7. Synchronizace procesů

- Problém Producent vs. Konzument
- Kritická sekce
    - Charakteristika a podmínky pro ošetření KS
    - Aktivní vs. pasivní čekání
- Sdílený prostředek
- Řešení KS
    - Zákaz přerušení
    - Zamykací proměnná
    - Přesné střídání
    - Petersonovo řešení
    - Atomická instrukce
    - Sleep() a Wakeup()
    - Semafory a transakce
- Klasické synchronizační problémy

1. Problém Producent vs. Konzument

- Tento problém se týká situace, kdy jeden proces produkuje data a druhý proces je konzumuje.
- Producent produkuje data do sdílené paměti, zatímco konzument je spotřebovává.
- Může nastat situace, kdy konzument zkonzumuje data předtím, než je producent stihne vyrobit, nebo naopak.

Výchozí stav: count = 3

Běžící proces	Akce	Výsledek
Producent	R0 = count	R0 = 3
	R0 += 1	R0 = 4
Konzument	R1 = count	R1 = 3
	R1 -= 1	R1 = 2
Producent	count = R0	count = 4
Konzument	count= R1	**count = 2**

Pozn: V jazycích vyšší úrovně je inkrementace/dekremntace otázkou jediného příkazu v rámci jazyků nižší úrovně se jedná o posloupnost několika instrukcí

2. Kritická sekce

- Kritická sekce je část programu, ve které se pracuje se sdílenými prostředky.
    - Hrozí přístup více procesů nebo vláken najednou

Podmínky pro ošetření KS

- Žádné dva procesy nesmí být v jeden čas ve stejné KS
- Proces mimo KS nesmí blokovat jiný proces, který by chtěl vstoupit do kritické sekce (trvalost postupu)
- Na KS nesmí proces čekat nekonečně dlouho (konečné čekání)
- Počet a rychlost CPU nesmí mít vliv na řešení KS

Aktivní vs. pasivní čekání

- Aktivní čekání znamená, že proces neustále testuje, zda může přistoupit do KS => plýtvání procesorového času
- Pasivní čekání znamená, že proces je uspán a čeká, až bude kritická sekce uvolněna. Nedochází k plýtvání procesorového času, hrozí však uváznutí (dead lock), kdy 2 procesy čekají na událost, jež může uvolnit / vyvolat proces, který ale také čeká. Dalším problémem je Aktivní zablokování, kdy si procesy snaží navzájem vyhovět (2 lidé a úzká chodba) Dalším problémem je stárnutí procesu, kdy si procesy vyměňují přístup ke sdílenému prostředku a třetí proces se k němu nedostane.

3. Sdílený prostředek

- Sdílený prostředek je zdroj, ke kterému mají přístup různé procesy.
- Pro zajištění správné synchronizace je třeba, aby procesy přistupovaly k sdíleným prostředkům koordinovaně.

4. Řešení KS

- ## Zákaz přerušení
  - Nejjednodušší
  - Zákaz všech přerušení procesem, který vstoupí do KS a opětovné povolení při jejím opuštění
    - Nedojde k přepnutí na jiný proces
  - Nevhodné, jelikož uživatelský proces zasahuje do běhu OS. Další problém je, že nemusí dojít k opětovnému povolení přerušení
  - V případě více CPU se zákaz týká pouze konkrétního CPU, tak další proces využívající jiný CPU může vstoupit do KS

- ## Zamykací proměnná
  - Ochrana KS pomocí sdílené zamykací proměnné „lock“
    - Lock = „0“ -> žádný proces není v KS
      - Proces vstupující do KS nastaví lock na „1“
    - Lock = „1“ -> proces čeká na uvolnění KS
      - Proces opouštějící KS nastaví lock na „0“
  - Hrozí zde problém přepínání kontextu, jelikož souběh se přenesen na zamykací proměnnou, čímž vzniká nová KS

```
// ...

void enterCS()
{
    while(lock == 1);   // aktivni cekani

    lock = 1;   // zamykaci promenna
}

// ...

void leaveCS()
{
    lock = 0;
}
```

- ## Přesné střídání

P_0
```
while(TRUE) {
        while(turn!=0);   /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1
```
while(TRUE) {
        while(turn!=1);   /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

  - Proměnná turn určuje, který proces může vstoupit do KS
  - Turn = „0“ -> P_0 může vstoupit do KS
  - Po dokončení práce v Ks nastaví P_0 turn na „1“
    - P_1 může vstoupit do KS
    - P_0 pracuje dále, ale už ve své nekritické sekci
  - P_1 je krátký (rychlý) ve své KS, nastaví turn na „0“
    - P_0 může vstoupit do KS
    - P_1 pracuje dál, ale už ve své nekritické části kódu
  - P_1 chce vstoupit do KS, ale nemůže, proč?
    - Jaká z podmínek je porušena?
    - Proces v kritické sekci nesmí blokovat proces mimo ni

- Petersonovo řešení

```
#define N 2

int turn;
int interested[N];      // defaultni hodnota je 0

// ...

/*
    Kazdy proces pred vstupem do KS vola enterCS()
    pro overeni, zda do ni muze vstoupit.
*/
void enterCS(int process)
{
    int otherProcess = 1 - process;     // Druhy proces.

    interested[process] = 1;            // Dany proces ma zajem o KS.

    // Kdo jako posledni zavola enterCS(), nastavi tak turn!
    turn = process;

    // Overeni, zda aktualni proces muze vstoupit do KS, pokud ne, testuje.
    while((turn == process) && (interested[otherProcess] == 1));
}

// ...

/*
    Kdyz proces dokonci cinnost v KS zavola leaveCS() pro zruseni zajmu o
    ni a zpristupneni ji dalsimu procesu.
*/
void leaveCS(int process)
{
    interested[process] = 0;
}
```

  - Kombinace zamykací a kontrolní proměnné, spolu se střídáním procesů

-

Atomická instrukce

- Příklad použití instrukce tas – Motorola 68000

```
enter_cs:  tas   lock        // Kopíruj lock do CPU a nastav lock na 1
           bnz   enter_cs    // Byl-li lock nenulový,
                             //   skok na opakované testování = aktivní čekání
           ret               // Byl nulový – návrat a vstup do kritické sekce
leave_cs:  mov   lock, #0    // Vynuluj lock a odemkni kritickou sekci
           ret
```

- Příklad použití instrukce xchg – IA32

```
enter_cs:  mov   EAX, #1     // 1 do registru EAX
           xchg  lock, EAX   // Instrukce xchg lock, EAX atomicky prohodí
                             //   obsah registru EAX s obsahem lock.
           jnz   enter_cs    // Byl-li původní obsah proměnné lock nenulový,
                             //   skok na opakované testování = aktivní čekání
           ret               // Nebyl – návrat a vstup do kritické sekce
leave_cs:  mov   lock, #0    // Vynuluj lock a odemkni tak kritickou sekci
           ret
```

  - Využití zamykací proměnné a speciální atomické instrukce
  - V době přístupu k zamykací proměnné proběhne její aktualizace celá
    - Jedna nedělitelná operace, i když se skládá z více kroků
    - CPU uzamkne paměťovou sběrnici, čímž zamezí přístup dalším procesům do paměti a po skončení ji opět uvolní
  - Nutná podpora HW
  - Procesor při vykonávání instrukce uzamkne datovou sběrnici
  - Proběhne celá jako jediná operace (nedělitelná)
  - Problém aktivního čekání je neustále testování, zda může proces vstoupit do kritické sekce
  - Hrozí zde nejen plýtvaní procesorovým časem, ale také uváznuti při čekání na kritickou sekci
  - Dva procesy, různá priorita
    - H, L
    - L -> je v kritické sekci, H přijde
    - H -> připraven vstoupit
    - H -> nemůže vstoupit, protože L je v kritické sekci
    - H -> aktivní čekání

- Sleep() a Wakeup()
  - Funkce pro uspání a probuzení procesu.

- Semafory a transakce
  - Obecný synchronizační nástroj
  - Programový prostředek (datová struktura)
  - Je poskytován OS a stará se o něj jádro OS
  - Nachází se na začátku kritické sekce a využívají se 2 operace
    - Před vstupem do Ks
    - Po vykonání KS
  - Operace jsou atomické => proběhne celá
  - Implementace musí zaručit, že žádné dva procesy nebudou provádět operace nad stejným semaforem současně

- Struktura semaforu

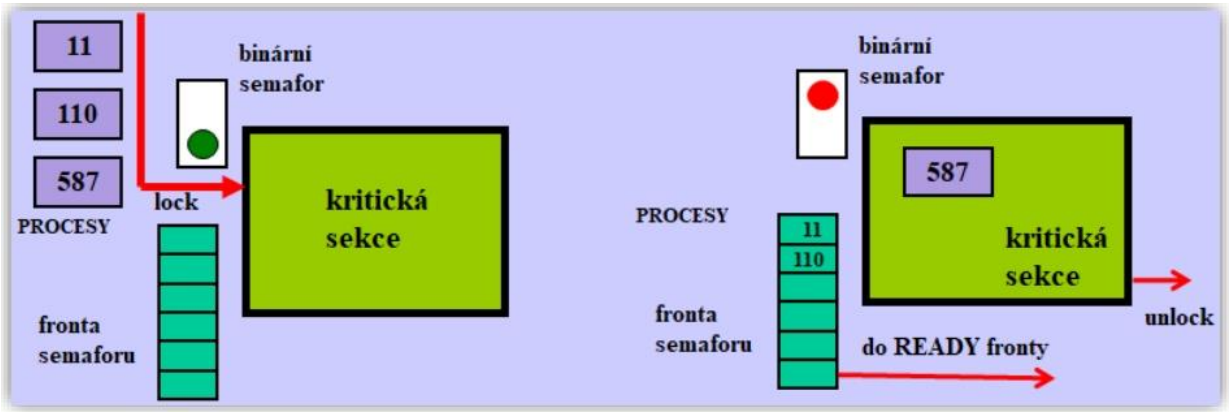
```
typedef struct {
    int value;                  // „Hodnota“ semaforu
    struct process *list;       // Fronta procesů stojících „před semaforem“
} semaphore;
```

- Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

```
void wait(semaphore S) {
    S.value= S.value - 1;
    if (S.value < 0)            // Je-li třeba, zablokuj volající proces a zařaď ho
        block(S.list);          //   do fronty před semaforem (S.list)
}
void signal(semaphore S) {
    S.value= S.value + 1;
    if (S.value <= 0) {
        if(S.list != NULL) {    // Je-li fronta neprázdná
            ...                 //   vyjmi proces P z čela fronty
            wakeup(P);          //    a probuď P
        }
    }
}
```

  - **Obecný semafor**
    - Jedná se o datovou strukturu obsahující celočíselný čítač a frontu čekajících procesů
    - Operace nad semaforem mohou provádět pouze funkce:
      - INIT()
        - Inicializuje semafor na nezápornou hodnotu, 1 – určuje, kolik procesů muže být voláno funkcí wait
      - WAIT()
        - Snižuje hodnotu čitače
        - Pokud je hodnota záporná, je proces blokován a zařazen do fronty čekajících
      - SIGNAL()
        - Zvyšuje hodnotu čitače
        - Pokud je ve frontě nějaký proces, je odblokován, může vstoupit do KS

- Binární semafor



  - Vstupující proces zavolá funkci lock pro ověření semaforu, při vstupu do KS
  - Datová struktura
    - Wait() -> lock()
    - Signal() -> unlock()
  - Místo čítače obsahuje booleovskou proměnnou defaultně inicializovanou na false
    - V KS je volno
  - První proces vstupující do KS uzamkne semafor a proces, který je poslední a již žádný další nečeká před semaforem, jej odemkne
    - Pokud existuje nějaký čekající proces, je probuzen a vstoupí do KS

5.Klasické synchronizační problémy

- Producent vs. Konzument
  - Problém omezené vyrovnávací paměti
- Čtenáři a písaři
  - Souběžnost čtení a modifikace dat
    - Přednost čtenářů -> stárnutí písařů
    - Přednost písařů -> stárnutí čtenářů
- Večeřící filozofové
  - Buď přemýšlí nebo jí (zpracovávání programu)
    - Potřeba dvou hůlek (sdílené prostředky)
  - Problém, když budou chtít jíst všichni