

Operační systémy

Přednáška 3: Komunikace mezi procesy

Časově závislé chyby

- Dva nebo několik procesů/vláken používá (čte/zapisuje) **společné sdílené prostředky** (např. sdílená paměť, sdílení proměnné, sdílené soubory,...).
- **Výsledek** výpočtu **je závislý na přepínání kontextu** jednotlivých procesů/vláken, které používají sdílené prostředky.
- Velmi špatně se detekují (náhodný výskyt)!

Příklad 1

- Dva uživatelé editují stejný soubor (např. v Unixu pomocí editoru vi).

User 1: vi f.txt

User 2: vi f.txt

Příklad 2

- Program v C přeložený do assembleru 32-bitového procesoru.

64-bit shared variable

```
unsigned long long var = 0x00000000ffffffff;      # 64 bit shared variable !  
                                                    # 8 bytes !
```

Thread A

```
...  
var++;  
...
```

```
addl $1, (%eax)      # ffffffff + 1 = carry bit  
...      # if context switch comes here the value of var is 0  
adcl $0, 4(%eax)     # add carry bit to 00000000  
...      # if context switch comes here the value of var != 0
```

Thread B

```
if (var == 0) { ... }  
else { ... }
```

Příklad 3

- Dvě vlákna inkrementují sdílenou proměnnou.

shared variable

```
...  
int counter = 0 ;  
...
```

Thread A

```
...  
counter++;  
...
```

```
...  
movl  ...  
addl $1, (%eax)  
movl  ...  
...
```

Thread B

```
...  
counter++;  
...
```

Kritické sekce

- **Kritická sekce**
 - Část programu, kde procesy používají sdílené prostředky (např. sdílená paměť, sdílená proměnná, sdílený soubor, ...).
- **Sdružené kritické sekce**
 - Kritické sekce dvou (nebo více) procesů, které se týkají stejného sdíleného prostředku.
- **Vzájemné vyloučení**
 - Procesům není dovoleno sdílet stejný prostředek ve stejném čase.
 - Procesy se nesmí nacházet ve sdružených sekcích současně.

Korektní paralelní program

- **Nutné podmínky**

1. Dva procesy se nesmí nacházet současně ve stejné sdružené sekci.
2. Žádné předpoklady nesmí být kladeny na rychlost a počet procesorů.
3. Pokud proces běžící mimo kritickou sekci nesmí být blokován ostatní procesy.
4. Žádný proces nesmí do nekonečna čekat na vstup do kritické sekce.

Bernsteinovy podmínky

$$R(p) \cap W(q) = \emptyset,$$

$$W(p) \cap R(q) = \emptyset,$$

$$W(p) \cap W(q) = \emptyset,$$

$R(i)$ - všechny sdílené proměnné pro čtení použité v procesu i ,

$W(i)$ - všechny sdílené proměnné pro zápis použité v procesu i .

- **Sdílené proměnné** mohou být **pouze čteny**.
- Příliš přísné => pouze teoretický význam.

Zákaz přerušení (DI)

- CPU je přidělováno postupně jednotlivým procesům za pomoci přerušení od časovače nebo jiného přerušení.
- Proces **zakáže všechna přerušení před vstupem** do kritické sekce a opět je **povolí až po opuštění** kritické sekce.
- Např. instrukce cli/sti pro Intel.
- **Nevýhoda:**
 - DI od jednoho uživatele blokuje i ostatní uživatele.
 - Ve víceprocesorovém systému, DI má efekt pouze na aktuálním CPU.
 - Zpomalí reakce na přerušení.
 - Problém se špatně napsanými programy (zablokují CPU).
- Užitečná technika uvnitř jádra OS (ale pouze na krátký čas).
- Není vhodná pro běžné uživatelské procesy!!!

Aktivního čekání vs. blokování

- Pouze **jeden proces může vstoupit** do kritické sekce.
- **Ostatní procesy musí počkat** dokud se kritická sekce neuvolní.
- **Aktivní čekání**
 - sdílená proměnná indikuje obsazenost kritické sekce
 - proces ve smyčce testuje aktuální hodnotu proměnné do okamžiku než se sekce uvolní
- **Blokování**
 - proces provede systémové volání, které ho zablokuje do okamžiku než se sekce uvolní

Lock proměnná

- Vzájemné vyloučení pomocí **(sdílené) lock proměnné**, kterou proces nastaví když vstupuje do kritické sekce.

```
int lock=0;
. . .
while (lock == 1);      /* busy waiting is doing */
lock=1;
critical_section();
lock=0;
. . .
```

- **Je to správné řešení?**

Striktní střídání

```
/* Process A */
```

```
...
```

```
while (TRUE) {  
    while (turn != 0); /* wait */  
    critical_section();  
    turn=1;  
    noncritical_section();  
}
```

```
/* Process B */
```

```
...
```

```
while (TRUE) {  
    while (turn != 1); /* wait */  
    critical_section();  
    turn=0;  
    noncritical_section();  
}
```

- **Nevýhody**

- Jeden proces může zpomalit ostatní procesy.
- Proces nemůže vstoupit do kritické sekce opakovaně (je porušen 3.bod z nutných podmínek ...).

Petersonův algoritmus

- T. Dekker (1968) navrhl algoritmus, který nevyžadoval striktní střídání.
- G. L. Peterson (1981) popsal jednodušší verzi tohoto algoritmu.
- L. Hoffman (1990) zobecnil řešení pro n procesů.

Petersonův algoritmus (2)

```
#define FALSE 0
#define TRUE  1
#define N      2                /* number of processes */

int turn;                       /* whose turn is it? */
int interested[N];              /* all values initially FALSE */

void enter_section (int process) /* process is 0 or 1 */
{
    int other;                  /* number of other processes */

    other = 1 - process;        /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;              /* set flag */
    while (turn == process && interested[other]); /* busy waiting */
}

void leave_section (int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from CS */
}
```

Instrukce TSL

- **Test and Set Lock** (TSL) instrukce **načte** obsah slova z dané adresy v paměti do registru a **nastaví** obsah slova na nenulovou hodnotu.
- CPU provádějící TSL instrukci **zamkne paměťovou sběrnici**, aby znemožnilo ostatním CPU v přístupu do sdílené paměti dokud se instrukce TSL nedokončí.
- TSL je atomická instrukce => **korektní hardwarové řešení.**
- **Výhody**
 - TSL může být použita při synchronizaci v multiprocesorových systémech se sdílenou pamětí.

Instrukce TSL (2)

enter_section:

```
TSL REGISTER, LOCK  
CMP REGISTER, #0  
JNE enter_section  
RET
```

| copy LOCK to REGISTER and set LOCK to 1
| was LOCK zero?
| if it was non zero, LOCK was set, so loop
| return to caller, critical section entered

leave_section:

```
MOVE LOCK, #0  
RET
```

| store a 0 in LOCK
| return to caller

Nevýhody aktivního čekání

- **Plýtvání časem procesoru.**
- Pokud OS používá prioritní plánování, potom může vzniknout **inverzní prioritní problém**
 - proces A má nižší prioritu a nachází se v kritické sekci
 - proces B má vyšší prioritu a čeká pomocí aktivního čekání na vstup do kritické sekce
 - pokud přiřazená priorita je fixní **dojde k uváznutí**

Vzájemné vyloučení pomocí blokování

- Lepší způsob než zákaz přerušení nebo aktivní čekání.
- Procesy, které nemohou vstoupit do kritické sekce jsou **zablokovány** (jejich stav se změnil na „blocked“ a jsou umístěny do čekací fronty).
- Blokující a deblokující operace jsou obvykle **implementovány jako služby jádra OS**.

Sleep a Wakeup

- **Sleep()**
 - Systémové volání, které zablokuje proces, který ho zavolal.
 - Zakáže alokování CPU pro tento proces a přesune ho do fronty kde bude čekat.
- **Wakeup(proces)**
 - Opačná operace, proces je uvolněn z fronty čekajících procesů a bude mu opět přidělováno CPU.
- Příklady:
 - MUTEX (MUTual EXclusion lock)
 - `pthread_mutex_lock()` , `pthread_mutex_trylock()`
 - `pthread_mutex_unlock()`
 - Podmíněné proměnné
 - `pthread_cond_wait()`
 - `pthread_cond_signal()`, `pthread_cond_broadcast()`

Problém producenta a konzumenta

- Jeden z klasických synchronizačních problémů, který demonstruje problémy paralelního programování.
- **Producent**
 - produkuje data a vkládá je do sdílené paměti.
- **Konzument**
 - vybírá data ze sdílené paměti.

Řešení pomocí sleep/wakeup

```
define N 100  
int count = 0;
```

```
/* number of slots in the buffer */  
/* number of items in buffer */
```

```
void producer(void)  
{  
    int item;  
    while(TRUE){  
        item = produce_item();  
        if (count == N) sleep();           /* if buffer is full, go to sleep */  
        insert_item(item);  
        count = count + 1;  
        if (count == 1) wakeup(consumer); /* was buffer empty? */  
    }  
}
```

```
void consumer(void)  
{  
    int item;  
    while (TRUE){  
        if (count == 0)                     /* critical point for context switching */  
            sleep();                         /* if buffer is empty, go to sleep */  
        remove_item(&item)  
        count = count - 1;  
        if (count == N - 1) wakeup(producer); /* was buffer full? */  
        consume_item(&item);}  
}
```

Sleep a Wakeup (3)

- Abychom se vyhnuli časově závislým chybám, musí být zaručeno, že nedojde k přepnutí kontextu v označeném místě.
- Tato podmínka není v tomto řešení garantována.
- **Problém**
 - Operace **wakeup()** je zavolána na proces, který není ještě uspán. Co se stane???
- **Řešení**
 - Wakeup waiting bit.
 - Když je **wakeup()** zavolána na proces, který ještě nespí, tento bit je nastaven.
 - Při pokusu uspat proces, který má nastaven tento bit, proces se neuspí, ale pouze se resetuje daný bit.