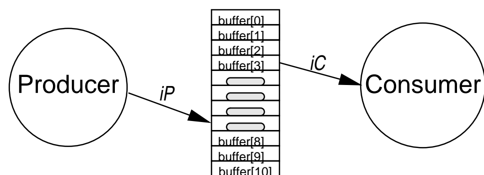
KIV/ZOS 2003/2004
Přednáška 4

Poznámka (k řešení problému producent/konzument z minula):

Vyrovnávací paměť se často implementuje jako pole - předpokládejme, že je to pole buffer[0..N-1]:



Oba procesy mají svůj vlastní index do pole buffer, např. iP (index producenta) a iC (index konzumenta). Operace "přidej do bufferu" by vypadala např. takto:

```
buffer[iP]:=položka; iP:=(iP+1) mod N;
```

Pokud je buffer implementován jako pole, vzájemné vyloučení pro přístup dvou procesů nebude potřebné, protože každý proces přistupuje pouze k těm položkám, ke kterým mu druhý proces přístup dovolí operací V(s).

[]

Domácí úkol:

Bude semafor m zapotřebí, pokud úlohu zobecníme na více producentů a konzumentů a buffer implementujeme jako pole? Pokuste se takové řešení navrhnout, včetně vložení položky do bufferu a vyjmutí položky z bufferu.

[]

Mutexy a jejich implementace

V předchozím textu jsme viděli, že někdy nepotřebujeme použít schopnost semaforů čítat, tj. stačila by nám pouze jejich schopnost zajistit vzájemné vyloučení (v podstatě totéž jako spin-lock, ale bez aktivního čekání).

OS i programovací jazyky takový mechanismus často poskytují, nazývá se "mutex" (z angl. mutual exclusion = vzájemné vyloučení), v češtině se také někdy užívá termín "paměťový zámek".

Mutexy se dají jednoduše a efektivně implementovat s malou pomocí jádra OS:

```
mutex_lock:TSL R, mutex   ;; atomicky provede R:=mutex a mutex:=1
           CMP R, 0       ;; byla v mutex hodnota 0?
           JE ok          ;; pokud byla (R=0, mutex odemčen), skáčeme na OK
           CALL yield     ;; vzdáme se procesoru - naplánuje se jiné vlákno
           JMP mutex_lock ;; zkusíme to znovu později
       ok: RET

mutex_unlock:
           LD mutex, 0     ;; ulož hodnotu 0 do proměnné mutex
           RET
```

Voláním CALL yield se volající dobrovolně vzdává procesoru ve prospěch jiných procesů. Jádro OS přesune proces mezi připravené a časem ho opět naplánuje.

Výše uvedená implementace se používá zejména pro vzájemné vyloučení mezi vlákny stejného procesu. Dá se snadno implementovat jako knihovna

programovacího jazyka.

Moderní OS (např. UNIX/Linux, Windows NT) poskytují aplikacím semafory i mutexy:

* obecné (čítající) semafory - jejich výhodou je obecnost, lze je užít i pro řešení některých problémů meziprocesové komunikace
* mutexy (paměťové zámky, binární semafory) - pouze pro vzájemné vyloučení, při vhodné implementaci efektivnější.

Poznámka (spin-lock může být někdy efektivnější)

* spin-lock je vhodný pouze pro případy, kdy je čekání krátké (několik instrukcí) a procesy běží paralelně
* není vhodné pro použití v aplikacích, protože aplikace se mohou chovat libovolně
* obvykle se používá uvnitř jádra OS, v knihovnách apod.

[]

Implementace semaforů

* procesům poskytneme možnost se pozastavit (zablokovat) při operaci P
* aktivace operací V

S každým semaforem s je sdruženo:

* celočíselná proměnná s.c; dovolíme jí nabývat i záporných hodnot, pak absolutní hodnota s.c vyjadřuje počet blokovaných procesů,
* binární semafor s.mutex pro vzájemné vyloučení při operaci nad semaforem,
* seznam blokovaných procesů s.L.

Proces, který nemůže dokončit operaci P bude zablokován a uložen do seznamu procesů blokovaných na semaforu s. Pokud při operaci V není seznam prázdný, vybere se ze seznamu jeden proces a odblokuje se.

Praktická otázka: Kam uložit datovou strukturu "semafor"?

Používají se dvě řešení:

1) Semafory apod. mohou být uloženy v jádře OS a může se k nim přistupovat pouze pomocí služeb systému.
2) Moderní OS umožňují procesům sdílet část adresního prostoru, tj. semafory jsou ve sdílené paměti.

Výše popsanou implementaci můžeme vyjádřit následujícím pseudokódem:

```
type semaphore = record
                       m: mutex;   // mutex pro přístup k semaforu
                       c: integer; // hodnota semaforu
                       L: seznam procesů
                 end;

P(s):  mutex_lock(s.m);
       s.c := s.c - 1;
       if s.c < 0 then
       begin
           zařaď volající proces do seznamu s.L;
           označ volající proces jako "BLOKOVANÝ";
           naplánuj některý připravený proces;
           mutex_unlock(s.mutex);
           přepni kontext na naplánovaný proces
       end
       else
           mutex_unlock(s.m);
```

```
V(s):  mutex_lock(s.m);
       s.c := s.c + 1;
       if s.c <= 0 then
       begin
           vyber a vyjmi proces ze seznamu s.L;
           odblokuj vybraný proces
       end;
       mutex_unlock(s.m);
```

Výše uvedený pseudokód se nezabývá detaily, kterými se bude muset zabývat programátor skutečné implementace - například organizací datových struktur a kontrolou chyb (např. je-li při operaci V záporné s.c a přitom s.L je prázdné).

Implementace v jádře OS obvykle využívá aktivní čekání (spin-lock nad s.mutex), ale pouze po dobu operace nad obecným semaforem, tj. po dobu max. desítek instrukcí, a proto by měla být efektivní.

Poznámka:

Moderní OS obvykle poskytují mutexy pro vzájemné vyloučení vláken v jednom procesu a obecné semafory pro synchronizaci mezi procesy. Proto jsem výše uvedl obvyklou implementaci mutexů (bývá implementováno jako knihovní fce, běží v uživatelském režimu) a obvyklou implementaci semaforů (implementuje jádro OS, běží v režimu jádra a má přístup k vnitřním datovým strukturám OS).

[]

Monitory
========

* problém - primitiva P a V mohou být použita kdekoli v programu
  - je možné snadno udělat chybu, není možná automatická kontrola při překladu
  - např. pokud by někdo ve výše uvedeném řešení problému producent/konzument omylem přehodil v producentovi P(e) a P(m) (oblíbená chyba ve zkouškové písemce):
    . buffer je plný, producent se zablokuje na P(e), ale m=0
    . konzument chce konzumovat, ale zablokuje se na P(m)
    . oba procesy zůstanou zablokovány (nazývá se uvíznutí)
    . pokud je konzument rychlejší, nemusí se projevit => chyba je obtížně reprodukovatelná
  - proto snaha najít primitiva vyšší úrovně, která zabrání části potenciálních chyb

* Hoare (1974) a Hansen (1973) nezávisle na sobě navrhli vysokoúrovňové synchronizační primitivum nazývané {monitor}
  - jejich návrhy se poněkud lišily, jak bude popsáno dále

Monitor je na rozdíl od semaforů jazyková konstrukce:

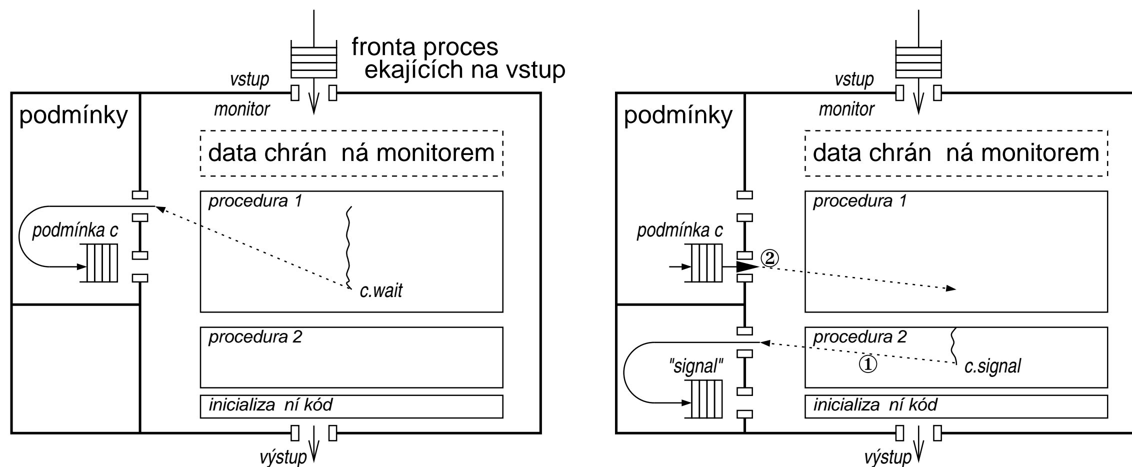
* monitor je speciální typ modulu, ve kterém jsou sdružena data a procedury, které s nimi mohou manipulovat
* procesy mohou volat procedury monitoru, ale nemohou přímo přistupovat k datům monitoru
* v monitoru může být v jednu chvíli aktivní pouze jeden proces; ostatní procesy jsou při pokusu o vstup do monitoru pozastaveny.

V OO terminologii:
 - snaha chápat kritickou sekci jako přístup ke sdílenému objektu
 - přístup k objektu pouze pomocí určených operací = metod
 - při přístupu k objektu vzájemné vyloučení, přístup po jednom

Příklad - monitory v BACI:

* monitor = Pascalský blok podobný proceduře nebo fci
* uvnitř monitoru jsou definovány proměnné, procedury a fce

```
* proměnné monitoru nejsou viditelné zvenčí, jsou dostupné pouze
  procedurám a fcím monitoru
* procedury a fce jsou viditelné a volatelné zvenčí (vně monitoru)

monitor m;
var proměnné ...
    podmínky ...

  procedure p; { procedura uvnitř monitoru }
  begin
     ...
  end;

begin
  inicializace;
end;

Použití monitoru:

* použití pro vzájemné vyloučení: kritickou sekci programu zapouzdříme do
  procedury nebo fce a přesuneme do monitoru

Např.:

    monitor m;           // příklad - vzájemné vyloučení
    var x: integer;

      procedure inc_x; { zvětší x }
      begin
         x:=x+1;
      end;

      function get_x: integer; { vrací x }
      begin
         get_x:=x
      end

    begin
       x:=0
    end; { inicializace x };

Problém:

* výše uvedená (částečná) definice dostačuje pro vzájemné vyloučení, ale
  nikoli pro synchronizaci, např. pro řešení problému producent / konzument
* konkrétně je zapotřebí mechanizmus umožňující procesu se pozastavit a
  tím uvolnit vstup do monitoru
* až s tímto mechanismem jsou monitory úplné

* synchronizace procesů v monitoru:
 - monitory poskytují speciální typ proměnné nazývané podmínka
   (condition variable)
 - podmínky mohou být definovány a použity pouze uvnitř monitoru
 - podmínky nejsou proměnné v klasickém smyslu, tj. neobsahují hodnotu
 - podmínku lze chápat spíše jako odkaz na určitou událost nebo stav
   výpočtu (mělo by se to odrážet i v názvu podmínky)

* nad podmínkami jsou definovány dvě operace, wait a signal
 - c.wait
  . volající bude pozastaven nad podmínkou c
  . pokud je některý proces připraven vstoupit do monitoru, bude mu to
    dovoleno
 - c.signal
  . pokud existuje jeden nebo více procesů pozastavených nad podmínkou c,
    reaktivuje jeden z pozastavených procesů, tj. bude mu dovoleno
    pokračovat v běhu uvnitř monitoru
  . pokud nad podmínkou nespí žádný proces, nedělá nic (na rozdíl od operace
    V(sem) nad semaforem, která si "zapamatuje" že byla zavolána)
```



Problém s operací signal:

* pokud by signal pouze vzbudil proces, běžely by v monitoru dva (vzbuzený a proces který volal c.signal), což je v rozporu s definicí monitoru ("v monitoru může být v jednu chvíli aktivní pouze jeden proces"), nutno řešit
* 2 řešení:
  - Hoare - proces volající c.signal se pozastaví, vzbudí se teprve až poté co předchozí reaktivovaný proces opustí monitor nebo se pozastaví
  - Hansen - signal smí být uveden pouze jako poslední příkaz v monitoru, po volání signal musí proces opustit monitor.

Podmínky v BACI:

* monitory podle Hoara
* waitc(cond: condition)
* signalc(cond: condition) - sémantika podle Hoara
* waitc(cond: condition, prio: integer) - čekajícímu je možné přiřadit prioritu, vzbuzen bude s nejvyšší prioritou (nejnižším číslem "prio")

[]

Poznámka z praxe (monitory v jazyce Java):

Existují i jiné varianty monitorů, např. zjednodušené monitory s primitivy wait a notify v jazyce Java a dalších jazycích:

* s každým objektem je sdružen monitor, může být i prázdný
* metoda nebo blok patřící do monitoru označena klíčovým slovem synchronized, např.:

```
class jméno {
    synchronized void metoda() {
        ....
    }
}
```

* s monitorem je sdružena jedna podmínka, metody:

```
wait()      - pozastaví volající vlákno
notify()    - označí jedno spící vlákno pro vzbuzení, vzbudí se až
              volající opustí monitor (na rozdíl od c.signal, které
              pozastaví volajícího)
notifyAll() - jako notify(), ale označí pro vzbuzení všechna spící
              vlákna
```

Výše uvedené je de facto třetí řešení problému, který Hansen a Hoare řešili odlišně (čekající může běžet až poté, co proces volající signal opustí monitor).

[]

Poznámka (zjednodušené monitory v jazyce Java)

Pokud by podmínek mohlo být více, narazili bychom na drobný problém - na rozdíl od Hoarovských monitorů:

* pokud se proces pozastavil protože hodnota proměnné B byla false, nemůže počítat s tím, že po vzbuzení bude B=true
* máme-li např. 2 procesy, zablokování nastalo:

```
proces 1: if not B1 then c1.wait;
proces 2: if not B2 then c2.wait;
```

pak proces běžící v monitoru způsobí splnění obou podmínek a oznámí to pomocí

```
if B1 then c1.notify;
if B2 then c2.notify;
```

Po opuštění monitoru se vzbudí proces 1, ten může způsobit že B2=false, čímž po vzbuzení procesu 2 bude B2 false.

Proto by v takovém případě - na rozdíl od Hoarovských monitorů - mělo být volání metody wait v cyklu:

```
while not B do
    c.wait;
```

[]

Poznámka pro zajímavost (monitory a klíčové slovo volatile v jazyce Java):

Vlákno v jazyce Java si může vytvořit soukromou pracovní kopii sdílené proměnné, kterou zapíše zpět do sdílené paměti pouze při vstupu/výstupu z monitoru. Pokud chcete aby vlákno zapisovalo hodnotu při každém přístupu k proměnné, musí být proměnná deklarována jako volatile.

[]

Shrnutí:
....

* základní varianta ("abstraktní primitivum") jsou Hoarovské monitory
* v reálných programovacích jazycích varianty
  - prioritní wait (viz např. BACI)
  - s primitivy wait a notify (např. Java, Borland Delphi apod.)
* výhoda monitorů - automaticky řeší vzájemné vyloučení, paralelní programování je odolnější proti chybám
* nevýhoda - monitory jsou koncepce programovacího jazyka => překladač musí rozpoznat zařídit.

Poznámka pro zajímavost:

Monitory jsou zajímavé, proto najdeme některé myšlenky i v rozhraní UNIXu/Linuxu pro práci s vlákny. Máme-li úsek kódu ohraničený pomocí pthread_mutex_lock(m) a pthread_mutex_unlock(m), můžeme v něm používat obdobu podmínek z monitorů:

```
pthread_cond_wait(c, m)    - atomicky odemkne m a čeká na podmínku
pthread_cond_signal(c)     - označí 1 vlákno spící nad c pro vzbuzení
pthread_cond_broadcast(c) - označí všechna vlákna spící nad c pro vzbuzení
```

[]

Řešení problému producent-konzument pomocí monitorů

```
{ Řešení problému producent-konzument pomocí monitoru.
 (Používám zde abstraktní zápis monitoru, nikoli monitory z BACI.) }

Monitor ProducerConsumer
var
   f, e: condition;
   i: integer;
```

```
procedure enter;
begin
  if i=N then wait(f);  { paměť je plná => čekám až se uvolní }
  enter_item;           { vlož položku do bufferu }
  i:=i+1;
  if i=1 then signal(e); { první položka => vzbudím konzumenta }
end;

procedure remove;
begin
  if i=0 then wait(empty);      { paměť je prázdná => čekám na data }
  remove_item;                  { vyjmi položku z bufferu }
  i:=i-1;
  if i=N-1 then signal(f); { je zase místo }
end;

begin
  i:=0; { inicializace }
end
end monitor;

begin // začátek programu
    cobegin
           while true do { producent}
           begin
                produkuj záznam;         { vymysli položku }
                ProducerConsumer.enter;  { vlož položku do bufferu }
           end {while}
           ||
           while true do { konzument }
           begin
                ProducerConsumer.remove;    { vyjmi položku }
                zpracuj záznam;             { spotřebuj jí }
           end {while}
    coend
end.
```

Otázka, resp. domácí úkol: mohlo by procedury "enter" a "remove" monitoru používat
a) víc producentů?
b) víc konzumentů?
Předpokládejme, že všichni producenti a všichni konzumenti jsou si rovni.

Implementace monitorů pomocí semaforů

Monitory musí umět rozpoznat překladač programovacího jazyka a přeložit je do odpovídajícího kódu. Zde si ukážeme, jak by takový kód mohl vypadat, pokud by OS poskytoval semafory, tj. implementujeme monitory pomocí semaforů.

Implementace musí zaručit:

1. běh procesů v monitoru musí být vzájemně vyloučen (pouze jeden v monitoru)
2. wait musí blokovat aktivní proces v příslušné podmínce
3. když proces opustí monitor nebo je blokován podmínkou a existuje jeden nebo více procesů čekajících na vstup do monitoru, musí být jeden z nich vybrán:
   - existuje-li proces pozastavený jako výsledek operace signal, pak je vybrán
   - jinak je vybrán jeden z procesů čekajících na vstup do monitoru
4. signal musí zjistit, zda existuje proces čekající nad podmínkou
   - existuje-li - aktuální proces je pozastaven a jeden z čekajících procesů bude reaktivován (např. s použitím FIFO disciplíny)
   - jinak pokračuje původní proces.

Překladač musí pro každý monitor zavést následující semafory a čítače:

```
- semafory
 . m = 1 ("m" jako mutex) - vzájemné vyloučení přístupu k procedurám v monitoru
 . u = 0 (urgent) - pro pozastavení procesu při vyvolání operace signal
 . w[i]=0 - pole semaforů, každý w[i] je pro jednu podmínku c[i];
            používá se pro pozastavení procesů, které provedly operaci
            wait

- čítače
 . ucnt=0 - čítá počet pozastavení jako výsledek operace signal
 . wcnt[i]- pole čítačů, každý wcnt[i] je čítač sdružený s podmínkou
            c[i], obsahuje počet procesů pozastavených na příslušné
            podmínce.
```

Tělo každého procesu v monitoru bude po překladu ohraničeno následujícím vstupním a výstupním kódem:

```
  P(m);            // vstupní kód - zamkneme semafor m

  ...              // tělo procedury;

                   // výstupní kód:
  if ucnt>0 then   //  je-li nějaký proces pozastavený v důsledku c.signal
      V(u)         //     dovolíme mu pokračovat
  else             //  jinak
      V(m);        //     vpustíme další procesy do monitoru
```

Každý c.wait se přeloží jako:

```
  wcnt := wcnt+1;   // zvětšíme wcnt[i], tj. počet čekajících nad c[i]
  if ucnt > 0 then  // stejný případ jako výše (dovolíme někomu pokračovat)
      V(u)
  else
      V(m);
  P(w);             // proces čeká na příslušném w[i]
  wcnt := wcnt-1;   // čekání skončilo, zmenšíme wcnt[i]
```

Každý c.signal se přeloží jako:

```
  ucnt := ucnt+1;  // aby se o nás vědělo
  if wcnt > 0 then // je někdo čekající nad podmínkou wcnt[i]?
  begin
      V(w);        // pustíme čekajícího
      P(u);        // sami musíme čekat
  end;
  ucnt := ucnt-1;
```

* přístup do monitoru je chápán jako privilegium, které si procesy předávají mezi sebou
* pokud není komu předat, semafor m se uvolní

❇