočty** (programovanie)
  silne spriahnuté viacprocesorové systémy
  komunikácia pomocou systémovej zbernice
  resp. prístupom k zdieľanej pamäti
- **Distribuované výpočty** (programovanie)
  slabo spriahnuté viacprocesorové systémy
  komunikácia posielaním správ
- často sa pojmy zamieňajú ...

proces a procesor

- **Proces**
  - jedna realizácia (vykonávanie) inštrukcií programu na konkrétnom zariadení (procesore)
  - z nášho pohľadu aj vlákna sú procesy
    nezamienať si proces v OS s procesom ako základným stavebným a konceptuálnym prvkom paralelných výpočtov
- **Procesor**
  - fyzické zariadenie vykonávajúce proces
- **Výpočtový uzol (node)**
  - (viacprocesorové) zariadenie, prepojené s ostatnými zariadeniami pomocou komunikačných kanálov

vlákno (thread)

- najbežnejšia realizácia konkuretného procesu



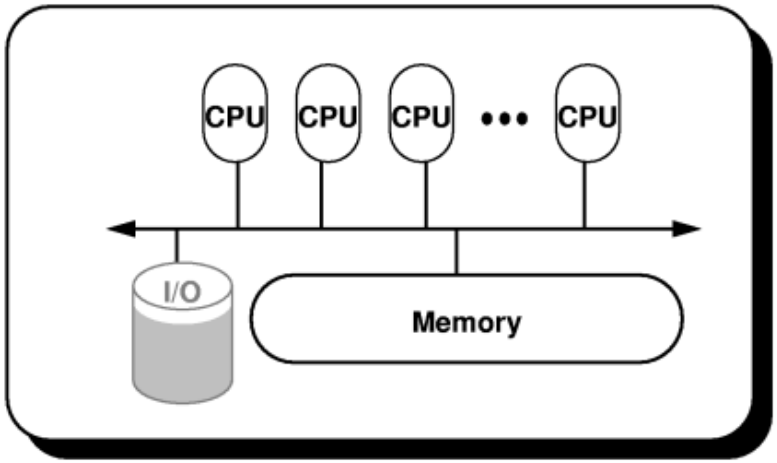
- každé vlákno má vlastný zásobník (stack)
- halda (heap) je zdieľaná

konkurentné programovanie

- Efektívne využitie zdrojov
  - 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 (využívanie procesora, pamäte, prístup k externým zariadeniam)
  - problémy vzájomného vylúčenia, uviaznutia, vyhladovania

viacprocesorové systémy – paralelné výpočty

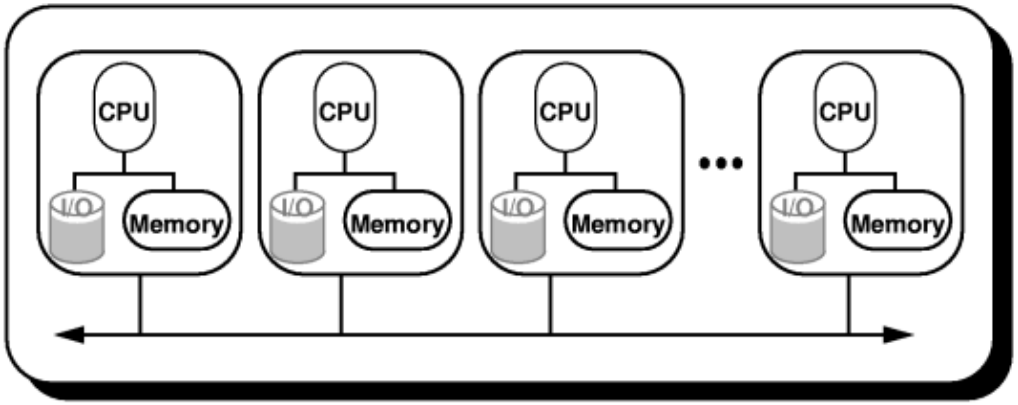


- centrálna a spoločná **zdieľaná pamäť**
  - všetky CPU pristupujú do spoločnej pamäte prostredníctvom zbernice
- procesory sú typicky blízko seba (napr. viacjadrové CPU, GPU, „vlákna v procese“)
- medziprocesorová komunikácia rýchlymi kanálmi

paralelné algoritmy

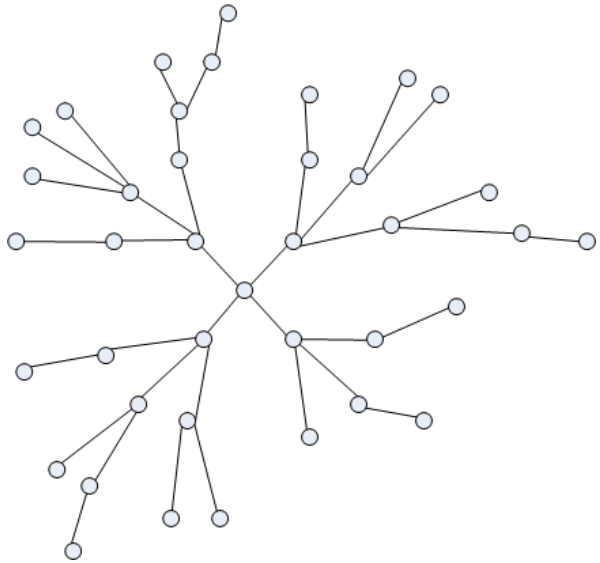
- efektívne využitie viacerých procesorov
- SIMD – jeden tok inštrukcií vykonávaný viacerými procesormi (jadrami) naraz (GPGPU)
  paralelizmus na dátovej úrovni – netreba riešiť konkurenciu
- SISD s implicitným vláknovým paralelizmom (work-time paralelizmus)
- riešenie kolízií prístupu do pamäte
- riešenie kolízií prístupu k spoločným zariadeniam

distribuované výpočty



- Autonómne procesory s vlastnou pamäťou prepojené komunikačnou sieťou
- Komunikácia realizovaná posielaním správ
- Procesory sú typicky od seba fyzicky vzdialené (clustre, BOINC, „superpočítače“, …)

fundamentálne distribuované algoritmy



- dôraz na komunikáciu
  (nie paralelné riešenie
  výpočtového problému)
- distribuované algoritmy
  na koordináciu procesov v uzloch siete:
  - smerovanie (routing), broadcasting, nájdenie minimálnej kostry, voľba šéfa, clustering, ...
  - odolnosť voči chybám (fault-tolerance)

Konkurentné programovanie

Konkurentné programovanie

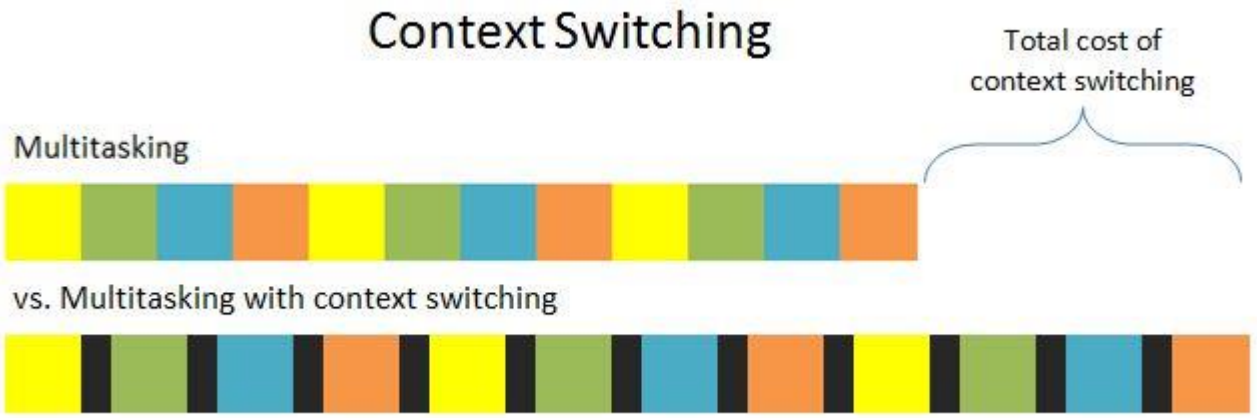
- možnosti súčasného vykonávania viacerých úloh na jenom procesore – konkurentné programovanie
- základná úloha riadenia výpočtu - **koordinácia procesov** pri pristupovaní k zdieľaným zdrojom procesora
  - väčšinu základných riešení známych z Operačných systémov
    - vzájomné vylúčenie, livelock, mutex, semafor, deadlock, starvation
  - dôležité aj pri paralelnom programovaní so zdieľanou pamäťou
    - zdieľaným prostriedkom je pamäťové miesto

možnosti interakcie procesov vo viacúlohovom OS (multitasking)

- procesy si „neuvedomujú“ iné procesy
  - preemptívne riadenie, scheduling
    (prideľovanie procesora po krátkych časových úsekoch
    s využitím mechanizmu prerušenia)
    súťaženie o zdieľané prostriedky, riadené OS
- procesy si „nepriamo uvedomujú“ iné procesy
  - kooperácia zdieľaním („ohľaduplné“ využívanie zdrojov)
- procesy si „uvedomujú“ iné procesy
  - kooperácia komunikáciou (bez zásahu OS)
  - možno efektívne využívať paralelné algoritmy

paralelizmus vs. pseudoparalelizmus

- **pseudoparalelizmus** niekoľko procesov beží „súbežne“ na jednom procesore, pričom na vytvorenie súbežnosti sa používa prideľovanie krátkych časových úsekov (time-slicing).
- preemptívny multitasking
  procesor zaťažuje aj prepínanie kontextu (context switching)



základný problém multitaskingu – súbeh (race condition)

- **Súbeh** (race condition) - situácia, kedy výsledok procesu je kriticky závislý na postupnosti alebo načasovaní iných udalostí.
- Premenná A zdieľaná dvomi rovnakými procesmi:

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

riešenie súbehu vzájomným vylúčením (mutual exclusion)

- **Vzájomné vylúčenie**
  - požiadavka, ktorá zabezpečí, že žiadne dva konkurujúce si procesy nie sú v kritickej sekcii v tom istom čase
- Riešenie hardvérové, softvérové, s podporou OS
- Riešenie pomocou **kritickej sekcie**
  - časť programu, ktorá nemôže byť vykonávaná viac než jedným z konkurujúcich si procesov
  - procesu sa nebráni vstúpiť do kritickej sekcie ak v nej nie je iný proces (inak hrozí vyhladovanie – starvation)
  - proces je v kritickej sekcii len konečný čas

vzájomné vylúčenie s podporou HW

- Zakázanie prerušenia
  - funguje pri pseudoparalelizme, nie pri multiprocesoroch
- Atomické inštrukcie (podpora 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
  - EXCHANGE (SWAP)

zámok pomocou operácie TEST-AND-SET

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

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

Aktívne 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 s podporou HW

- **x86, x64**
  - xchg, cmpxchg – explicitné uzamknutie, ak jeden z operandov je adresa v pamäti
  - LOCK prefix – uzatvorí zbernicu a zabezpečí výlučný prístup do pamäte pre inštrukciu
- Alpha, PowerPC, MIPS, **ARM**
  - LL (load-link) – načíta obsah pamäťového miesta
  - SC (store-conditional) – zapíše novú hodnotu, ak obsah pamäťového miesta z predchádzajúcej LL inštrukcie nebol medzitým modifikovaný

vzájomné vylúčenie algoritmom

- Dekkerov algoritmus (2 procesy)
- Petersonov algoritmus (2 procesy)
- Lamportov „pekárenský“ algoritmus (n procesov)
- Eisenberg & McGuirov algoritmus (n procesov)
- Szymanského algoritmus (n procesov)

Lamportov algoritmus „pekára“

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

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

  for (int j = 0; j < NUM_THREADS; j++) {
    L2: while (choosing[j]);
    L3: 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
  }
}
```

pasívne vs. aktívne čakanie

- Predchádzajú príklady boli ukážky aktívneho čakania (**`while (…) ;`**) - „busy waiting“
  - mrhá sa časom procesora
  - spinlock – aktívne čakanie sa niekedy využíva v jadrách OS, kde sa predpokladá krátke čakanie (v porovnaní s prepnutím kontextu a preplánovaním)
- Pasívne čakanie (zámky, monitory, semafory)
  - proces je zablokovaný, až kým nie je možnosť pokračovať alebo nenastane prerušenie procesu
  - vyžaduje sa podpora OS (**`sleep; wakeup(p)`**)

pasívny zámok - mutex

```
struct mutex {
    enum {zero, one} value;
    queueType queue;
};

void lock(mutex m)
{
   if (m.value == one) m.value = zero;        /*  ??? race condition */
   else {
            /* place this process in m.queue */;
            /* block this process - sleep */;
   }
}

void unlock(mutex m)
{
   if (m.queue is empty()) m.value = one;     /*  ??? race condition */
   else {
            /* remove a process P from m.queue */
            /* place process P on ready list - wakeup(P) */
   }
}
```

zámky a monitory

- Podpora pre vzájomné 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 inicializovaným na nejakú nenulovú hodnotu
- Základné operácie:
  - wait (resp. acquire)
    - atomicky zníži hodnotu počítadla o 1
    - ak je hodnota počítadla záporná, proces sa zablokuje (uspí)
  - signal (resp. release)
    - atomicky zvýši hodnotu počítadla o 1
    - ak je hodnota počítadla menšia ako 1, zobudí (zmení stav na ready) jeden z čakajúcich procesov

rekurzívny vs. nerekurzívny zámok

- Rekurzívny (reentrantný) 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 chybe (uviaznutiu)
  - binárny semafor (počítadlo iniciálne nastavené na 1)

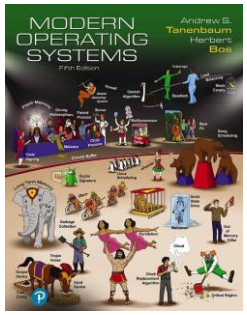
API pre konkurentné programovanie

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

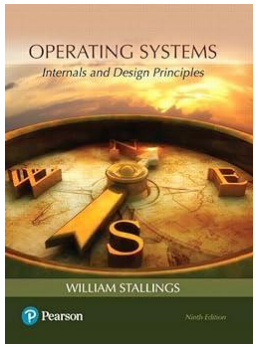
Ďakujem za pozornosť !

Odporúčaná literatúra na opakovanie

- A. Silberschatz, P. B. Galvin, G. Gagne: Operating System Concepts, 10.ed., Wiley, 2021, ISBN 978-1119800361



- A. S. Tanenbaum, H. Bos: Modern Operating Systems, 5.ed., Pearson, 2022



- W. Stallings: Operating Systems: Internals and Design Principles, 9.ed., Pearson, 2017, ISBN 978-0134670959