Konkurentné programovanie

Konkurentné programovanie

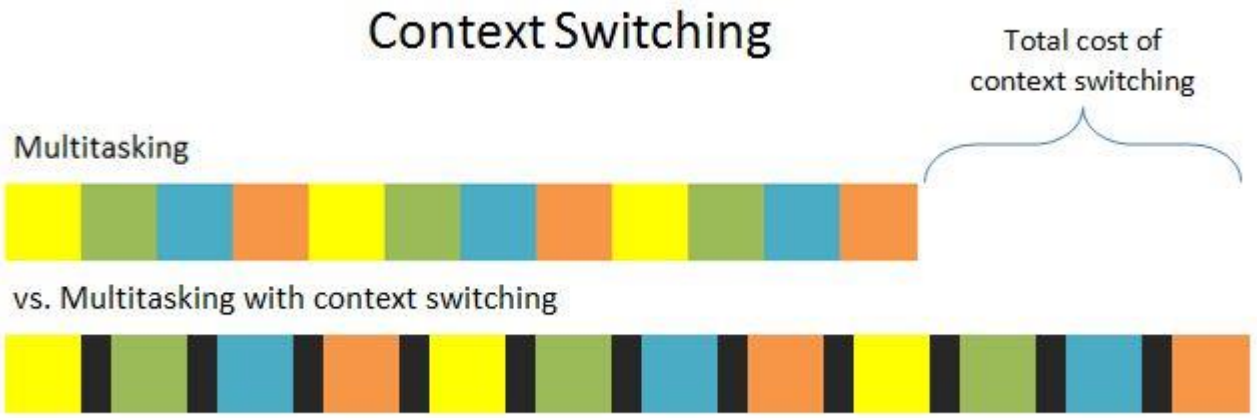
- možnosti súčasného vykonávania viacerých úloh na jenom procesore – konkurentné programovanie
- základná úloha riadenia výpočtu - **koordinácia procesov** pri pristupovaní k zdieľaným zdrojom procesora
  - väčšinu základných riešení známych z Operačných systémov
    - vzájomné vylúčenie, livelock, mutex, 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
    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
- **Kritická sekcia**
  - časť programu, ktorá nemôže byť vykonávaná viac než jedným z konkurujúcich si procesov

Podmienky vzájomného vylúčenia

- V skratke (z OS):
  - 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

- Riešenia vzájomného vylúčenia:
  - hardvérové
  - softvérové
  - s podporou operačného systému

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)

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

- **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

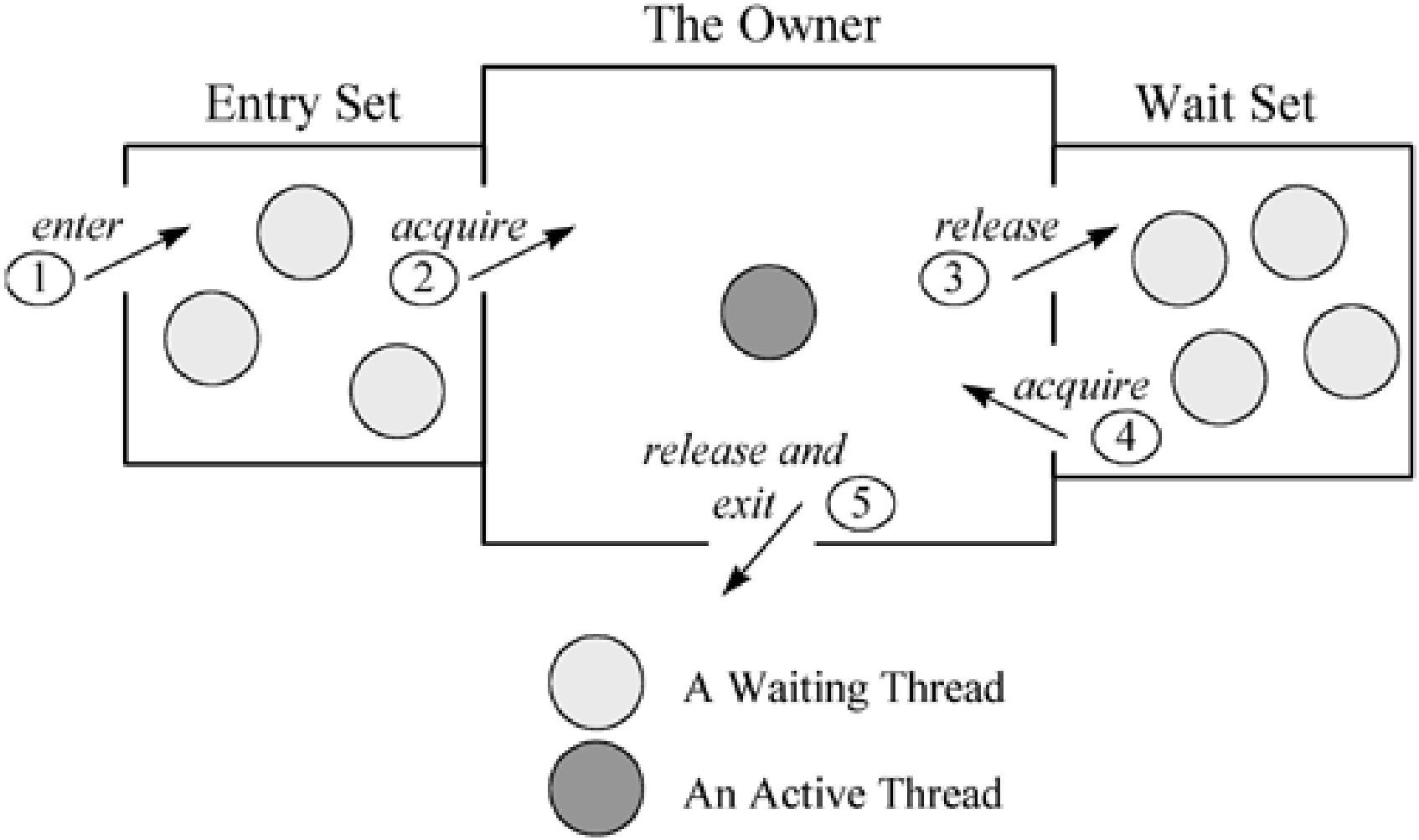
Klasické synchronizačné problémy

- 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

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

Schéma fungovania monitorov



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 sa vzdá monitora dovtedy, kým od iného vlákna neprijme notifikáciu
  - *notify(), notifyAll()* – vlákno oznámi nejakému čakajúcemu vláknu, resp. čakajúcim vláknam, že môžu v kritickej sekcii (zviazanej s monitorom) pokračovať
  - *wait(), notify(), notifyAll()* sú inštančné metódy objektu, s ktorým je asociovaný monitor

Volatilné premenné

- volatile označuje premenné, ktoré môžu byť modifikované viacerými vláknami a inštruuje prekladač, aby neoptimalizoval za sebou idúce operácie zápisu a čítania toho istého pamäťového miesta

```
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)

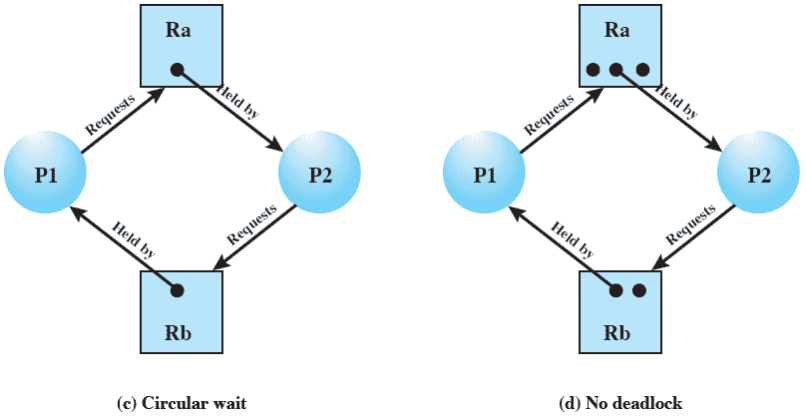
- Analogické mechanizmy sú aj v pthreads
  - monitor (Java) = condition variable (pthreads)

Deadlock, livelock, starvation

- **Uviaznutie (deadlock)**
  - situácia, kedy dva alebo viac procesov nie je schopných pokračovať, pretože každý z nich čaká na ukončenie alebo prostriedky držané iným procesom
- **Nekonečný cyklus (livelock)**
  - situácia, kedy dva alebo viac procesov sústavne mení svoj stav reagujúc na zmenu stavu iného procesu bez toho, aby napredovali
- **Vyhladovanie (starvation)**
  - situácia, kedy procesu je neustále odmietané pridelenie prostriedkov potrebných na napredovanie

podmienky vzniku uviaznutia

- mutual-exclusion - obmedzený počet procesov (jeden), ktoré môžu získať zdroje
- hold-and-wait – proces môže žiadať ďalšie zdroje aj keď už nejaké získal
- no preemption – získané zdroje sa neodoberajú
- circular wait – existuje cyklus, v ktorom každý proces vlastní zdroj, na ktorý čaká nasledujúci proces v cykle



(c) Circular wait (d) No deadlock

Riešenie problému uviaznutia

- ignorovanie
- detekcia – vysledovanie závislosti pokračovania procesov od čakania na procesy, vlastniace zdroje
  - ukončenie jedného procesu v cykle
  - odobratie prostriedkov procesu
- prevencia – odstránenie podmienok vzniku uviaznutia
  - dovoliť pridelenie len jedného zdroja (-> vyhladovanie)
  - obmedziť pseudoparalelizmus
  - prideľovať zdroje v usporiadaní podľa identifikátorov
  - obmedziť vzájomné vylúčenie (bankárov algoritmus)

Príklad (1)

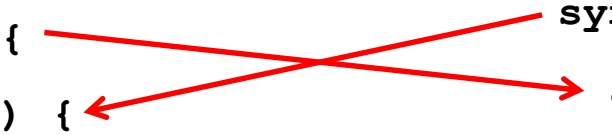
- Chceme implementovať bankový systém
  - podpora pre súbežné spracovanie transakcií
  - cieľ maximalizovať paralelizmus
    - intuícia: snažíme sa uzamykať „čo najmenšie časti“
    - jemnozrnné (fine-grained) uzamykanie

- 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);
    }
  }
}
```



- Jemnozrnné (fine-grained) zamykanie môže viesť k uviaznutiu
- Treba zaviesť „globálny poriadok“

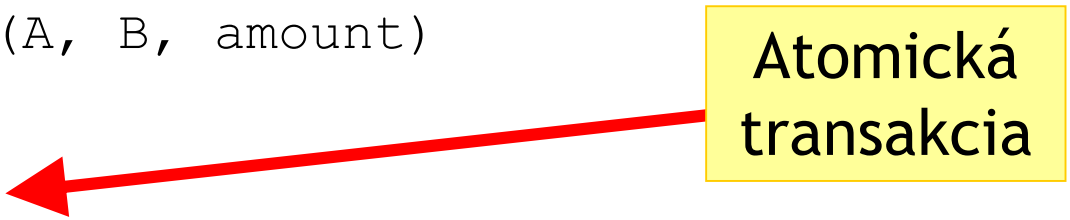
Príklad (3)

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

- Hrubozrnné (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

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

Transakčná pamäť

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

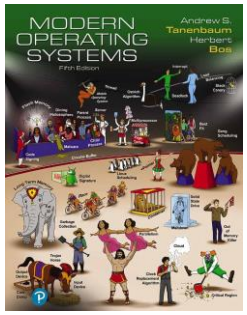
Transakčná pamäť

- O transakcie sa stará transakčný manažér:
  - softvérová transakčná pamäť (STM)
    - asi 2-8 krát pomalšia ako sekvenčné spracovanie
  - hardvérovo akcelerovaná TM (Intel TSX)
    - 1,8 – 5,6 krát pomalšia ako sekvenčné spracovanie
- Konkurentné programovanie bez zámkov (zjednodušenie paralelizácie riešenia úloh)
- Implementácie STM pre mnohé programovacie jazyky

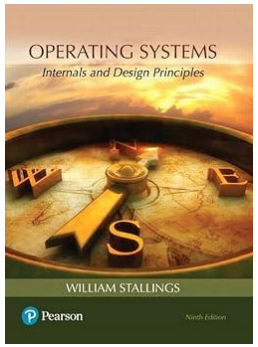
Ďakujem za pozornosť !

Odporúčaná literatúra

- 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