KIV/ZOS 2003/2004
Přednáška 5

Doposud uvedené mechanismy předpokládaly objekt ve sdílené paměti. To ale někdy není vhodné nebo možné:

* není vhodné: např. bezpečnost - globální data jsou přístupná kterémukoli procesu bez ohledu na semafor (tj. lepší bude, aby data nebyla globální)
* není možné: procesy běží na různých strojích, které spolu komunikují po síti.

V takových případech se používá mechanismus předávání zpráv (message passing) nebo další mechanismy na něm postavené.

Předávání zpráv - primitiva send a receive

Zavedeme 2 primitiva:

* send(adresát, zpráva) - odeslání zprávy
* receive(odesilatel, zpráva) - příjem zprávy

Základní tvar primitiv by bylo možné popsat takto:

* primitivum send způsobí, že ”zpráva” (= libovolný datový objekt) bude zaslána adresátovi
* primitivum receive provede příjem zprávy od určeného odesilatele; přijatá zpráva se uloží do proměnné ”zpráva”.

Na rozdíl od semaforů a apod. existuje celá řada variant, které se liší chováním.

Základní rozlišení je podle následujících kritérií:

1. Když je zpráva primitivem send vysílána, čeká send na přijetí zprávy nebo se po odeslání vrátí a proces může pokračovat v činnosti?
2. Co se stane, pokud při zavolání receive není ve frontě žádná zpráva?
3. Musí se ve volání send specifikovat jeden příjemce nebo je možné specifikovat celou skupinu příjemců?
4. Musí se v primitivu receive specifikovat jeden odesilatel nebo je možné přijímat zprávy pocházející od různých odesilatelů?

Podle odpovědi na otázky 1 a 2 dělíme na primitiva na blokující a neblokující (synchronní a asynchronní), podle odpovědi na otázky 3 a 4 dělíme pojmenování komunikujících na explicitní a implicitní.

1. Čeká-li primitivum send na převzetí zprávy příjemcem, je send {blokující}, někdy se používá termín {synchronní}.

   Ve většině systémů je send neblokující, tj. send se vrací ihned po odeslání zprávy.

2. Pokud při zavolání receive není ve frontě žádná zpráva, může se receive buď zablokovat (blokující receive) nebo se může vrátit s chybou (neblokující receive).

   Ve většině systémů je receive blokující, tj. čeká na doručení zprávy.

Poznámka:

Kromě blokujícího receive bývá někdy dispozici ještě varianta s omezeným čekáním receive(vysílač, zpráva, t), které čeká na příchod zprávy určenou dobu t. Pokud zpráva v určeném čase nepřijde, vrací se volání s chybou.

[]

3. Můžeme-li poslat zprávu skupině procesů, nazýváme tuto možnost multicast (skupinové adresování). Zprávu obdrží každý proces ve skupině. Posíláme-li zprávu ”všem” procesům (tj. více nespecifikovaným příjemcům), nazýváme to broadcast (všesměrové vysílání).

4. Receive které umožňuje příjem od více procesů se nazývá receive s implicitním pojmenováním.

Většina skutečných systémů umožňuje odeslání zprávy skupině procesů a příjem zprávy od kteréhokoli procesu.

Další důležité otázky:

* vlastnosti fronty zpráv, např. kolik zpráv může fronta obsahovat?
* co se stane, je-li v době pokusu odeslat zprávu fronta zpráv plná? (Většinou je odesilatel pozastaven.)
* v jakém pořadí jsou zprávy doručeny? (Většinou v pořadí FIFO.)
* jaké je zpoždění mezi odesláním zprávy a možností zprávu přijmout?
* jaké mohou v systému nastat chyby, např. mohou se zprávy ztrácet?

Volbu konkrétního chování primitiv send a receive provádějí návrháři systému, některé systémy nabízejí několik alternativních primitiv send a receive s různým chováním.

Poznámka (terminologický zmatek):

V některých systémech se pojmem ”neblokující alias asynchronní send” myslí takový send, který se vrací ihned, dokonce ještě před odesláním zprávy. Odeslání zprávy se pak provádí paralelně s další činností procesu. Toto schéma má pochopitelné nevýhody, proto se používá zřídka.

[]

V dalším textu budeme předpokládat následující chování primitiv send a receive:

* send je neblokující, receive je blokující
* receive umožňuje příjem od libovolného adresáta (odesilatel=ANY): tj. receive(ANY, zpráva) znamená požadavek na příjem zprávy od libovolného odesilatele
* fronta zpráv je dostatečně velká, abychom do ní mohli zaslat všechny zprávy, které potřebujeme
* zprávy jsou doručeny v pořadí FIFO a neztrácejí se.

Příklad:

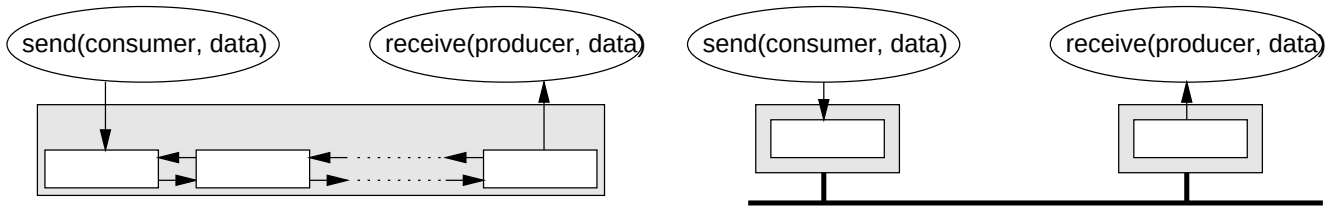
Pomocí zpráv vyřešíme problém producent/konzument.

K tomu se nám hodí na problém producent/konzument nahlížet jako na symetrický problém, kde producent generuje plné položky pro spotřebování konzumentem, zatímco konzument generuje prázdné položky pro využití producentem.

```
cobegin
    while true do { producent }
    begin
        produkuj záznam;
        receive(konzument, m);   // čeká na prázdnou položku
        m := záznam;             // vytvoří zprávu
        send(konzument, m);      // pošle položku konzumentovi
    end {while}
    ||
    for i:=1 to N do
       send(producent, e);       // na začátku pošleme N prázdných položek
    while true do { konzument }
    begin
        receive(producent, m);   // přijme zprávu obsahující data
        záznam := m;
```

```
        send(producent, e);      // prázdnou položku pošleme zpět
        zpracuj záznam;
    end {while}
coend.
```

Podstatnou výhodou mechanismu zpráv je to, že procesy nemusejí být na stejném stroji, ale mohou komunikovat po síti - z hlediska komunikujících procesů v tom není rozdíl.



Mailboxy (schránky) a porty

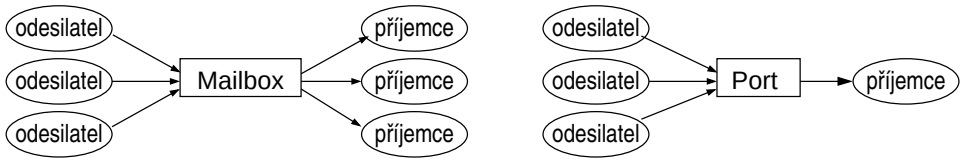
Zatím jsme předpokládali, že posíláme zprávy procesům. Problém je, jak určit adresáta, tj. jak procesy pojmenovat? Procesy nejsou ”trvalé” entity, v systému vznikají a zanikají, můžeme mít více instancí stejného programu.

Řešením je zviditelnit fronty zpráv a použít nepřímou komunikaci:

* proces pošle zprávu = zpráva se připojí k určené frontě zpráv
* jiný proces přijme zprávu = vyjme zprávu z určené fronty zpráv

Fronta zpráv, která může být využívána více odesilateli a více příjemci se nazývá {mailbox}. Mailbox je velmi obecné schéma, avšak operace receive může být pro mailbox drahá, zejména pokud procesy přijímající zprávy z mailboxu běží na různých strojích.

Proto se většinou používá omezená forma mailboxu, nazývaná {port}, ze které může přijímat zprávy pouze jeden příjemce.



Implementace mechanismu zpráv

Zde nastává spousta zajímavých problémů, které nenastanou u semaforů ani monitorů, zvláště pokud komunikující procesy běží na různých strojích propojených sítí.

Co kdyby se zpráva ztratila?

* obvykle jakmile je zpráva přijata, posílá se zpět potvrzení (acknowledgement)
* pokud vysílač nedostane potvrzení do nějakého časového okamžiku, zprávu pošle znovu

Co když zpráva došla v pořádku, ale potvrzení se ztratilo?

* každá zpráva má obvykle číslo, duplicitní zprávy se ignorují

Problém autentizace - jak obě strany vědí, že komunikují s kým chtějí a ne s podvodníkem?

* zprávy je možné šifrovat, klíč bude známý pouze autorizovaným uživatelům

(resp. jejich procesům)
* každá zašifrovaná zpráva by měla obsahovat nějakou redundanci, která umožní detekovat změnu zašifrované zprávy
* konkrétní mechanismy a problémy jsou složitější (viz KIV/BIT)

Běží-li odesilatel i příjemce na stejném stroji, můžeme snížit cenu zasílání zpráv?

* jednoduchá implementace by prováděla dvojí kopírování (z procesu odesilatele do fronty v jádře, z jádra do procesu příjemce)
* řešením např. mechanismus rendezvous - eliminujeme frontu zpráv:
 - je-li send zavolán dříve než receive, odesilatel je zablokován
 - ve chvíli kdy je vyvolán send i receive, je možné zprávu zkopírovat z odesilatele přímo do příjemce
 - efektivnější, ale méně obecné
* další způsoby využívají zejména mechanismy virtuální paměti (paměť obsahující zprávu je ”přemapována” z procesu odesilatele do procesu příjemce, tj. zpráva se nekopíruje)

Jak vypadá typická aplikace využívající zprávy?

* aplikace má typicky strukturu klient/server:
 - jedna množina procesů (klienti) požaduje vykonat práci
 - jiná množina procesů (servery) práci vykonává

Nejčastěji aplikace (klienti) vyžadují od serveru data nebo provedení nějaké operace nad daty, například:

klient	server
.....	.....
WWW klient (MSIE)	WWW server (Apache)
účetní aplikace	databázový systém (Oracle, MySQL)
klientský OS	souborový server (AFS)

Typický průběh komunikace:

* klient pošle zprávu obsahující požadavek (např. ”chci hodnotu proměnné x”)
* server odpoví (např. ”125”)

Volání vzdálených procedur (RPC)

Používání send/receive je opět nestrukturované: programátor používá send a receive bez ohledu na strukturu programu (procedury apod.).

Problém byl dlouho znám, ale až Birell a Nelson (1984) přišli se zajímavým řešením: dovolit procesům (klientům) volat procedury umístěné v jiném procesu (serveru). Tomuto mechanismu se říká volání vzdálených procedur (Remote Procedure Call, RPC).

Poznámka:

Variantou RPC je i volání vzdálených metod (Remote Method Invocation, RMI), které se používá v objektově orientovaných jazycích, např. v Javě.

[]

Jak to provést, aby se volání vzdálené procedury co nejvíce podobalo normálnímu volání podprogramu?

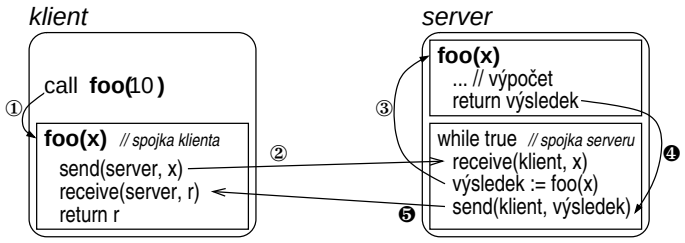
* klientský program musí být sestaven s knihovní fcí, nazývanou spojka klienta (client stub) - reprezentuje vzdálenou proceduru v adresním prostoru klienta (má stejné jméno, počet a typ argumentů jako vzdálená procedura)
* program serveru je sestaven se spojkou serveru (server stub).

Spojka klienta a spojka serveru zakrývají, že volání není lokální.

Komunikace mezi klientem a serverem probíhá v těchto krocích:

1. Klient zavolá spojku klienta, reprezentující požadovanou vzdálenou proceduru.
2. Spojková procedura argumenty zabalí do zprávy a zprávu pošle serveru
3. Spojka serveru zprávu přijme, vyjme z ní argumenty a zavolá proceduru
4. Procedura se vrátí, návratovou hodnotu pošle spojka serveru zpět klientovi
5. Spojka klienta přijme zprávu obsahující návratovou hodnotu a návratovou hodnotu vrátí volajícímu.

Pro ilustraci si ukážeme konkrétní příklad, a to volání vzdálené procedury foo(x: integer): integer.



1. Klient volá spojku klienta foo(x) s argumentem x=10.
2. Spojka klienta vytvoří zprávu a pošle jí serveru:

```
procedure foo(x: integer):integer;
begin
   send(server, m);    // zpráva obsahuje argument, tj. hodnotu ”10”
```

3. Server přijme zprávu a volá vzdálenou proceduru:

```
receive(klient, x); // spojka přijme zprávu, tj. hodnotu ”10”
výsledek = foo(x);  // spojka volá fci foo(10)
```

4. Procedura foo(x) provede výpočet a vrátí výsledek.
5. Spojka serveru výsledek zabalí do zprávy a pošle zpět spojce klienta:

```
send(klient, výsledek);
```

6. Spojka klienta výsledek přijme, vrátí ho volajícímu (jako kdyby ho spočetla sama):

```
receive(server, výsledek);
foo = výsledek;
return;
```

Ve výše uvedeném příkladu umí server vyvolat jedinou proceduru. Pokud server obsahuje více procedur (což je časté), rozlišíme je číslem. Spojka klienta by ve zprávě předala kromě parametrů také číslo požadované procedury, a spojka serveru by podle čísla požadované procedury rozlišila, kterou proceduru má vyvolat. Spojka serveru by v takovém případě obsahovala skeleton následujícího typu:

```
while true do
begin
     receive(klient, m);   // přijmi zprávu obsahující č. procedury a parametry
     if (m.číslo_procedury = 1) then výsledek = foo(m.x);
     if (m.číslo_procedury = 2) then výsledek = bar(m.x);
     ...
```

```
     send(klient, výsledek); // odešli zpět návratovou hodnotu
end
```

RPC je dnes nejpoužívanější jazyková konstrukce pro implementaci distribuovaných systémů a distribuovaných programů bez explicitního předávání zpráv (DCE RPC, Java RMI, CORBA ...).

Programování RPC:

* ve speciálním jazyce (Interface Definition Language, IDL) definujeme rozhraní mezi klientem a serverem (datové typy, procedury),
* kompilátor jazyka IDL vygeneruje spojky pro klienta i pro server
* program resp. procedury serveru sestavíme se spojkou serveru
* spojka klienta má podobu knihovny, se kterou je možné sestavovat klientské programy.

Problémy RPC:

* parametry předávané odkazem
  - klient a server jsou v různých adresních prostorech, odeslání ukazatele by nemělo smysl
  - pokud se odkazujeme na jednoduchý datový typ, na záznam nebo na pole, můžeme použít následující trik:
    . spojka klienta pošle odkazovaná data spojce serveru
    . spojka serveru vytvoří nový odkaz na data, předá volané proceduře
    . po návratu z procedury pošle spojka serveru modifikovaná data zpět
    . spojka klienta přepíše původní data
* globální proměnné
  - zatímco normální procedury mohou komunikovat pomocí globálních proměnných, při volání vzdálené procedury to není možné.

Další problém je společný pro předávání zpráv i pro RPC - problém reprezentace informace:

* pokud procesy běží na strojích různých architektur, může se lišit vnitřní reprezentace datových typů:
  - řetězce - kódování
  - numerické typy
    . liší se způsob uložení (little endian, big endian)
    . liší se velikost (např. integer může být 32 nebo 64 bitů, podle typu procesoru)
* řešení:
  - definujeme, jak budou data reprezentována při přenosu mezi počítači (”síťový formát”)
  - spojka před odesláním data převede do síťového formátu, po přijetí do lokálního formátu
  - problém rozdílné velikosti - definujeme novou množinu numerických typů, která bude mít stejnou velikost na všech podporovaných architekturách (např. int32_t - integer velikosti 32 bitů; viz hlavičkový soubor <stdint.h> v jazyce C podle normy ISO C99).

Ekvivalence výše uvedených primitiv

Lze implementovat semafory pomocí zpráv i zprávy pomocí semaforů, tj. mají semafory i zprávy stejnou vyjadřovací sílu?

* zprávy pomocí semaforů:

Je to řešení problému producent/konzument, pouze jinak strukturujeme - vložení zprávy umístíme do procedury send a vyjmutí zprávy do procedury receive.

* semafory pomocí zpráv:
  - zavedeme si pomocný synchronizační proces, který bude pro každý semafor udržovat čítač (hodnotu semaforu) a seznam blokovaných procesů
  - operace P a V budeme implementovat jako funkce, které provedou odeslání požadavku a poté čekají na odpověď pomocí receive

- synchronizační proces obsluhuje v jednom čase jeden požadavek, čímž je zaručena podmínka vzájemného vyloučení
- pokud synchronizační proces obdrží požadavek na operaci P a semafor>0, odpoví ihned; jinak neodpoví, čímž volajícího zablokuje
- pokud synchronizační proces obdrží požadavek na operaci V a je blokovaný proces, jednomu blokovanému procesu odpoví, čímž ho vzbudí.

Podobně můžeme ukázat, že je možné implementovat semafory pomocí monitoru, monitory pomocí semaforů apod.

Proto můžeme říci, že všechna dříve uvedená primitiva mají stejnou vyjadřovací sílu.

Otázka: Platí to i o mutexech?

Platí, semafory lze implementovat pomocí mutexů. Pro zajímavost jednu takovou implementaci uvedu (Barz 1983):

```
type semaphore = record
                     val: integer;
                     m:   mutex;   // pro vzájemné vyloučení
                     d:   mutex;   // pro blokování (delay)
                 end;

procedure Initsem(var s: semaphore, count: integer);
begin
     s.val:=count;
     s.m  :=0;                 // odemčeno
     if count=0 then s.d := 1  // zamčeno, někdo musí provést V(s)
                else s.d := 0  // odemčeno
end;

procedure P(var s: semaphore);
begin
   mutex_lock(s.d);       // pokud je s.val=0, budeme zde čekat
   mutex_lock(s.m);       // kritická sekce - přístup k proměnné s.val
   s.val := s.val - 1;    // vždy bude platit s.val>=0
   if s.val > 0 then
      mutex_unlock(s.d); // vpustíme další proces do operace P
   mutex_unlock(s.m)     // konec přístupu k proměnné val
end;

procedure V(var s: semaphore);
begin
   mutex_lock(s.m);
   s.val := s.val + 1;
   if s.val = 1 then
      mutex_unlock(s.d)
   mutex_unlock(s.m)
end;
```

Poznámka:

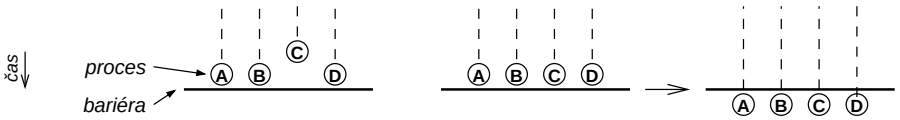
Z důvodu efektivity mají mutexy někdy omezení, že mutex smí odemknout pouze to vlákno, které předtím provedlo jeho uzamknutí (například mutexy podle standardu POSIX.1). Takové mutexy se ovšem pro implementaci obecných semaforů použít nedají, tj. jsou slabší než všechna výše uvedená primitiva.

[]

Bariéry
=======

* bariéry jsou synchronizační mechanismus pro skupiny procesů, používají se zejména v oblasti vědecko-technických výpočtů
* aplikace se skládá z fází, žádný proces nesmí do následující fáze dokud všechny procesy nedokončily fázi předchozí
* na konci každé fáze proces synchronizaci na bariéře (volá ”barrier”), ta

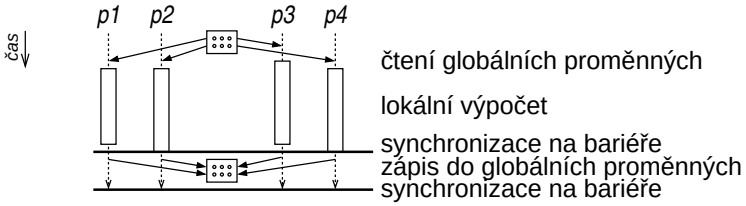
volajícího pozastaví dokud všechny spolupracující procesy také nezavolají "barrier"
* všechny procesy opustí bariéru současně.



a) procesy přicházejí na bariéru a čekají

b) po příchodu posledního procesu všechny společně projdou bariérou

Například iterační výpočty: matici Xi+1 vypočítáme z matice Xi, každý proces počítá jeden prvek nové matice. Po dokončení kroku výpočtu můžeme zahájit další iteraci, pro synchronizaci použijeme bariéru.



Klasické problémy meziprocesové komunikace

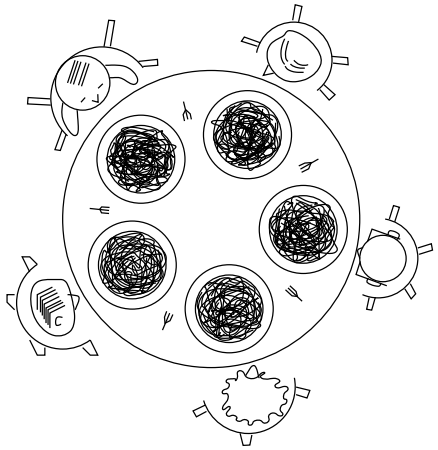
(V angličtině Interprocess Communication, zkratka IPC.)

* my jsme už viděli problém producent/konzument
* literatura o OS je plná dalších zajímavých problémů, které byly diskutovány a analyzovány

Problém večeřících filosofů

* 1965 Dijkstra předložil a vyřešil (angl. dining philosophers)
  - modeluje procesy soupeřící o výhradní přístup k omezenému počtu zařízení
  - od té doby autoři ukazují eleganci nových synchronizačních primitiv na řešení tohoto problému

* 5 filosofů sedí kolem kulatého stolu
* každý filosof má před sebou talíř se špagetami
* mezi každými dvěma talíři je vidlička
* špagety jsou tak klouzavé, že filosof potřebuje 2 vidličky, aby mohl jíst



```
* život filosofa se skládá z jedení a přemýšlení
* když filosof dostane hlad, pokusí se vzít si dvě vidličky; pokud
  uspěje, nějakou dobu jí, pak položí vidličky a pokračuje v přemýšlení
* úkolem je napsat program pro každého filosofa tak, aby pracoval podle
  předpokladu a nedošlo k potížím (tj. aby se každý najedl podle potřeby)

* první řešení - chybné:
 - očíslujeme filosofy: 0 .. 4
 - očíslujeme vidličky - pro filosofa 0 je levá 0 a pravá 1
 - procedura zvedni(v) počká, až bude vidlička v k dispozici a pak jí
   zvedne

const N = 5;

procedure filosof(i: integer);
begin
     přemýšlej;
     zvedni(i);
     zvedni((i+1) mod N);
     jez;
     polož(i);
     polož((i+1) mod N)
end;

* chyba - může dojít k uvíznutí (deadlock)
 - všichni filosofové zvednou najednou levou vidličku
 - žádný z nich už nemůže pokračovat
 - deadlock = cyklické čekání, dva nebo více procesů čeká na událost,
   kterou může způsobit další z nich (f[0] čeká až f[1] položí
   vidličku, f[1] čeká až f[2] položí vidličku, 2-3, 3-4, 4-0 = cyklus)

* modifikace programu - stále ještě chybná:
 - program zvedne levou a zjistí, zda je pravá vidlička dostupná
 - není-li, položí levou, počká nějakou dobu a zkusí znovu

* chyba - pokud by filosofové vzali najednou levou vidličku, budou
  běžet cyklicky (vidí že pravá vidlička není volná, položí...)
 - program běží bez toho, že by vykonával užitečnou činnost (situace
   "až po vás")
 - název vyhladovění (starvation) - používá se i pro jiné situace než pro
   problém večeřících filosofů

* Dijkstra ukázal řešení pomocí semaforů (které vymyslel)
* řešení pomocí semaforů je delší, ukážu řešení pomocí monitorů:

monitor večeřící_filosofové;

const N=5;

var
   f: array [0 .. N-1] of integer;   { počet vidliček dostupných filosofovi }
   a: array [0 .. N-1] of condition; { podmínka "vidličky jsou k dispozici" }
   i: integer;

procedure chci_jíst(i: integer)
begin
   if f[i] < 2 then a[i].wait;
   decrement( f[(i-1) mod N] ); // pro zkrácení zápisu; decrement(x) je x:=x-1
   decrement( f[(i+1) mod N] );
end;

procedure mám_dost(i: integer)
begin
   increment( f[(i-1) mod N] ); // zvětši o 1
   increment( f[(i+1) mod N] );
   if f[(i-1) mod N] = 2 then a[(i-1) mod N].signal;
   if f[(i+1) mod N] = 2 then a[(i+1) mod N].signal
end;
```

```
begin  // inicializace
   for i:=0 to 4 do
      f[i] := 2;
end;
```

- když je filosof(i) připraven jíst, zavolá proceduru chci_jíst(i)
- při skončení jídla zavolá mám_dost(i)

- algoritmus využívá pole 5 čítačů f[i], každý prvek sdružený s filosofem i
- f[i] říká kolik má filosof i k dispozici vidliček, na začátku f[i]=2
- filosof chce jíst, volá chci_jíst(i)
  . pokud je f[i] < 2, jeden nebo oba sousedi jedí, musíme čekat
  . jinak zmenšíme počet vidliček dostupných sousedním filosofům a dovolíme filosofovi pokračovat
- při skončení jídla volá mám_dost(i)
  . zvětší počet vidliček dostupných sousedům
  . pokud má některý soused obě vidličky, provedeme signal aby se mohl případně pustit do jídla

- ochrana před uvíznutím - obě vidličky musí zvednout ve stejném okamžiku, tj. ve stejné kritické sekci

Poznámka drobná:

Tento algoritmus má stále ještě potíž, že dva konspirující filosofové by mohli zabránit třetímu, aby se najedl, viz [Bic, pp. 86-87.].

[]

❇