Operační systémy

4. Synchronizace procesů a kritická sekce
3. ročník

Synchronizace procesů

- Souběžný přístup ke sdíleným prostředkům může způsobit jejich nekonzistenci
  - Sdílená paměť, soubory, …
- Procesy je nutno koordinovat
  - Komunikace mezi procesy
    - IPC – Inter Process Communication
    - Výměna informací
- Producent vs. konzument

Producent vs. Konzument

- 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*

Kritická sekce – Critical Section

- Část zdrojového kódu, kde dochází k přístupu ke sdílenému prostředku
  - Hrozí zde přístup více procesů nebo vláken najednou

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

Kritická sekce – řešení (obecně)

- Softwarové:
  - Na aplikační úrovni
    - Celé v režii programátora
    - Základní řešení
    - Aktivní čekání *(busy waiting)*
  - Zprostředkované OS
    - Jádro OS
    - Pasivní čekání
- Hardwarové:
  - Speciální instrukce CPU
    - HW podpora
  - Aktivní čekání

Kritická sekce – řešení (konkrétně)

1. Vzájemné vyloučení s aktivním čekáním
   a) Zákaz přerušení
   b) Zamykací proměnná
   c) Přesné střídání
   d) Petersonovo řešení
   e) Atomická instrukce

2. Semafory (pasivní čekání)
   a) Obecný
   b) Binární

Zákaz přerušení

- Nejjednodušší
- Zákaz všech přerušení procesem, který vstoupí do KS a opětovné povolení při jejím opuštění
  - Nedojde k přepnutí na jiný proces
- Nevhodné:
  - Uživatelský proces zasahuje do běhu OS
  - Problém, pokud nedojde k opětovnému povolení přerušení
- V případě více CPU se zákaz týká pouze konkrétního CPU
  - Jiný proces využívající jiný CPU tak může vstoupit do KS

Zamykací proměnná

```
// ...

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

    lock = 1;    // zamykaci promenna
}

// ...

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

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

- Jaký zde hrozí problém?
  - Přepínání kontextu
  - Souběh je přenesen na zamykací proměnnou
    - Nová KS

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

Přesné střídání – nástin běhu

- turn = 0
  - P0 může vstoupit do KS
- Po dokončení práce v KS nastaví P0 turn na 1
  - P1 může vstoupit do KS
  - P0 pracuje dále, ale už ve své nekritické části kódu
- P1 je krátký (rychlý) ve své KS, nastaví turn na 0
  - P0 může vstoupit do KS
  - P1 pracuje dále, ale už ve své nekritické části kódu
    - Zde je také rychlý
- P1 chce vstoupit do KS, ale nemůže, proč?
  - Jaká z podmínek je porušena?

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;
}
```

Petersonovo řešení

- Kombinace zamykací a kontrolní proměnné, spolu se střídáním procesů
- Parametrem funkce centerCS() a leaveCS() je ID volajícího procesu
- Pokud je jeden proces v KS, druhý uvázne v testovací smyčce
- Co se stane, když dojde k současnému spuštění procesů?
  - Ve skutečnosti k současnému spuštění nedojde, vždy bude drobný časový odstup (řádově v ms/us)

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
```

Atomická instrukce

- 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 HW podpora
- Existuje více druhů implementací:
  - TAS, TSL, XCHG, …

Problém aktivního čekání

- Neustálé testování přístupu do KS
  - Plýtvání procesorového času
- Problém u rozdílných priorit:
  - Mějme 2 procesy pojmenovány H a L (dle priority)
  - Proces L vstoupí do KS
    - V průběhu zpracování se objeví proces H a chce do KS
    - Nemůže -> čeká
  - Proces L má menší prioritu než H
    - Pozastaví se a čeká na H
  - Ani jeden z procesů se nedokončí

Pasivní čekání

- Uspání a probuzení
  - Sleep() x Wakeup()
- Nedochází k plýtvání procesorového času, hrozí však jiné problémy
  - Uváznutí (deadlock)
    - Dva, či více procesů čeká na prostředek/událost, jež může uvolnit/vyvolat proces, který ale také čeká
  - Stárnutí (starvation)
    - Procesy i vyměňují přístup ke sdílenému prostředku a třetí proces se k němu nedostane
  - Aktivní zablokování (livelock)
    - Procesy si navzájem snaží vyhovět (2 lidé a úzká chodba)
  - Inverze priorit (priority inversion)
    - 3 procesy, každý s jinou prioritou (H, M, L)
    - L má sdílený prostředek, který chce i H (tento se zablokuje)
    - M prostředek nepotřebuje, jelikož má větší prioritu než L, nedovolí mu jej uvolnit

Producent vs. Konzument

- Producent generuje data do buffer, pokud je volno, jinak se uspí
  - Pokud se jednalo o první položku, probudí konzumenta
- Konzument, pokud jsou data v buffer, tak je odebírá, jinak se uspí
  - Pokud se jednalo o poslední položku, probudí producenta

```
#define N 10

int buffer[N], count = 0;

// ...

void producer() {
    while(1) {
        if(count == N) { sleep(producer); }

        buffer[count] = nextProduced;

        count++;

        if (count == 1) { wakeup(consumer); }
    }
}

// ...

void consumer() {
    while(1) {
        if(count == 0) { sleep(consumer); }

        nextConsumed = buffer[count];

        count--;

        if(count == N-1) { wakeup(producer); }
    }
}
```

Producent vs. Konzument

- buffer[N], N = 10; v count je aktuálně 0

Běžící proces	Akce	Výsledek
Konzument	Čtení count	
	R0 = count	R0 = 0
Producent	Vložení položky	buffer[count] = #data
	count++	count = 1
	Zjištění, že jde o 1. položku	wakeup(konzument)
Konzument	? R0 == 0 ?	sleep(konzument)
Producent	Vkládá položky	
	? R1 == N ?	sleep(producent)

Semafor

- Obecný synchronizační nástroj
- Programový prostředek
  - Datová struktura
- Dvě atomické operace nad semafory
  - wait() – testovat/snížit, před vstupem do KS
  - signal() – zvýšit, po vykonání KS
- Poskytováno OS
  - O správu se stará jádro OS
  - Musí být zabezpečeno, že žádné dva procesy nebudou provádět operace P a V se stejným semaforem současně

Implementace obecného semaforu

- Struktura semaforu

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

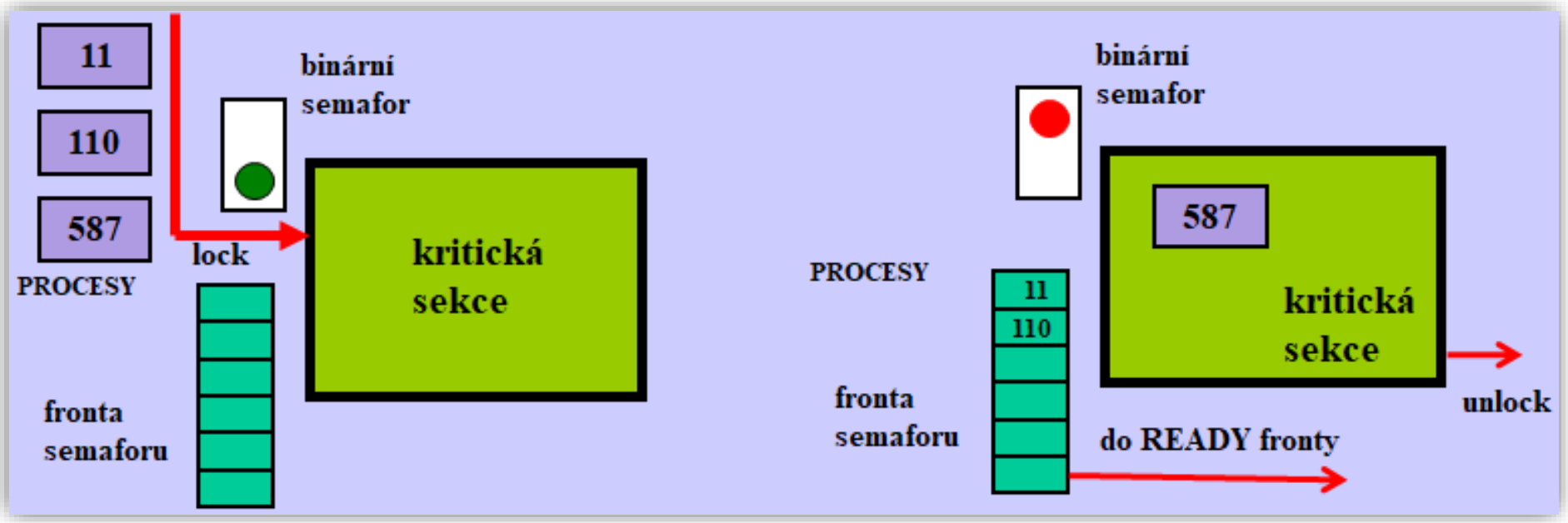
- Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

```
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

- Datová struktura obsahující:
  - Celočíselný čítač
  - Frontu čekajících procesů
- Operace nad S mohou provádět pouze tyto funkce:
  - init()
    - Inicializace semaforu na nezápornou hodnotu (kolik procesů může vstoupit do KS – volání wait() bez blokování)
  - wait()
    - Snižuje hodnotu čítače, pokud je záporná dochází k blokaci procesu a zařazení do fronty
    - Záporná, respektive absolutní hodnota udává, kolik procesů čeká před semaforem
  - signal()
    - Zvyšuje hodnotu čítače, pokud je nějaký proces ve frontě, je z ní vyjmut a odblokován

Binární semafor – mutex



Binární semafor – mutex

- Datová struktura
  - wait() -> lock()
  - signal -> unlock()
- Místo čítače obsahuje booleovskou proměnnoudefaultně 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

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 všichni jíst
- Spící holič
  - Holič, čekárna, křeslo
  - Problém uváznutí

KONEC

Zdroje

- http://labe.felk.cvut.cz/vyuka/A3B33OSD/Tema-05-IPC+Deadlock-OSD-4.pdf [11. 5. 2020]
- https://cw.fel.cvut.cz/old/_media/courses/b4b35osy/lekce04.pdf [17. 4. 2021]
- https://zcu.arcao.com/kiv/zos/zos/OdSobi/Materialy/Buris/dalsi-materialy/operacni_systemy-01/part05.html [17. 4. 2021]