KIV/ZOS 2003/2004
Přednáška 3

Časový souběh může nastat v mnoha podobných situacích:

* při přidávání prvku do seznamu apod., kdy může dojít k nekonzistencím ve výsledné datové struktuře
* 2 procesy chtějí vytvořit soubor a zapsat do něj
  . 1. proces: zjistí, že soubor není
  . 2. proces: zjistí, že soubor není, vytvoří, zapíše
  . 1. proces: pokračuje, tj. vytvoří a zapíše, čímž přepíše co zapsal druhý proces.

Hledání časových souběhů v reálných programech není jednoduché, protože časový souběh se obvykle projevuje nedeterministicky a programy běží po většinu času bez problémů.

Řešení časového souběhu
....

Časový souběh by nenastal, pokud by čtení+modifikace proběhly atomicky, tj. jako jedna nedělitelná operace. Zařídit HW většinou není praktické.

Budeme hledat SW řešení - časovému souběhu zabráníme tak, že v jednom okamžiku dovolíme číst a zapisovat společná data pouze jednomu procesu (ostatním procesům v tom musíme zabránit).

Problém přesněji:

* máme několik sekvenčních procesů, které spolu mohou komunikovat přes společnou datovou oblast,
* místo v programu, kde je prováděn přístup ke společným datům nazveme "kritická sekce" (critical section, critical region)

Řešíme úlohu, jak implementovat procesy tak, aby k jednu chvíli byl v kritické sekci pouze jeden z nich.

Poznámka (co může být "společná datová oblast")

  ''Společnou datovou oblastí'' nemusí být pouze hlavní paměť, může jí být i soubor. I při práci se souborem nastává tentýž problém, pokud jeden proces pracuje s jinou hodnotou, než jakou předpokládá druhý proces. U souborů se problém časového souběhu řeší pomocí zamykání částí souboru.

[]

Poznámka (kritická sekce se vztahuje k datům)

 Každá kritická sekce se vztahuje ke konkrétním datům, ke kterým se v ní přistupuje.

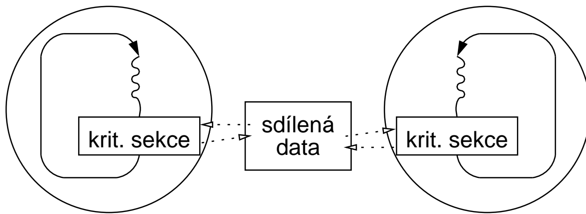
[]

Dále budeme z ilustrativních důvodů předpokládat, že procesy jsou cyklické, např.:

```
cobegin
  P1: while true do        // nekonečná smyčka
      begin
        nevinná_činnost;   // pracuje pouze s vlastními daty
        kritická_sekce     // přístup do sdílených dat
      end
  ||
  P2: ...                  // totéž co P1
coend
```

Při běhu obou procesů se tedy nevinná_činnost střídá s vykonáváním

kritické_sekce, proces který chce vstoupit do kritické sekce s tím bude
muset počkat, až jiný proces KS opustí.



Dobré řešení musí mít 3 vlastnosti ("pravidla"):

1. Vzájemné vyloučení: žádné dva procesy nesmějí být současně uvnitř
   své kritické sekce
2. Proces běžící mimo svou kritickou sekci nesmí blokovat jiné procesy
   (např. jim bránit ve vstupu do kritické sekce)
3. Žádný proces nesmí na vstup do své kritické sekce čekat nekonečně
   dlouho (např. proto, že jiný do ní vstupuje opakovaně, nebo proto,
   že se se neumí dohodnout v konečném čase, kdo vstoupí jako první).

Zakázání přerušení
....

* v systému se sdílením času přepíná CPU mezi procesy pouze jako
  důsledek přerušení
* zakážeme-li přerušení, k přepínání nedochází:

```
        zakaž_přerušení;
        kritická_sekce;
        povol_přerušení;
```

* nejjednodušší řešení, možné pouze v jednoprocesorovém systému
* není dovoleno v uživatelském režimu (co kdyby uživatel zakázal
  přerušení a už nepovolil?)

=> používáno často uvnitř jádra OS, ale není vhodné pro uživatelské
   procesy

Řešení s aktivním čekáním

SW řešení problému vzájemného vyloučení, která si uvedeme, budou
postavena na těchto základních předpokladech o systému:

1) zápis a čtení ze společné datové oblasti jsou nedělitelné operace;
   simultánní reference (čtení/zápis) ke stejné oblasti více než jedním
   procesem povede k sekvenčním odkazům v neznámém pořadí.
2) kritické sekce nemohou mít přiřazenou prioritu
3) relativní rychlost procesů je neznámá
4) proces se může pozastavit mimo kritickou sekci.

Poznámka:

  Bod (1) praxi platí, dokud přistupujeme k datum kratším nebo rovným
  délce slova (zapotřebí je jeden přístup do paměti). V některých RT OS
  neplatí (2), tím se ale nebudeme v tomto předmětu zabývat (o
  problémech s tím souvisejících se pouze zmíním).

[]

Pokud by nám stačilo, že do kritické sekce budeme přistupovat střídavě,
by možné problém vyřešit jednoduše:

```
--------------------------------------------------
program striktní_střídání; // Porušuje naše pravidlo č. 2
var turn: integer;         // (uvádím pouze z pedagogických důvodů)
```

```
begin
    turn := 1;

    cobegin
    P1: while true do
        begin
             while turn = 2 do; // čekací smyčka
             KS1;               // kritická sekce
             turn := 2
        end
    ||
    P2: while true do
        begin
             while turn = 1 do; // čekací smyčka
             KS2;               // kritická sekce
             turn:= 1
        end
    coend
end
--------------------------------------------------
```

* na počátku turn=1
* P1 zjistí, že turn=1, vstoupí do KS1
* P2 zjistí, že turn=1, a čeká ve smyčce, dokud se turn nezmění

Poznámka:

Průběžné testování proměnné (ve smyčce) dokud nenabude očekávanou hodnotu nazýváme {aktivní čekání}. Většinou se mu snažíme vyhnout, protože plýtvá časem CPU. Ve skutečných OS se používá pouze pokud můžeme předpokládat, že čekání bude krátké ("spin lock").

[]

* po dokončení KS1 se nastaví turn=2
* proces P2 vypadne ze smyčky while a vstoupí do KS2
* po dokončení KS2 nastaví turn=1, to umožní vstoupit P1 do KS1

Pokud je proces P2 podstatně rychlejší než P1, nemůže do KS2 vstoupit 2x za sebou, přestože P1 není v KS1

=> Není seriózní kandidát na řešení, protože porušuje pravidlo 2.

Petersonovo řešení
....

První úplné a funkční SW řešení navrhl Dekker, ale je poměrně složité. Jednodušší a elegantnější algoritmus navrhl Peterson (1981), jeho řešení pro dva procesy si uvedeme:

```
--------------------------------------------------
program petersonovo_řešení;

var turn:       integer;
    interested: array [0..1] of boolean; // na začátku {false, false}

procedure enter_CS(process: integer);
var other: integer;
begin
     other:=1-process;                    // ten druhý proces
     interested[process]:=true;           // oznámí zájem o vstup
     turn:=process;                       // nastaví příznak
     while turn=process and interested[other]=true do
          ;
end;
```

```
procedure leave_CS(process: integer);
begin
     interested[process]:=false;         // oznámí odchod z KS
end;

begin
     interested[0]:=false; // inicializace
     interested[1]:=false;
     cobegin
            while true do {cyklus - vlákno 1}
            begin
                 enter_CS(0);
                 KS1;
                 leave_CS(0);
            end {while}
     || {vlákno 2 neuvádím z důvodu šetření papírem}
     coend
end.
--------------------------------------------------
```

Jak to funguje:

```
* na začátku není v KS žádný proces
* např. proces 0 volá enter_CS(0)
 - nastaví interested[0]:=true, turn:=0
 - protože interested[1]=false, nebude se čekat ve smyčce
* pokud proces 1 volá enter_CS(1)
 - nastaví interested[1]:=true, turn:=1
 - bude čekat ve smyčce, dokud se interested[0] nenastaví na false, v
   našem případě voláním procedury leave_CS(0)
* co kdyby oba procesy volaly enter_CS téměř současně?
 - oba nastaví interested na true
 - oba nastaví turn na své číslo; "téměř souběžný zápis" se ale provede
   sekvenčně (viz předpoklad 1 výše), tj. nejdříve nastaví turn jeden,
   hodnota bude přepsána druhým
 - např. proces 1 jako druhý, tedy turn=1
 - oba se dostanou do while, proces 0 projde, proces 1 aktivně čeká.
```

Poznámka pro zajímavost (zobecnění Petersonova řešení)

Petersonovo řešení může být zobecněno pro N procesů. Toto zobecnění lze nalézt např. v [Stallings 1998, str. 246].

[]

Spin-lock s instrukcí TSL
....

Něco jednoduchého a koncepčního - co kdybychom měli proměnnou "zámek"?

```
 - na počátku 0
 - proces který chce vstoupit do KS otestuje
  . pokud 0 nastaví na 1 a vstoupí do KS
  . pokud 1 čeká
 - problém časového souběhu:
  . jeden proces přečte, vidí 0
  . druhý proces je naplánován, přečte, vidí 0, nastaví na 1, vstoupí
    do KS
  . po naplánování první zapíše 1, a máme dva procesy v KS
```

=> řešení vyžaduje {HW podporu}

```
* většina současných počítačů má instrukci, která otestuje hodnotu a
  nastaví paměťové místo v jedné nedělitelné operaci
* "teoretická" operace nazývána Test and Set Lock => TSL nebo TS:

   TSL R, lock

* vlastnosti instrukce TSL:
 - R je registr CPU
```

- lock je buňka paměti, bude obsahovat buď 0 (false) nebo 1 (true)
- lock budeme považovat za boolean
- instrukce TSL bude provádět (zapsáno ve fiktivním assembleru):

```
LD R, lock   ;; R <- lock
LD lock, 1   ;; x <- true
```

- nedělitelně = atomicky = žádný proces nemůže k "lock" přistoupit do dokončení instrukce TSL
- v případě víceprocesoru zamkne paměťovou sběrnici po dobu provádění instrukce

Pomocí TSL můžeme "zámek" implementovat takto:

```
;; Procesy musejí volat spin_lock před vstupem do KS,
;; a spin_unlock po opuštění KS.
;;
spin_lock:
           TSL R, lock    ;; atomicky provede R:=lock a lock:=1
           CMP R, 0       ;; byla v lock 0?
           JNE spin_lock  ;; pokud nebyla (R<>0), byl zámek nastaven -> cyklus
           RET            ;; návrat, tj. vstup do KS

spin_unlock:
           LD lock, 0     ;; ulož hodnotu 0 do lock
           RET
```

* cyklus v spin_lock se bude provádět, dokud lock=1
* ve chvíli, kdy někdo vyvolá spin_unlock přečtu 0 a mohu vstoupit do KS
* pokud na vstup do KS čeká více procesů, hodnotu 0 přečte jenom jeden z nich (první kdo vykoná TSL).

Poznámka, nebo spíš domácí cvičení:

V jádře operačního systému Linux je spin-lock implementován takto:

```
spin_lock:                ;; druhá implementace (podle Linuxu)
           TSL R, lock    ;; atomicky provede R:=lock a lock:=1
           CMP R, 0       ;; byla v lock hodnota 0?
           JE cont        ;; pokud byla (R=0), skočíme na cont
loop:      CMP lock, 0    ;; je lock=0?
           JNE loop       ;; pokud není, skočíme na loop
           JMP spin_lock  ;; pokud je, skočíme na spin_lock
cont:      RET            ;; návrat, tj. vstup do KS
```

Operace spin_unlock je implementována stejně jako v předchozím případě. Čím se liší operace spin_lock, resp. v čem může být tato druhá implementace výhodnější?

[]

Pokud instrukce typu TSL není k dispozici:

* na jednoprocesorových systémech můžeme nedělitelnost zajistit zakázáním přerušení po dobu operace (DI/EI, CLI/STI)
* ve víceprocesorových systémech se užívá primitivních operací s uzamčením sběrnice

Např. na i8086 se implementuje jako:

```
MOV AL, 1        ; do AL 1 (= true)
LOCK XCHG AL, X  ; zamkne sběrnici pro operaci XCHG, zamění AL a X
```

Poznámka:

Pro zjednodušení zápisu algoritmů budeme občas operaci TSL používat jako boolovskou fci v Pascalu; v BACI by jí bylo možno zapsat jako:

```
atomic function TSL(var x: boolean): boolean;
begin
    TSL := x;
    x := true;
end;
```

Pak bychom implementaci spin-locku s aktivním čekáním zapsali nějak takto:

```
type lock = boolean;

procedure spin_lock(var m: lock);
begin
     while TSL(m) do; { čeká pokud m=true }
end;

procedure spin_unlock(var m: lock);
begin
     m := false
end;
```

[]

Poznámka 2:

V literatuře se vyskytuje jak možnost, že instrukce TSL nastavuje true, tak možnost, že TSL nastaví false. Jak by se změnila implementace mutex_lock a mutex_unlock, pokud TSL nastaví paměť na hodnotu false?

[]

Problémy řešení s aktivním čekáním

Petersonovo řešení i spin-lock fungují, ale mají podstatné nevýhody:

* ztracený čas CPU

  V době kdy proces je v KS, jiný proces může běžet ve smyčce a přistupovat ke společným proměnným => krade paměťové cykly aktivnímu procesu; pokud sdílí CPU, konzumuje čas bez toho, aby něco dělal.

* problém inverze priorit (toto je jediné místo, kde pracovně upustím od předpokladu, že synchronizující se procesy nemohou mít přiřazenou prioritu)

Např. pokud bychom měli dva procesy, jeden s vysokou prioritou H a druhý s nízkou L, přičemž H by se spustil jakmile bude připraven. Pak může nastat:

- L je v kritické sekci
- H se stane připravený (např. dostane vstup)
- H začne aktivní čekání
- L ale nebude už nikdy naplánován, nemá šanci dokončit KS

=> H bude aktivně čekat donekonečna.

Nazývá se problém inverze priorit (priority inversion problem).

Proto se hledala primitiva, která proces zablokují, místo aby čekal aktivně.

Semafory

Dijkstra (1965) navrhl primitivum, které zjednodušuje komunikaci a synchronizaci procesů - semafory.

Nejdříve si popíšeme abstraktní semafor, pak jak se implementuje.

* semafor = proměnná, obsahuje nezáporné celé číslo
* semaforu lze přiřadit hodnotu pouze při deklaraci
* nad semafory pouze operace P(s) a V(s):

```
  operace P(S):
    pokud S>0, sníží S o 1
    jinak pozastaví proces, který chtěl provést operaci P.

  operace V(sem):
    pokud je nad semaforem S zablokovaný jeden nebo více procesů,
       vzbudí jeden z procesů; proces pro vzbuzení je vybrán náhodně
    jinak zvýší S o 1
```

* operace P i V jsou nedělitelné (atomické) akce, tj. jakmile započne operace nad semaforem, nikdo k semaforu nemůže přistoupit, dokud operace neskončí nebo se nezablokuje
* když se několik procesů pokouší přistoupit současně ke stejnému semaforu, operace se provedou sekvenčně v libovolném pořadí

[Ilustrace semaforů: most který unese pouze 3 auta.]

Poznámka pro zajímavost:

V některých učebnicích, programovacích jazycích atd. je P(s) nazýváno wait(s) nebo down(s) a V(s) je nazýváno signal(s) nebo up(s). My se budeme držet původního Dijkstrova pojmenování P (z holandského "proberen"=otestovat) a V (verhogen=zvětšit).

[]

Vzájemné vyloučení pomocí semaforů - přímé řešení:

* vytvoříme semafor s počáteční hodnotou 1
* před vstupem do KS P(s), po vystoupení V(s)
* je-li libovolný proces v KS, je s=0, jinak s=1

```
----------------------------------------------------------------
var s: semaphore = 1;
cobegin
    while true do
    begin
        ...
        P(mutex);
        KS1;
        V(mutex);
        ...
    end
    || {totéž druhý proces}
coend
-----------------------------------------------------------------
```

Semafory s počáteční hodnotou 1 se dají použít jako výše uvedený mutex. Na rozdíl od mutexu se však semafory dají využít i pro řešení jiných problémů než jen pro vzájemné vyloučení - proto si něco povíme o problematice kooperace procesů.

Kooperace procesů

Problém kritické sekce nastává v situacích, kdy:

- procesy soupeří o zdroj
- ke zdroji nemůže přistupovat více než jeden proces v daném čase
- každý proces může existovat bez ostatních, interakce je nutná pouze pro zajištění serializace přístupu ke zdroji.

Pokud procesy chtějí kooperovat na řešení společného problému, nastává

odlišná situace:

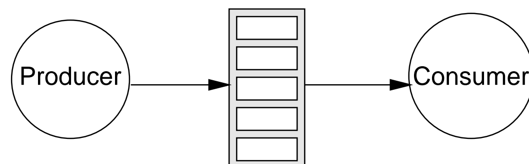
- procesy se navzájem potřebují, potřeba vzájemné výměny informací
- v nejjednodušším případě jsou zapotřebí pouze synchronizační signály
- v mnoha případech si procesy chtějí vyměňovat i jinou informaci než synchronizační signály - např. zasílání zpráv.

Do této kategorie patří úloha producent / konzument.

Problém producent / konzument

Problém producent / konzument (producer-consumer problem), také známý pod názvem "problém ohraničené vyrovnávací paměti" (bounded buffer problem, Dijkstra 1968).

- dva procesy společnou paměť (buffer) pevné velikosti N položek
- jeden proces je "producent" - generuje (produkuje) nové položky a ukládá je do vyrovnávací paměti
- paralelně běží proces "konzument", který data vyjímá a spotřebovává



Příklad:

- hlavní program produkuje tiskovou sestavu - blok dat = stránka
- tiskový server tiskne

nebo:

- obslužný program čte data ze zařízení
- hlavní program je zpracovává.

[]

Procesy mohou běžet různými rychlostmi => musí být zabezpečeno aby nedošlo k přetečení/podtečení:

- konzument musí být schopen čekat na producenta, nejsou-li data
- producent musí být schopen čekat na konzumenta, je-li buffer plný.

Řešení problému producent / konzument pomocí semaforů

- pro synchronizaci obou procesů a vzájemné vyloučení nad kritickou sekcí můžeme použít semafory
- proces se může zablokovat operací P, jiný proces ho může vzbudit operací V

Řešení bude používat dva semafory:

- e = počet prázdných položek v bufferu dostupných producentovi (empty)
- f = počet plných položek ještě nespotřebovaných konzumentem (full)

Protože přidávání a vybírání ze společné paměti může být kritickou sekcí, přidáme semafor m pro vzájemné vyloučení.

```
var
   e: semaphore = N; // empty
   f: semaphore = 0; // full
   m: semaphore = 1; // mutex
```

```
cobegin
    while true do { producent}
    begin
        produkuj záznam;
        P(e);
        P(m); vlož do bufferu; V(m);
        V(f);
    end {while}
    ||
    while true do { konzument }
    begin
        P(f);
        P(m); vyber z bufferu; V(m);
        V(e);
        zpracuj záznam;
    end {while}
coend.
```

❇