Operační systémy 1

# Synchronizace procesů a uváznutí

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

# Synchronizace vláken a procesů

- procesy a vlákna přistupují ke sdíleným zdrojům (paměť, souborový systém)
- příklad: současné zvýšení hodnoty proměnné o 1 (problém ABA)

```
;; X++
mov eax, [0xdeadbeef]
add eax, 1
mov [0xdeadbeef], eax
```

- scénář
  1. A: načte hodnotu proměnné X z paměti do registru (X = 1, eax = 1)
  2. A: zvýší hodnotu v registru o jedna (X = 1, eax = 2)

     — proces A je přerušen OS a pokračuje B —
  3. B: načte hodnotu proměnné X z paměti do registru (X = 1, eax = 1)
  4. B: zvýší hodnotu v registry o jedna (X = 1, eax = 2)
  5. B: uloží hodnotu zpět do paměti (X = 2, eax = 2)

     — proces B dokončí činnost a pokračuje A —
  6. A: uloží hodnotu zpět do paměti (X = 2, eax = 2)
- chyba souběhu (race-condition) $\Longrightarrow$ náročné na debuggování
- nejznámější chyba: Therac-25
- řešení $\Longrightarrow$ atomické operace a kritická sekce

# Atomický přístup do paměti

- obecně přístupy do paměti nemusí být atomické (záležitost CPU, překladače)
- $\Longrightarrow$ vícevláknové aplikace (přerušení); víceprocesorové počítače (cache)
- lze vynutit určité chování $\Longrightarrow$ klíčové slovo `volatile` – často záleží na překladači
- *memory barriers* umožňují vynutit si synchronizaci (záležitost CPU)

**Atomické operace**

- Test-and-Set (TAS): nastav proměnnou a vrať její původní hodnotu
- Swap: atomicky prohodí dvě hodnoty
- Compare-and-Swap (CAS): ověří, jestli se daná hodnota rovná požadované, a pokud ano, přiřadí ji novou hodnotu (CMPXCHG)
- Fetch-and-Add: vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna (XADD)
- Load-link/Store-Conditional (LL/CS): načte hodnotu, a pokud během čtení nebyla změněna, uloží do ní novou hodnotu

# Kritická sekce (critical section) (1/5)

- obecně třeba zajistit, aby se sdílenými zdroji pracoval jen jeden proces
- $\Longrightarrow$ vzájemné vyloučení (mutual exclusion)
- $\Longrightarrow$ problém kritické sekce
- kritické sekce je část kódu, kdy program pracuje se sdílenými zdroji (např. pamětí)
- pokud je jeden proces v kritické sekci, další proces nesmí vstoupit do kritické
- každý proces před vstupem žádá o povolení vstoupit do kritické sekce
- ukázka kódu:

```
do {
   // vstupni protokol KS
   ... prace se sdilenymi daty ...
   // vystupni protokol KS
   ... zbyly kod
} while (1);
```

- obr. Tan 103

# Kritická sekce (critical section) (2/5)

**Požadavky na kritickou sekci**

- vzájemné vyloučení – maximálně jeden proces je v daný okamžik v KS
- absence zbytečného čekání – není-li žádný proces v kritické sekci a proces do ní chce vstoupit, není mu bráněno
- zaručený vstup – proces snažící se vstoupit do KS do ní v konečném čase vstoupí

**V kontextu OS**

- potřeba synchronizovat činnost uživatelských procesů/vláken
- v kontextu jádra řada souběžných činností
  - nepreemptivní jádro OS (Linux $< 2.6$, Windows 2000, XP)
  - preemptivní jádro (Linux $\geq 2.6$, Solaris, IRIX)

# Kritická sekce (critical section) (3/5)

**Řešení**

- zablokování přerušení (použitelné v rámci jádra OS); více CPU $\Longrightarrow$ neefektivní

**Aktivní čekání**

- spinlocks
- řešení č. 1

```
int lock = 0;
while (lock) { } // cekej
lock = 1;
// kriticka sekce
lock = 0;
```

- vstup do kritické sekce a její zamčení není provedeno atomicky!!!
- **chyba souběhu (race-condition) !!!**

# Možný překlad pro aktivní čekání

## X86

```
wait:                           ;; while (lock) { }
    mov eax, [ebp - 4]
    cmp eax, 0x0
    jne wait
    mov eax, 0x01           ;; lock = 1
    mov [ebp - 4], eax
```

## Java Byte Code

```
0: aload_0              // while (lock) { }
1: getfield      #2     // Field lock:I
4: ifeq          10
7: goto          0
10: aload_0              // lock = 1
11: iconst_1
12: putfield      #2     // Field lock:I
```

# Kritická sekce (critical section) (4/5)

**Řešení č.2**

- uvažujme následující **atomickou** operaci

```
bool test_and_set(bool *target) {
    bool rv = *target;
    *target = true;
    return rv;
}
```

- a kód

```
while (test_and_set(&lock) == true) { /* prazdna smycka */ }
    // kriticka sekce
lock = false;
```

# Kritická sekce (critical section) (5/5)

**Řešení č.3**

- uvažujme následující **atomickou** operaci, která prohodí dvě hodnoty

```
void swap(bool *a, bool *b) {
    bool tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- a kód

```
key = true;
while (key == true)
    swap(&lock, &key);
// kriticka sekce
lock = false;
```

# Petersonův algoritmus

- řešení vzájemného vyloučení bez použití atomických operací

**Proces A**

```
lockA = true;
turn = B;
while (lockB && (turn == B)) { }
                ...
lockA = false;
```

**Proces B**

```
lockB = true;
turn = A;
while (lockA && (turn == A)) { }
                ...
lockB = false;
```

- vyžaduje férové plánování

# Semafor (1/3)

- chráněná proměnná obsahující počítadlo s nezápornými celými čísly
- operace P (*proberen* – zkusit): pokud je hodnota čísla nenulová, sníží hodnotu o jedna, jinak čeká, až bude hodnota zvýšena (operace někdy označována i jako *wait*)

```
void P(Semaphore s) {
    while (s <= 0) { }
    s--;
}
```

- operace V (*verhogen* – zvýšit): zvýší hodnotu o jedna (operace někdy označována jako *signal*, *post*)

```
void V(Semaphore s) {
    s++;
}
```

- operace P a V se provádí atomicky

# Semafor (2/3)

- binarní semafor – může nabývat hodnot 0, 1 (*mutex*, implementace kritické sekce)
- obecný semafor – slouží k řízení přístupu ke zdrojům, kterých je konečné množství
- implementace s pomocí aktivního čekání nebo OS ($\Longrightarrow$ pasivní čekání)

```
struct sem {
    int value;
    struct process * list;
};
void P(struct sem * s) {
    s->value--;
    if (s->value < 0) {
        // pridej proces do s->list;
        block(); // uspi aktualni proces
    }
}
```

# Semafor (3/3)

- dokončení...

```
void V(struct sem * s) {
    s->value++;
    if (s->value <= 0) {
        // odeber process P z s->list
        wakeup(P);
    }
}
```

- operace musí být provedeny atomicky (řešení $\Longrightarrow$ spin-lock na začátku operace)
- seznam by měl být jako FIFO
- spolupráce `wakeup` s plánovačem
- všimněte si záporné hodnoty `s->value` $\Longrightarrow$ počet čekajících procesů

# Další synchronizační nástroje

**Bariéry**

- synchronizační metoda vyžadující, aby se proces zastavil v daném bodě, dokud všechny procesy nedosáhnou daného bodu

**Read-Write zámky**

- vhodné pro situace, které čtou i zapisují do sdíleného prostředku
- čtecí a zapisovací režim zámku
- vhodný, pokud jde rozlišit čtenáře a písaře

**Podmíněná proměnná**

- čekání na změnu proměnné – neefektivní aktivní čekání
- operace `wait`, `signal`
- kombinace se zamykáním

# Monitor

- modul nebo objekt
- v jeden okamžik může kteroukoliv metodu používat pouze jeden proces/vlákno
- nutná podpora prog. jazyka
- Java (`synchronized`), .NET (`lock`)
- rozšíření o podporu čekání (`Wait`, `Pulse`, `PulseAll`) $\Longrightarrow$ možnost odemčít zámek společně s čekáním

# Monitor v Javě

```
public class Bank {
    private final String name;
    private int[] account;
    public synchronized void transfer(int from, int to, int amount) {
        account[from] -= amount;
        account[to] += amount;
    }
    public synchronized int summary() {
        int sum = 0;
        for (Integer a: account)
            sum += a;
        return sum;
    }
    public String getName() {
        return name;
    }
}
```

# Monitor v Javě a C#

**Java**

```
public synchronized void foo() {
    // kod
}
public void foo() {
    synchronized (this) {
        // kod
    }
}
```

**C#**

```
public void foo() {
    lock(this) {
        //kod
    }
}
```

# Synchronizační primitiva ve Windows

- obecný mechanismus – synchronizační objekty se nacházejí ve dvou stavech (signalizovaný vs. nesignalizovaný)
- signalizovaný objekt je dostupný (mutex, semaphore, event, thread, etc.)
- (univerzální) čekací funkce (`WaitForSingleObject`, `WaitForMultipleObject`) – čeká dokud se objekt(y) nedostanou do signalizovaného stavu
- čekací funkce slouží také k manipulaci s mutexy, semafory, ...
- `CreateMutex`, `CreateSemaphore`, ... (možnost vytvořit pojmenované objekty)
- `ReleaseMutex`, `ReleaseSemaphore`, `SetEvent`
- `SignalObjectAndWait` – kombinuje předchozí operace do jedné atomické

**Další synchronizační metody**

- Interlocked API (atomické operace), spinlocks (jádro)
- kritická sekce (`EnterCriticalSection`, `LeaveCriticalSection`)

# Synchronizační primitiva v unixech

**Synchronizace procesů**

- SYSTEM V IPC
- sdílená paměť, semafory, zasílání zpráv
- práce semafory (skupiny semaforů) – `semget`, `semctl`, `semop` (mj. společné rozhraní pro operace typu P a V) ...
- sdílené všemi procesy $\Longrightarrow$ správa oprávnění

**Synchronizace vláken**

- libpthread – mutexy, semafory, rw-zámky, bariéry (`pthread_mutex_lock`,`pthread_mutex_trylock`, `pthread_mutex_unlock`, `pthread_cond_wait`, `pthread_cond_signal`, `sem_wait`, `sem_post`, ...)
- futexy

**Atomické operace**

- chybí obecné rozhraní v uživatelském prostoru
- glib, lib_atomic_ops
- jádro používá vlástní sadu operaci (`atomic_read`, `atomic_set`, ...)

# Deadlock

# Deadlock

- uváznutí – systém se dostal do stavu, kdy nemůže dál pokračovat
- *U množiny procesů došlo k uváznutí (deadlocku), pokud každý proces z této množiny čeká na událost, kterou pouze proces z této množiny může vyvolat.*

**Užívání prostředků**

- request – požadavek na prostředek, není-li k dispozici, proces čeká
- use – proces s prostředkem pracuje
- release – uvolnění prostředku pro další použití

**Podmínky vzniku**

- Mutual exclusion – alespoň jeden prostředek je výlučně užíván jedním procesem
- Hold & wait – proces vlastní alespoň jeden prostředek a čeká na další
- No preemption – prostředek nelze násilně odebrat
- Circular wait – cyklické čekání (proces A vlastní prostředek 1, chce prostředek 2, který drží B a současně žádá o 1)

# Řešení deadlocku (1/4)

**Ignorace**

- ,,neřešení", v praxi často používané

**Detekce (detection & recovery)**

- pokud vznikne deadlock, je detekován a některý proces odstraněn
- k detekci se používá *alokační graf prostředků* a *graf čekání*
- alokační graf:
  - orientovaný graf
  - dva typy uzlů – prostředek, proces
  - hrana proces-prostředek – proces čeká na prostředek
  - hrana prostředek-proces – prostředek je vlastněn procesem
- graf čekání vznikne vynecháním uzlů prostředků a přidáním hran $P_n \to P_m$ pokud existovaly hrany $P_n \to R$ a $R \to P_m$, kde $P_n$ a $P_m$ jsou procesy a $R$ je prostředek
- deadlock vznikne, pokud je v grafu čekání cyklus
- $\Longrightarrow$ odebrání prostředků, odstranění procesu (Jak vybrat oběť?), opakované zpracování (rollback)
- Kdy má smysl provádět detekci?

# Řešení deadlocku (2/4)

**Zamezení vzniku (prevention)**

- snažíme se zajistit, že některá z podmínek není splněna
- zamezení výlučnému vlastnění prostředků (často nelze z povahy zařízení)
- zamezení držení a čekání
  - proces zažádá o všechny prostředky hned na začátku
  - problém s odhadem
  - plítvání a hladovění
  - množství prostředků nemusí být známé předem
  - jde použít i v průběhu procesu (ale proces se musí vzdát všech prostředků)
- zavedení možnosti odejmout prostředek – vhodné tam, kde lze odejmout prostředky tak, aby nešlo poznat, že byly odebrány
- zamezení cyklickému čekání – zavedení globálního číslování prostředků a možnost žádat prostředky jen v daném pořadí

# Řešení deadlocku (3/4)

**Vyhýbání se uváznutí (avoidance)**

- procesy žádají prostředky libovolně
- systém se snaží vyhovět těm požadavkům, které nemohou vést k uváznutí
- je potřeba znát předem, kolik prostředků bude vyžádáno
- tomu je přizpůsobeno plánovaní procesů
- **bezpečný stav** – existuje pořadí procesů, ve kterém jejich požadavky budou vyřízeny bez vzniku deadlocku
- systém, který není v bezpečném stavu, nemusí být v deadlocku
- systém odmítne přidělení prostředků, pokud by to znamenalo přechod do nebezpečného stavu (proces musí čekat)

# Řešení deadlocku (4/4)

**Algoritmus na bázi alokačního grafu**

- vhodný, pokud existuje jen jedna instance každého prostředku
- do alokačního grafu přidáme hrany (proces-prostředek) označující potenciální žádosti procesu a prostředky
- žádosti a prostředek se vyhoví pouze tehdy, pokud konverze hrany na hranu typu (prostředek-je vlastněn-procesem) nepovede ke vzniku cyklu