



Principy programovacích jazyků

zpracoval Martin Kuba

16. května 1995

Obsah

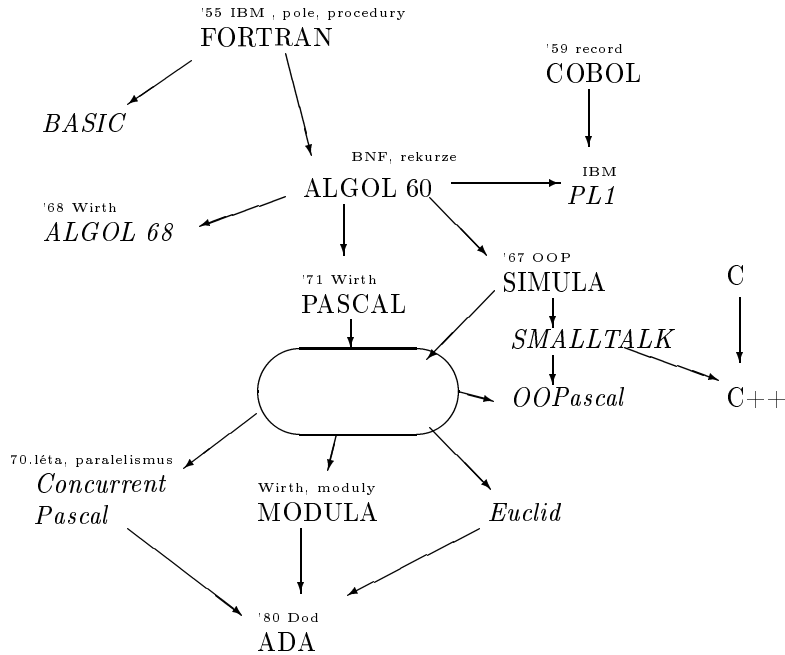
| | |
|--|-----------|
| 1 Úvod | 3 |
| 1.1 Syntaxe | 3 |
| 2 Sémantika | 4 |
| 2.1 Proměnné | 4 |
| 2.2 Datové typy | 5 |
| 2.2.1 Typy | 5 |
| 2.2.2 Podtypy | 5 |
| 2.2.3 Odvozené typy | 6 |
| 2.2.4 Skalární typy | 6 |
| 2.2.5 Pole | 7 |
| 2.2.6 Záznam | 7 |
| 2.2.7 Ukazatelové typy | 8 |
| 3 Srovnání síly jazyků | 8 |
| 3.1 D-programy | 9 |
| 3.2 D'-programy | 10 |
| 3.3 BJ_n programy | 10 |
| 3.4 do-repeat-cycle-exit programy | 11 |
| 3.4.1 RE_n programy | 11 |
| 3.4.2 DRE_n programy | 11 |
| 3.4.3 REC_n programy | 11 |
| 3.4.4 $DREC_n$ programy | 11 |
| 3.5 Kosarajanova hierarchie | 12 |
| 4 Podprogramy | 12 |
| 4.1 Náhrada formálního parametru skutečným | 13 |
| 4.1.1 Hodnotou | 13 |
| 4.1.2 Výsledkem | 13 |
| 4.1.3 Hodnotou-výsledkem | 13 |
| 4.1.4 Odkazem | 14 |
| 4.1.5 Jménem | 14 |
| 4.1.6 Srovnání rozdílů | 14 |

| | | |
|----------|--|-----------|
| 4.2 | Režim parametrů - ADA | 14 |
| 4.3 | Vedlejší efekty podprogramů | 15 |
| 4.4 | Volání podprogramů | 15 |
| 5 | Moduly | 16 |
| 5.1 | Abstraktní datový typ FRONTA | 16 |
| 5.1.1 | Realizace a použití několika front | 18 |
| 5.1.2 | Realizace a použití front různého typu | 19 |
| 6 | Kompilace programu | 19 |
| 6.0.3 | Sestavení, knihovny | 19 |
| 6.0.4 | Aktuálnost knihovny | 20 |
| 6.0.5 | Rekompilační seznam | 20 |
| 6.0.6 | Vyjímky | 20 |
| 7 | Paralelní programování | 21 |
| 7.1 | Paralelní procesy | 21 |
| 7.2 | Problém vzájemného vyloučení pro N procesů | 21 |
| 7.2.1 | Vzájemné vyloučení – řešení č.1 | 22 |
| 7.2.2 | Vzájemné vyloučení – řešení č.2 | 22 |
| 7.2.3 | Vzájemné vyloučení – řešení č.3 | 23 |
| 7.2.4 | Vzájemné vyloučení – řešení č.4 | 23 |
| 7.2.5 | Vzájemné vyloučení – Dereerův algoritmus | 24 |
| 7.2.6 | Vzájemné vyloučení pro N procesů | 25 |
| 7.3 | Semaforey – datový typ | 26 |
| 7.3.1 | Vzájemné vyloučení N procesů pomocí semaforu | 26 |
| 7.4 | Producent a konzument | 27 |
| 7.5 | Monitory | 27 |
| 7.6 | Readers and writers | 28 |
| 7.7 | The problem of the dining philosophers | 29 |
| 7.8 | Synchronizace procesů – Ada | 32 |
| 7.8.1 | Synchronní výměna dat | 32 |
| 7.8.2 | Selektivní čekání volaného na randevous | 32 |
| 7.8.3 | Selektivní volání randevous | 33 |
| 7.8.4 | Producent a konzument v Adě | 34 |

1 Úvod

Programovací jazyky je možno rozdělit na:

- **imperativní** — výpočet je posloupnost změn stavu paměti. Používají proměnné a přiřazovací příkaz.
- **funkcionální (aplikativní)** — definice a aplikace funkcí na argumenty. Používají podmíněný příkaz, rekurzi, skládání funkcí.



Obrázek 1: Vývoj programovacích jazyků

1.1 Syntaxe

Programovací jazyky jsou jazyky kontextové. Protože však pro analýzu kontextových jazyků nebyly dosud vynalezeny dostatečně účinné metody, popisuje se syntaxe prog. jazyků pomocí bezkontextových gramatik (CFG). Zapisujeme je v BNF — Baccusově-Naurově formě nebo pomocí syntaktických diagramů:

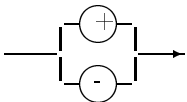
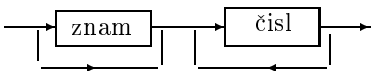
- BNF:

```

<celé-číslo> ::= [ <znaménko> ] <číslice> { <číslice> }
<znaménko> ::= +|-

```

- syntaktický diagram:



Syntaxe by měla být odolná vůči překlepům, viz slavný příklad ve FORTRANU, kdy v deklaraci cyklu byla omylem napsána tečka místo čárky, takže překladač to považoval za přiřazovací příkaz implicitně deklarované proměnné a díky tomu Američanům spadla kosmická družice za mnoho miliónů dolarů:

```
DO 10 I = 1.5 — vznikla proměnná DO10I = 1.5
A(I) =I
10 CONTINUE
```

2 Sémantika

- NEFORMÁLNÍ — definována přirozenou řečí – chyby, nejednoznačnost, neúplnost
- FORMÁLNÍ
 - **operační** — abstraktní počítač
 - **axiomatická** — pravidla ovlivňování stavu paměti konstrukty $\{P[e \rightarrow x]\} := e\{P\}$
 - **denotační** — teorie domén, Scott, Strackley (pomocí funkcí)

Správně navržená sémantika by měla splňovat tyto podmínky:

RYCHLOST PŘEKLADU by měla být vysoká

EFEKTIVITA KÓDU by měla být velká

ORTOGONALITA znamená, že má malý počet jasně odlišených konstruktů, jejichž kombinací se dělá vše ostatní

UNIFORMITA znamená, že podobné věci se zapisují syntakticky podobně

2.1 Proměnné

Proměnná má

- jméno
- atributy (typ)
- umístění (paměť, registr)
- hodnotu

Deklarace proměnných může být **implicitní** nebo **explicitní**, tj. buď proměnná vznikne při jejím prvním použití, nebo se musí předem jasně definovat způsobem [var] jméno: určení-typu [:= init-value]

- FORTRAN: INTEGER X
- PASCAL: x:INTEGER; a:array[1..10] of REAL;
- ADA,ALGOL60: a:array[M..M]of REAL; kde velikost pole se určí za runtime.

Je zajímavé si všimnout, že v přiřazovacím příkazu $x:=x+1$ znamená x na levé straně příkazu umístění proměnné, tzv. L-hodnotu, kdežto x na pravé straně hodnotu proměnné, tzv. R-hodnotu.

ALIASING proměnných je případ, kdy více proměnných různých typů sdílí totéž místo v paměti, např. Pascalovské variantní záznamy nebo Céčkové uniony:

```
FORTRAN:  REAL X
          LOGICAL Y
          EQUIVALENCE X,Y
```

```
PASCAL:  record
          case DRUH:boolean of
            TRUE: x:integer;
            FALSE: y:real;
          end;
```

```
C:  union {int x; float y;}
```

Deklarace typu

- type jméno-typu = definice-typu
- předdefinované typy (char,int,boolean)

Typová kompatibilita Kdy jsou dvě proměnné téhož typu ?

- jsou-li deklarovány pomocí téhož typu (ekvivalence jménem)
- mají-li jejich typy tutéž strukturu se shodnými komponentami (strukturní ekvivalence)

Příklad 2.1

```
type T=array[1..5] of real;
var  x:array[1..5] of real;
     y:array[1..5] of real;
     z:T;
```

Podle strukturální ekvivalence jsou x,y,z stejného typu, podle ekvivalence jménem mají různé typy. Je tu ještě jeden zádrhel, a to při násobných deklaracích a ekvivalenci jménem:

```
A,B :array[1..5] of real;
```

V Pascalu mají stejný typ, v jazyku ADA různý.

2.2 Datové typy

2.2.1 Typy

Typ je dán množinou hodnot a množinou operací. Zatím byly vynalezeny typy

- **Skalární** – digits,integer,enum,char...
- **Strukturovaný** –array,record,set,file
- **Ukazatel** – hodnoty jsou umístění objektů

2.2.2 Podtypy

Podtyp je dán bazovým typem a omezením množiny hodnot:

```
SUBTYPE T1 is T range 1..10
```

- T1 není nový typ, operace má totožné
- hodnoty objektů typu T1 jsou omezeny
- T je podtypem T (tj. sebe sama)

2.2.3 Odvozené typy

Z rodičovského typu odvodím odvozený typ:

```
TYPE OT is new RT [omezení]
```

- OT je nový typ
- $\text{val}(\text{OT}) = \text{val}(\text{RT})$
- OT zdědí mutace všech operací RT
- explicitní konverze

Odvozený typ hlídá sčítání hruštiček s jablíčkama:

```
type počhrušek is new integer;
type počjablek is new integer;
var H, H1: počhrušek;
    J: počjablek;
    I: integer;
```

Lze sčítat $H + H1 + 2$, ale nelze sčítat $H + J$.

2.2.4 Skalární typy

Skalární typy mohou být:

- **výčtové** — diskrétní
- **celočíselné** — diskrétní numerické
- **reálné** — numerické

Hodnoty skalárních typů jsou uspořádané — pro každý typ existují konstanty S'FIRST (minint) a S'LAST (maxint). Na diskrétních typech se definují operace předchůdce PRED a následníka SUCC, na výčtových navíc operace POS a VAL pro konverzi na pořadovou hodnotu a zpět.

U reálných typů je nutné si uvědomit, že reálné číslo je reprezentováno jen do určité přesnosti, takže některé matematicky správné rovnosti nemusí platit, viz slavný příklad:

```
for k:=1 to 17 do
  begin
    a[1]:=1/k;
    for j:= 2 to 40 do a[j]:=(k+1)*a[j-1]-1;
  end;
```

Teoreticky platí $a[j] = (1 + k)^{\frac{1}{k}} - 1 = \frac{1}{k}$. Teoreticky tedy jsou v poli hodnoty 3.3E-01... 3.3E-01 Prakticky tam budou hodnoty 3.3E-01... -9.16E+10 Celkem solidní nepřesnost.

Reálné typy se dělí na:

- FLOAT - s pohyblivou řádovou čárkou, relativní přesnost daná počtem cifer
- FIXED - s pevnou řádovou čárkou, absolutní přesnost v daném rozsahu

2.2.5 Pole

Pole je homogenní struktura, zadaná

- dimenzí – počtem indexů
- rozsahem indexů
- typem komponenty

Pole mohou být:

- determinovaná — interval diskretních hodnot (Pascal)
 - ★ statická (Pascal)
 - ★ dynamická
- nedeterminovaná — lib. interval diskretního typu – určen na úrovni deklarace

Příklad:

```
type VEKTOR is array ( int<> ) of int;
type MATICE is array (int<>,int<>) of int;
V1: VEKTOR (-1..8);
V2: VEKTOR (0..100);
NM: MATICE (1..N,1..M);
```

Atributy:

```
B'FIRST      1
B'LAST       100
B'RANGE      1..100
B'LENGTH     100
NM'LAST      hodnota N
NM'LAST(2)   hodnota M
```

Deskriptor pole:

| |
|--------------------|
| Typ |
| dolní mez indexu 1 |
| horní mez indexu 1 |
| dolní mez indexu 2 |
| horní mez indexu 2 |
| ⋮ |
| horní mez indexu n |
| Typ komponenty |
| Délka komponenty |

Operace zpřístupnění komponenty:

$$LocA[I] = LocA + (I - D) * dk$$

$$LocA[I, J] = LocA + (I - D_1) * dr + (J - D_2) * dk$$

$$LocA[I, J] = \alpha + I * dr + J * dk$$

$$dr = (H_2 - D_2 + 1) * dk$$

$$\alpha = LocA - D_1 * d - d_2 * dk$$

2.2.6 Záznam

Záznam je nehomogenní struktura:

```
type DAT is record DEN:1..31;
                   MES:(JAN,FEB,...);
                   ROK:1..2000;
end;
```

Selektor komponenty D:DAT;

- D.DEN
- DEN IN D
- DEN OF D
- D := (DEN=>12, MES=>OCT, ROK=>1946)

Diskriminant Speciální komponenta diskrétního typu.

- počáteční hodnota komponenty
- definice typu komponenty
- určení varianty ve variantní části
- není povolena operace := (nutno změnit celou hodnotu záznamu)

```
type BUFSIZE is 0..MAX
type BUFFER(SIZE:BUFSIZE := 100) is record
    UK: BUFSIZE :=0;
    HODN:array(1..SIZE)of char;
end;
VELKY: BUFFER(200); ZPRÁVA:BUFFER;
```

Variantní záznam

```
type POHLAVI is (M,Ž);
type OSOBA(P :POHLAVI := M) is record
    jméno: string(1..20);
    case P of
        M: výška:integer;
           vous:boolean;
        Ž: míry:array(1..3)of integer;
    endcase;
endrec;
A,B(Ž):OSOBA;
```

2.2.7 Ukazatelové typy

type UKAZ is ↑M; množina hodnot je skrytá, kromě nil. Alokace objektu: new(x)

Vrácení na haldu: dispose(x)

Při přiřazování ukazatelů mohou vzniknout nedostupné objekty, tak bacha !

3 Srovnání síly jazyků

Budeme srovnávat sílu jednotlivých tříd řídicích struktur zkoumáním vztahů (relací) mezi třídami programů generovaných použitím pouze řídicích struktur příslušných tříd. Relace definujeme na základě různých typů transformací mezi programy.

Typy transformací programu P_1 na P_2 (všechny zachovávají funkční ekvivalentnost P_1 a P_2):

- **velmi přísná** (vp) — je zakázáno
 - ★ zavádět nové primitivní akce, proměnné, predikáty
 - ★ měnit výpočetní historie pro jednotlivé vstupy

★ duplikovat původní akce a predikáty

- **sémantická** (sem) — je zakázáno

★ zavádět nové primitivní akce, proměnné, predikáty

je povoleno

★ měnit výpočetní historie

★ duplikovat původní akce a predikáty

- **funkční** (f) — je povoleno vše

Příklad:

konstrukci

```
CYKL: if b goto VEN;
```

```
  a;
```

```
  goto CYKL;
```

```
VEN:
```

můžeme velmi přísně transformovat na `while not(b) do a;`

Nebo `if b then a else c` funkčně ztransformujeme na:

```
p:=true;
```

```
while (b and p) do begin a; p:=false; end;
```

```
while p do begin c; p:=false;end;
```

Relace mezi třídami programů

- $T_1 \leq_t T_2$ jestliže každý program třídy T_1 může být transformován na program třídy T_2 .

- $T_1 <_t T_2$ jestliže $T_1 \leq_t T_2$, ale ne $T_2 \leq_t T_1$

- $T_1 \equiv T_2$ jestliže $T_1 \leq_t T_2$ a $T_2 \leq_t T_1$

$T_1 \equiv T_2$ je silnější výsledek pro transformaci přísnějšího typu (např. $T_1 \equiv_{sem} T_2 \Rightarrow T_1 \equiv_f T_2$).

$T_1 < T_2$ je silnější pro transformaci méně přísného typu ($T_1 <_{sem} T_2 \Rightarrow (T_1 <_{vp} T_2)$ nebo (T_1 není v žádné relaci vp s T_2).

"jakákoliv třída programů" $\leq L$ pro všechny typy transformací!

L je třída programů s lib. návěštím a `goto`.

a- primitivní akce (přířazovací příkaz, I/O příkaz, volání procedury, ...) je programem každé třídy. Sekvence programů každé třídy je opět programem této třídy.

3.1 D-programy

Jsou-li D_1, D_2 D-programy, pak i

- větvení `if b then D_1 else D_2`

- cyklus `while b do D_1`

jsou D-programy.

Böhm, Jacopini: $D \equiv_f L$

Transformace zavádí booleovské proměnné, jejichž hodnoty se ukládají do zásobníku, test vrcholu zásobníku se využívá k rozhodování o předání řízení (viz též Hořejš, SOFSEM 1974).

jsou BJ_n -programy.

$$D \equiv_{vp} BJ_1 \Rightarrow D' \equiv_{sem} D \equiv_{sem} BJ_1 <_{sem} L$$

Cyklus c_1 z BJ_1 -struktur je cyklus while.

Pro všechna n je $BJ_n <_{sem} BJ_{n+1}$.

3.4 do-repeat-cycle-exit programy

3.4.1 RE_n programy

Tyto programy obsahují

- větvení `if b then P_1 else P_2`
- cyklus `repeat P_1 end`
- příkaz `exit(i)`, kde $0 < i \leq n$

3.4.2 DRE_n programy

Tyto programy obsahují totéž co RE_n programy plus

- cyklus `while b do P_1`

3.4.3 REC_n programy

Tyto programy obsahují totéž co RE_n programy plus

- příkaz `cycle(i)`, kde $0 < i \leq n$

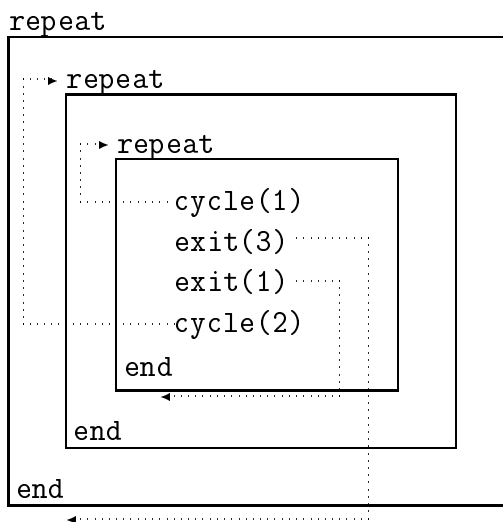
3.4.4 $DREC_n$ programy

Tyto programy obsahují obsahují všech pět konstrukcí z DRE_n a REC_n .

`repeat P end` je cyklus, ve kterém se P provádí tak dlouho, dokud se nenarazí na `exit(i)` resp. `cycle(i)`.

`exit(i)` je ukončení všech `repeat ... end` cyklů nadřazených příkazu `exit(i)` až do úrovně i včetně (hloubka uzávkování `exit(i)` "závorkami" `repeat ... end`).

`cycle(i)` – znovuzahájení provádění cyklu nadřazeného `cycle(i)` na i -té úrovni



Zřejmě $RE_n \leq_{sem} DRE_n, REC_n, DREC_n$.

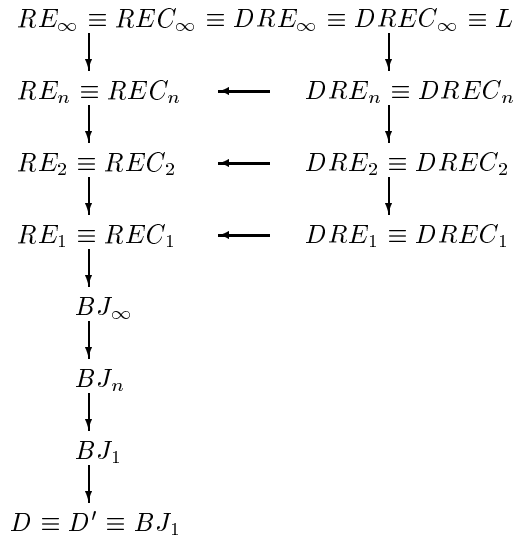
Platí $BJ_n <_{sem} RE_1$, a protože $BJ_n <_{sem} BJ_{n+1}$, každý BJ_{n+1} program lze vyjádřit jako RE_1 program.

Ledgard : $RE_{equiv_{sem}} REC_n <_{sem} DRE_n \equiv DREC_n$.

Analogicky $DRE_n <_{sem} DRE_{n+1}$. Kosarajan s Petersonem dokázali, že $RE_\infty \equiv_{sem} L$. Libovolný L-program s n -predikáty, $n \geq 2$ (včetně vícenásobných výskytů téhož predikátu) lze vyjádřit jako RE_{n-1} program takový, že

- maximální úroveň vnoření cyklů repeat-end je menší nebo rovna $n - 1$
- maximální úroveň vnořování if-then-else je menší nebo rovna n

3.5 Kosarajanova hierarchie



V tomto obrázku znamená \downarrow relaci $<_{sem}$ a \leftarrow relaci \leq_{sem} .

4 Podprogramy

Motivace: Podprogramy jsou procedury a funkce. Zmenšují velikost kódu, protože odstraňují jeho opakování. Podporují rozklad problému na podproblémy, což se uplatní při metodice návrhu algoritmů.

Podprogram se definuje deklarací, použije se voláním. Volání procedury je příkaz, kdežto volání funkce je výraz. Definice má dvě části: hlavičku, což je dohoda s uživatelem, a tělo, které je realizací dohody, pro uživatele není podstatné. Důvody pro fyzické oddělení hlavičky a těla:

- čitelnost
- rekurze (při vzájemném volání, při volání sebe sama)
- podprogram modulu (modulární výstavba programů)

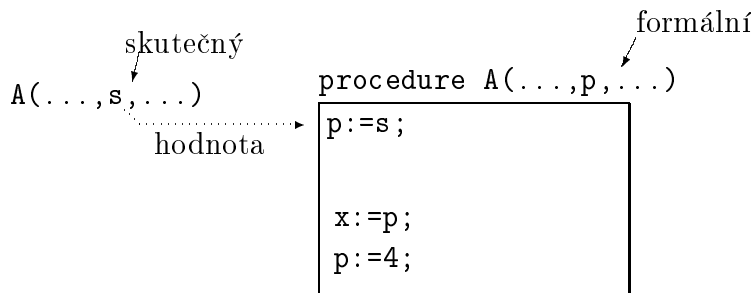
Hlavička udává

- jméno podprogramu (nabízené služby)
- seznam formálních parametrů
 - ★ specifikace jejich typu
 - ★ způsob náhrady formálního skutečným
 - hodnotou (Pascal)
 - výsledkem (Ada)
 - hodnotou-výsledkem (Ada)

- odkazem (adresou-Pascal, jménem-Algol60)
- ★ režim (místo způsobu náhrady)
 - in – vstupní parametr
 - out – výstupní parametr
 - inout – v/v parametr
 - u funkce typ vrácené hodnoty

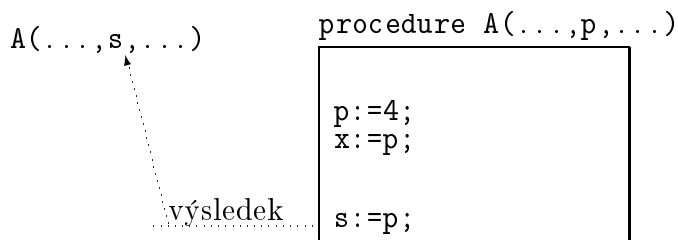
4.1 Náhrada formálního parametru skutečným

4.1.1 Hodnotou



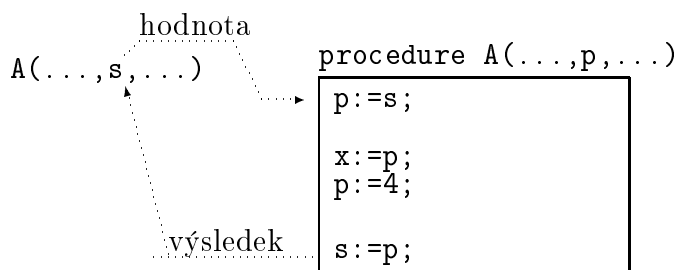
Při volání procedury je hodnota skutečného parametru zkopírována do lokální proměnné p . Změny hodnoty p se na skutečném parametru neprojeví, hodnota skutečného parametru není změněna.

4.1.2 Výsledkem



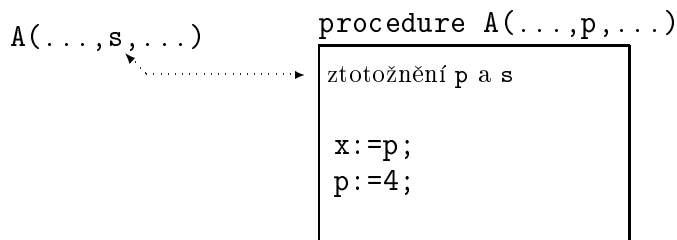
Hodnota skutečného parametru v době volání není použita. Úskalí tohoto je, že záleží na pořadí, v kterém se zpracovávají parametry, např. $A(s, s)$ $A(B(i), i)$.

4.1.3 Hodnotou-výsledkem



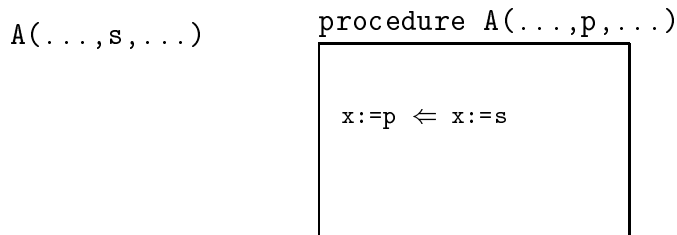
V průběhu výpočtu procedury není globální s měněno, až na konci se zkopíruje hodnota p do s .

4.1.4 Odkazem



Při volání odkazem je formální parametr ztotožněn se skutečným, takže změny formálního parametru se ihned projeví na obsahu skutečného parametru.

4.1.5 Jménem



Zavolám-li A skutečným parametrem, tak jsou všechny výskyty p nahrazeny textově s.

4.1.6 Srovnání rozdílů

```
I:integer; A:array[1..10]of integer;
procedure Výměna(x,y);
  pom:integer;
  begin pom:=x; x:=y; y:=pom; end;
begin I:=3; A[I]:=6;
  write(I,A[I]);
  Výměna(I,A[I]);
  write(I,A[I]);
end;
```

Výstup:

| | | |
|-------------|---|-----|
| | 3 | 6 |
| hodnotou | 3 | 6 |
| výsledkem | ? | ? |
| hodn.-výsl. | 6 | 3/6 |
| odkazem | 6 | 3 |
| jménem | 6 | 3 |

4.2 Režim parametrů - ADA

– neuvádí způsob náhrady formálního skutečným explicitně, ale vyžaduje udání režimu parametru.

in vstupní — hodnota dána skutečným parametrem, povolena pouze reference, nelze měnit jeho hodnotu

out výstupní parametr — jeho hodnota dává hodnotu skutečného parametru, povolena pouze definice, nelze ho uvádět na pravé straně přiřazovacího příkazu

inout povolena definice i reference

Parametr skalárního datového typu lze předávat

- náhrada hodnotou (in)
- výsledkem (out)
- hodnotou–výsledkem (inout)

Parametr strukturovaného typu lze předávat kopírováním nebo referencí, jsou-li výsledky různé, jde o chybný program.

4.3 Vedlejší efekty podprogramů

jsou změny hodnot globálních proměnných v těle podprogramu a to buď přímo změnou hodnoty globální proměnné, nebo pomocí parametrů. U procedur jsou vedlejší efekty nutné, jinak by byly jen prázdnými příkazy. U funkcí je vedlejší efekt nežádoucí, u některých jazyků je zákaz kontrolovaný kompilátorem.

```
function F(var x:real):real;
begin x:=abs(x);
      F:=sqrt(x);
end;
...
x:=-9;
write(x+F(x));
```

Volání odkazem mění hodnotu skutečného parametru během výpočtu, takže není jasné, co program vytiskne.

4.4 Volání podprogramů

Zajištění návratu (z hlediska implementace) a předání řízení. Před předáním řízení úschova registrů, po návratu obnova. Podprogramy jsou

- otevřené – na místo volání se zkopíruje tělo
- uzavřené

Jednoduchý návratový mechanismus spočívá v tom, že při volání podprogramu se do něj zapíše adresa, kam se má vrátit řízení. Tento model selhává při rekurzivním volání procedur.

Rekurzivní podprogramy mohou opakovaně volat sebe sama, nebo se mohou volat navzájem, např. podprogram A zavolá podprogram B, ten zavolá C a ten opět zavolá A. Klasickým příkladem rekurzivního volání je program pro výpočet Fibonacciho čísel:

```
function Fib(n):int;
begin if n=1 or n=2 then Fib:=1
      else Fib:=Fib(n-1)+Fib(n-2)
end;
```

Zásobník návratových adres je nutný, stejně jako aby každé zavolání procedury vytvořilo další kopii lokálních proměnných.

Aktivační záznam bloku (procedury) obsahuje:

- parametry (skutečné)
- návratovou adresu
- ukazatel na aktivační záznam volajícího
- lokální proměnné
- ukazatel na aktivační záznam s globálními proměnnými
- pomocné proměnné pro vyhodnocení výrazů

Bloková struktura Blok \rightarrow [`<deklarace><příkazy>`]
 =posloupnost deklarací následovaná posloupností příkazů

- BASIC, COBOL - program je jedinný blok, všechny proměnné jsou globální
- FORTRAN - hl. program je blok, každý podprogram je blok, jsou zapisovány za sebou
- PASCAL, ADA, ALGOL - program je jedinný blok, ale má vnořené bloky — vnořování bloků (nesting).

S tím souvisí rozsah platnosti deklarací. Scope deklarace — platnost pouze pro vnitřek bloku, s výjimkou vnořených bloků s redeklarací téhož jména. Objekt je lokální ve svém bloku, globální ve vnořených blocích, a neexistuje v nadřazených blocích.

Přetěžování jmen podprogramů (overloading)

- mají-li různé podprogramy stejné jméno \Rightarrow overloading
- nezakrývají se, pokud mají různý profil parametrů (stejný počet, různý typ)
- mají-li parametry stejný profil, zakrývají se stejným způsobem jako objekty

Syntactic sugar

- je možno pojmenovat skutečné parametry
`proc P(PAR1,PAR2,PAR3);`
 volání: `P(PAR2=>1,PAR3=>A,PAR1=>1);`
 uvedeme-li jména parametrů, nemusíme zachovávat pořadí
- je možnost explicitního zadávání parametrů
`proc P(P1,in P2:=0);`
 volání: `P(P1=>1);`
- možnost definovat operátory
`function "+"(levý,pravý:vektor):vektor;`
`A,B,C:vektor;`
`A:=B+C místo +(B,C)`

5 Moduly

Moduly jsou programové jednotky, nabízející celou řadu služeb programových (procedury a funkce) a datových (typy, objekty). Každý modul má dvě části

| | |
|----------|-------------------------------------|
| Hlavička | viditelná část – specifikace služeb |
| Tělo | skrytá část – realizace služeb |

Modul je abstraktní deklarací. Tělo může obsahovat vnitřní procedury, funkce, proměnné. Příklad modulu si uveďme na implementaci fronty.

5.1 Abstraktní datový typ FRONTA

Signatura:

- `new` vytváří frontu
- `add` přidává prvek
- `remove` odstraňuje prvek
- `empty` vrací booleovskou hodnotu, zda je fronta prázdná
- `front` vrací prvek v čele fronty

AXIOMY:

- `empty(add(x,f))=false`
- `front(add(x,new))=x`
- `front(add(y,add(f,x)))=front(add(x,f))`
- `remove(add(x.new))=new`
- `remove(add(y,add(f,x)))=add(y,remove(add(x,f)))`
- `front(new)`, `remove(new)` nedefinováno

Operační sémantika:

- $new = \langle \rangle$
- $add(x \langle x_1 x_2 \dots x_n \rangle) = \langle x_1 x_2 \dots x_n x \rangle$
- $front(\langle x_1 x_2 \dots x_n \rangle) = x_1$
- $remove(\langle x_1 x_2 \dots x_n \rangle) = \langle x_2 \dots x_n \rangle$
- $empty(\langle x_1 x_2 \dots x_n \rangle) = false$
- $empty(\langle \rangle) = true$

Fronta je implementována jako lineárně zřetězený seznam, doplněný o dva ukazatele — `čelo` ukazuje na prvek v čele fronty a `volný` ukazuje na volné místo za koncem fronty. Modul neumožňuje nesmysly typu `čelo:=18`.

```

pack FRONTA is
  proc Ini;
  proc Vlož(x:int);
  proc Odeber(x:int);
  fun empty:bool;
end FRONTA;

pack body FRONTA is
  type Elem is rec Další:=^Elem;
                    Hodn:int;
                    end rec;
  čelo:^Elem;volný:^Elem;

  proc Ini;
begin
  čelo:=new(Elem);
  volný:=čelo;
end

  proc Vlož(x:int);
begin
  Volný^.hodn:=x;
  Volný^.Další:=new(Elem);
  Volný:=Volný^.Další
end

  proc Odeber(x:int);
  ...

```

```

fun empty:bool;
begin Empty:=čelo=volný end;

end.

```

Využití služeb modulu fronta

```

program B;
with FRONTA;
begin
...
Fronta.Ini;
Fronta.Vlož(25);
...
end B;

```

5.1.1 Realizace a použití několika front

I. Použití generických programových jednotek

```

generic pack VZOR-FRONTY is
  proc Ini;
  proc Vlož(x:int);
  proc Odeber(x:int);
  fun Empty:bool;
end;

```

```

pack body VZOR-FRONTY is
  to samé co u modulu FRONTA
end;

```

```

program A; with VZOR-FRONTY
  pack Vstupní-fronta is new VZOR-FRONTY
  pack Výstupní-fronta is new VZOR-FRONTY
begin
  Vstupní-fronta.Ini; Výstupní-fronta.Ini;
  ...
end A;

```

II. Export datového typu

```

pack ADT-FRONTA is
  type FRONTA is private;
  proc Ini(F:FRONTA);
  proc Vlož(F:FRONTA,x:int);
  ...
end

```

```

pack body ADT-FRONTA is
  type Elem is ...
  type FRONTA is rec čelo:^Elem;
                    volný:^Elem;
                    end rec;
  proc Ini(F:FRONTA);
begin F.čelo:=new(Elem);
      F.volný:=F.čelo;

```

```
end;
atd.
```

5.1.2 Realizace a použití front různého typu

Zavedeme parametr generického modulu

```
generic
  type T is private;
pack PAR-VZOR-FRONTA is
proc Ini
  ...
end.

program A;
pack FRONTA-INT is new PAR-VZOR-FRONTA(T=>int);
pack FRONTA-CHAR is new PAR-VZOR-FRONTA(T=>char);
```

6 Kompilace programu

programová jednotka – procedura, funkce, modul, proces, ...

kompilační jednotka – program, progr. jednotka, specifikace (hlavička) prog. jednotky, tělo prog. jednotky

Překlad

- všechny programové jednotky společně (Pascal)
- všechny programové jednotky odděleně a nezávisle (FORTRAN)
výhoda: při ladění jedné procedury nemusím znova překládat celý program
nevýhoda: nezachytí nekorektnosti při vzájemném volání procedur (různý počet parametrů, neshodnost typů)
- programové jednotky možno kompilovat separátně v takovém pořadí, aby bylo možno kontrolovat korektnost použití služeb jiných programových jednotek (Ada)

Knihovna kompilačních jednotek

| |
|---|
| <pre>jméno – jednoznačná definice (proc, fun, pack, task) – druh integer – verze, při každém překladu, který mění služby +1 {(před1, ver1), ..., (předn, vern)} – seznam kompilačních předchůdců {(násl1, ..., násln)} – následníci proc P(in X:T1; out y:T2); fun F(in X:T3):T4; type T ... – služby</pre> |
|---|

6.0.3 Sestavení knihovny

prověření úplnosti knihovny vzhledem k hlavní programové jednotce Main

```
proc KOMP-STR(lib, M, CH, KS) kde
```

- lib – knihovna, vstupní parametr
- M – hlavní pr. jednotka (Main)
- CH – chybějící pr.j. (nevrátí všechny), výstupní parametr
- KS – kompilační struktura pro M

```

KS:=∅; CH:={M};
while CH≠∅ and CH∩lib≠∅ do
necht n∈CH∩lib;
CH:=CH- $\{n\}$ ∪ $\{m \mid (n,v) \in n.před\} \cup n.násl$ ;
CH:=CH-KS; KS:=KS∪ $\{n\}$ 
end do;
if CH=∅ then "úplná, v KS kompilační struktura"
else "neúplná, v KS chybějící"

```

6.0.4 Aktuálnost knihovny

```

fun AKTUÁLNÍ(KS):Bool
úplná kompilační struktura pro pr.j. M
AKT:=true;
POM:=KS;
while POM≠∅ and AKT do
necht n∈POM
AKT:={ $p \mid (p,v) \in n.před$  and  $v \neq p.verze$ }=∅;
POM:=POM- $\{n\}$ ;
end do;
aktuální:=AKT;

```

6.0.5 Rekompilační seznam

```

proc REKOMP(KS,OK,RS)

```

- KS – kompilační struktura
- OK – pr.j. z KS, které jsou OK
- RS – rekompilační seznam

```

OK:=∅; PS:=<>; POM:=KS;
while POM≠∅ do
necht n∈POM tak, že  $\{p \mid (p,v) \in n.před\} \subseteq OK \cup RS$ ;
if  $\{p \mid (p,v) \in n.před\} \subseteq OK$  and  $\forall (p,v) \in n.před$ 
then OK:=OK∪ $\{n\}$ 
else RS:=RS+ $\langle n \rangle$ ;
POM:=POM- $\{n\}$ ;
end do;

```

6.0.6 Vyjímky

standardní – numeric, domain, range; zpracování: hlášení a abort

uživatelské • deklarace: error, exception

- vyvolání: raise error
- zpracování

```

begin
...
exception
when error => ...;
when numeric.error => ...;
when other => ...;
end

```

Rozsah platnosti jména vyjímky určen staticky. Předání řízení po vyvolání vyjímky (nalezení handleru).

7 Paralelní programování

7.1 Paralelní procesy

- společná paměť, zasilání zpráv
- asynchronní běh
 - ★ žádné předpoklady o vzájemné rychlosti procesů
 - ★ libovolné prolínání instrukcí procesů
 - ★ prolínají se *atomické instrukce*

```
N:integer := 0;
task body P1 is
  begin N:=N+1; end P1;
task body P2 is
  begin N:=N+1; end P2;
```

Může dojít k libovolnému prolínání atomických instrukcí:

... příkaz $N:=N+1$ je atomický

| proces | instrukce | N |
|--------|-----------|---|
| | | 0 |
| P1 | INC N | 1 |
| P2 | INC N | 2 |

... příkaz $N:=N+1$ není atomický

| proces | instrukce | N | R1 | R2 |
|--------|------------|---|----|----|
| P1 | Load R1,N | 0 | 0 | 0 |
| P2 | Load R2,N | 0 | 0 | 0 |
| P1 | INC N | 0 | 1 | 0 |
| P2 | INC N | 0 | 1 | 1 |
| P1 | Store R1,N | 1 | 1 | 1 |
| P2 | Store R2,N | 1 | 1 | 1 |

7.2 Problém vzájemného vyloučení pro N procesů

N procesů provádí posloupnosti příkazů, které mohou být rozděleny na části, tzv. *kritické sekce* a sekce, které nejsou kritické. Program musí mít *vlastnost vzájemného vyloučení* — provádění příkazů kritických sekcí dvou či více procesů se nesmí prolínat, tj. žádné dva procesy nesmí být současně ve svých kritických sekcích.

Požadavky a předpoklady

1. Řešení musí spočívat v přidání jistých příkazů — tzv. *vstupního protokolu* před příkazy kritické sekce a jiných příkazů — tzv. *výstupního protokolu* za příkazy kritické sekce. Obecně:

```
loop
  Nekritická sekce;
  Vstupní protokol;
  Kritická sekce;
  Výstupní protokol;
endloop;
```

2. Předpokládá se, že procesy nemohou skončit během provádění kritických sekcí a protokolů. Mohou skončit v sekcích, které nejsou kritické, což nesmí ovlivnit provádění ostatních procesů.
3. Nesmí dojít k *uváznutí (deadlocku)*, jestliže několik procesů provádí vstupní protokol, jeden z nich musí uspět.
4. Nesmí dojít ke *strádání (starvation)* jednotlivého procesu — provádí-li průběžně vstupní protokol (v konkurenci s ostatními procesy), musí v konečné době uspět. Provádí-li proces vstupní protokol bez konkurence jiných procesů, musí uspět (a to, pokud možno, rychle).

7.2.1 Vzájemné vyloučení – řešení č.1

čp: integer range 1..2 := 1 ... který proces může do KS

```
task body P1 is
begin loop
  Nekritická_sekce_1;
  loop
    exit when čp=1;
  end loop;
  Kritická_sekce_1;
  čp:=2;
end loop;
end P1;
```

```
task body P2 is
begin loop
  Nekritická_sekce_2;
  loop
    exit when čp=2;
  end loop;
  Kritická_sekce_2;
  čp:=1;
end loop;
end P1;
```

- + má vlastnost vzájemného vyloučení (kdyby P1 i P2 byly současně v KS, při vstupu jednoho bylo čp=1, při vstupu druhého čp=2, přičemž v KS není možno čp měnit – spor)
- + nenastane deadlock (oba procesy mohou nekonečně dlouho současně provádět svoje vstupní protokoly, muselo by současně platit čp=1, čp=2)
- + nikdo nestrádá (P1 by musel neustále vstupovat do KS, přičemž P2 by současně neustále prováděl svůj výstupní protokol, ale jakmile P1 opustí KS, změní čp tak, že do KS vstoupí P2)
- může selhat bez konkurence partnera (jestliže P2 skončí v nekritické sekci, hodnota čp nemůže být již nikdy změněna z 2 na 1. P1 tedy nejpozději při druhém pokusu o vstup do KS uvázne ve svém vstupním protokolu)

7.2.2 Vzájemné vyloučení – řešení č.2

C1,C2: integer range 0..1:=1

```
task body P1 is
begin loop
  Nekritická_sekce_1;
  loop exit when C2=1;end loop
  C1:=0;
  Kritická_sekce_1;
  C1:=1;
end loop;
end P1;
```

```
task body P2 is
begin loop
  Nekritická_sekce_2;
  loop exit when C1=1;end loop
  C2:=0;
  Kritická_sekce_1;
```

```

    C2:=1;
    end loop;
end P2;

```

– nemá vlastnost vzájemného vyloučení. K porušení vede tato posloupnost:

1. P1 testuje C2 a C2=1
2. P2 testuje C1 a C1=1
3. P1 nastaví C1 na 0
4. P2 nastaví C2 na 0
5. P1 vstoupí do KS
6. P2 vstoupí do KS

7.2.3 Vzájemné vyloučení – řešení č.3

```

C1,C2: integer range 0..1:=1
task body P1 is
begin loop
    Nekritická_sekce_1;
    C1:=0;
    loop exit when C2=1;end loop
    Kritická_sekce_1;
    C1:=1;
end loop;
end P1;

```

```

task body P2 is
begin loop
    Nekritická_sekce_2;
    C2:=0;
    loop exit when C1=1;end loop
    Kritická_sekce_2;
    C2:=1;
end loop;
end P2;

```

```
;
```

+ má vlastnost vzájemného vyloučení

– může nastat deadlock

1. P1 nastaví C1 na 0
2. P2 nastaví C2 na 0
3. P1 testuje C2 a zůstává viset
4. P2 testuje C1 a zůstává viset

7.2.4 Vzájemné vyloučení – řešení č.4

```

C1,C2: integer range 0..1:=1
task body P1 is
begin loop
    Nekritická_sekce_1;
    C1:=0;

```

```

    loop
      exit when C2=1;
      C1:=1;
      C1:=0;
    end loop
    Kritická_sekce_1;
    C1:=1;
    end loop;
end P1;

```

```

task body P2 is
begin loop
  Nekritická_sekce_2;
  C2:=0;
  loop
    exit when C1=1;
    C2:=1;
    C2:=0;
  end loop
  Kritická_sekce_2;
  C2:=1;
  end loop;
end P2;

```

+ má vlastnost vzájemného vyloučení

– proces může strádat

– může nastat deadlock

7.2.5 Vzájemné vyloučení – Dereerův algoritmus

C1,C2: integer range 0..1:=1;

čp: integer range 1..2:=1;

```

task body P1 is
begin
  loop
    Nekritická_sekce_1;
    C1:=0;
    loop
      exit when C2=1;
      if čp=2 then
        C1:=1;
        loop exit when čp=1; end loop
        C1:=0;
      endif;
    end loop
    Kritická_sekce_1;
    C1:=1;
    čp:=2;
  end loop;
end P1;

```

```

task body P2 is
begin

```



```

loop
  Nekritická_sekce_2;
  C2:=0;
  loop
    exit when C1=1;
    if čp=1 then
      C2:=1;
      loop exit when čp=2; end loop
      C2:=0;
    endif;
  end loop
  Kritická_sekce_2;
  C2:=1;
  čp:=1;
end loop;
end P1;

```

- + má vlastnost vzájemného vyloučení
- + nenastane deadlock
- + nikdo nestrádá
- + nesehává ani bez konkurence partnera

7.2.6 Vzájemné vyloučení pro N procesů

Algoritmus "bakery"

```

Vybrá: array(1..N) of integer :=(others=>0);
Číslo: array(1..N) of integer :=(others=>0);
task body Pi is
I:constant integer := (číslo tohoto procesu);
begin
loop
  Nekritická_sekce_i;
  Vybrá(I):=1;
  Číslo(I):=1+max(Číslo);
  Vybrá(I):=0;
  for J in 1..N
    loop
      if J<>I then
        loop exit when Vybrá(J)=0; endloop;
        loop exit when ((Číslo(J)=0) or (Číslo(I)<Číslo(J))
          or (Číslo(I)=Číslo(J) and I<J)); endloop;
      endif;
    endloop;
  Kritická_sekce_i;
  Číslo(I):=0;
endloop;
end Pi;

```

- + má vlastnost vzájemného vyloučení
- + nenastane deadlock
- + nikdo nestrádá
- + nesehává ani bez konkurence partnerů

7.3 Semafory – datový typ

množina hodnot: nezáporná celá čísla (obecný semafor) nebo $\{0,1\}$ (binární semafor)

Operace (semafor s neurčitým čekáním)

- `Wait(s)`
if $S > 0$ then $S := S - 1$ else pozastav provádění procesu na semaforu S
- `Signal(S)`
if na semaforu S jsou pozastaveny nějaké procesy
then vyber jeden z nich a nech ho pokračovat
else $S := S + 1$

Vlastnosti semaforu

- `Wait(S)` a `Signal(S)` jsou atomické operace (zajištěno HW nebo OS). Konkrétně u `Wait` žádný příkaz nemůže prolínat mezi testem $S > 0$ a snížením hodnoty S nebo pozastavením procesu.
- Semafor musí být inicializován nezápornou počáteční hodnotou
- Pro semafor S platí tyto invarianty: $S \geq 0$ and $S = S_0 + \#Signal - \#Wait$

Varianty semaforu

– semafor s neurčitým čekáním

– semafor s čekáním ve frontě FIFO

- `Wait(s)`
if $S > 0$ then $S := S - 1$ else pozastav provádění procesu na semaforu S a zařaď ho do fronty na semaforu S ;
- `Signal(S)`
if na semaforu S jsou pozastaveny nějaké procesy
then vyber první z fronty a nech ho pokračovat
else $S := S + 1$

– semafor s aktivním čekáním

- `Wait(s)`
loop if $S > 0$ then $S := S - 1$; exit; endif; endloop;
- `Signal(S)`
 $S := S + 1$;

7.3.1 Vzájemné vyloučení N procesů pomocí semaforu

```
S:=semaphore:=1;
task body Pi is
begin loop Nekritická_sekce_i;
    Wait(S);
    Kritická_sekce_i;
    Signal(S);
endloop;
end Pi;
```

+ má vlastnost vzájemného vyloučení

+ nenastane deadlock

- + nikdo nestrádá pro semafore s čekáním ve frontě
- pro $N > 2$ může dojít ke strádání pro semafore s neurčitým čekáním
- může dojít ke strádání pro semafore s aktivním čekáním

7.4 Producent a konzument

Problém: Jsou dva typy procesů. *Producenti* produkují datové elementy, které zasílají konzumentům. *Konzumenti* zpracovávají datové elementy, které dostávají od producentů. Producent zasílá elementy do fronty (FIFO bufferu), odkud si je konzument vyzvedává, tj. všechny vyprodukované jsou zkonsumovány ve stejném pořadí. Producent nesmí zasílat do plné fronty, konzument nesmí číst z prázdné.

Řešení pomocí kruhového bufferu a semaforů

```
Buff: array(0..N-1)of element;
In,Out:integer:=0;
Počet:semaphore:=0; počet prvků v bufferu
Volné:semaphore:=N; počet volných míst
task body Producent is
I:element;
begin
loop Produkuj(I);
Wait(Volné);
Buff(In):=I;
In:=(In+1)mod N;
Signal(Poččet);
endloop;
end Producent;
```

```
task body Konzument is
I:element;
begin
loop Wai(Poččet);
I:=Buf(Out);
Out:=(Out+1)mod N;
Signal(Volné);
Konzumuj(I);
endloop;
end Konzument;
```

7.5 Monitory

Řešení problému producent–konzument

```
monitor PKmon is
Počet:integer:=0;
In,Out:integer:=0;
Buf:array(0..N-1) of element;
Není_plný,Neprázdný:condition;    <- podmínkové proměnné

procedure Ulož(in I:integer) is
begin
if Počet=N then Wait(Není_plný); endif;
Buf(In)=I;In:=(In+1)mod N; Počet:=Počet+1;
Signal(Neprázdný);
```

```

end;

procedure Odeber(out I:integer) is
begin
  if Počet=0 then Wait(Neprázdný); endif;
  I:=Buf(Out); Out:=(Out+1)mod N; Počet:=Počet-1;
  Signal(Není_plný);
end;

end monitor;

task body Producent is
I:element;
begin loop Produkuj(I);Ulož(I);endloop; end;

task body Konzument is
I:element;
begin loop Odeber(I);Konzumuj(I); endloop; end;

```

Monitor

- programová jednotka: datové objekty, procedury, podmínkové proměnné
- vlastnost vzájemného vyloučení, pouze jeden proces může provádět proceduru monitoru
- podmínkové proměnné – operace:

Wait(c) proces je pozastaven ve FIFO frontě na proměnnou C. Nevztahuje se na něj vlastnost ME.

Signal(c) je-li fronta na C neprázdná, je oživen proces z čela fronty

Nonempty(C) funkce vrací true, je-li fronta na C neprázdná

Emulace semaforu pomocí monitoru

```

monitor Emulace_semaforu is
S:integer:= ...;
Nenulový:condition;

procedure Wait_semaforu is
begin if S=0 then Wait(Nenulový); endif;
      S:=S-1;
end;

procedure Signal_semaforu is
begin S:=S+1;Signal(Nenulový);
end;

```

Poznámka — rovněž emulace monitoru pomocí semaforů je možná, ale je komplikovanější, monitor je prostředek vyšší úrovně.

7.6 Readers and writers

Čtenáři a písaři - abstrakce přístupu do DB. Problém: jsou dva typy procesů — **Readers** čtou data ze sdílené paměti, vůči sobě navzájem nemají požadavek vzájemné výlučnosti. **Writers** zapisují data do sdílené paměti, požadují vzájemnou výlučnost jak vůči sobě navzájem, tak i vůči čtenářům.

```

task body Reader is
begin loop Zahaj_čtení;
    Čti_data;
    Konec_čtení;
endloop;
end Reader;

```

```

task body Writer is
begin loop Zahaj_zápis;
    Piš_data;
    Konec_zápisu;
endloop;
end Writer;

```

Monitor pro problém Readers and writers

```

monitor RWM
    čtenáři:integer:=0;
    píše:boolean:=false;
    OK_čtení,OK_zápis:condition;

procedure Zahaj_čtení is
begin if píše or Nonempty(OK_zápis)
    then Wait(OK_čtení); endif;
    čtenáři:=čtenáři+1;
    Signal(OK_čtení);
end;

procedure Konec_čtení is
begin čtenáři:=čtenáři-1;
    if čtenáři=0 then Signal(OK_zápis); endif;
end;

procedure Zahaj_zápis is
begin if čtenáři<>0 or píše then Wait(OK_zápis); endif; end;

procedure Konec_zápis is
begin píše:=false;
    if nonempty(OK_čtení) then Signal(OK_čtení)
        else Signal(OK_zápis);
    endif;
end;

end RWM;

```

7.7 The problem of the dining philosophers

Problém: ve společnosti je 5 filozofů. Tito jen jedí a přemýšlejí:

```

task body Filozof is
begin loop přemýšlení;
    vstupní_protokol;
    jídlo;
    výstupní_protokol;
endloop;
end Filozof;

```

U stolu je do kruhu 5 talířů, mezi nimi je 5 vidliček. Máme navrhnout protokoly, aby byly splněny požadavky

- Filozof může jíst jen má-li 2 vidličky
- Jednu vidličku nemohou uchopit 2 filozofové současně
- nenastane deadlock
- nikdo nestrádá (nehladoví)
- chování filozofa, kterému nekonkuruje soused, je efektivní

řešení pomocí semaforů

```
Vidlička:array(0..4) of semaphore :=(others=>1);
```

```
task body Filozof is (I je identifikace filozofa)
begin loop
  Přemýšlení;
  Wait(Vidlička(I));
  Wait(Vidlička((I+1) mod 5));
  Jídlo;
  Signal(Vidlička(I));
  Signal(Vidlička((I+1) mod 5));
endloop;
end Filozof;
```

- jednu vidličku nemohou uchopit dva filozofové současně
- může nastat deadlock (všichni současně uchopí vidličky po levici)

Deadlock předchozího řešení odstraníme například tím, že současně povolíme jíst jen čtyřem filozofům.

```
Volno:semaphore :=4;
```

```
Vidlička:array(0..4) of semaphore :=(others=>1);
```

```
task body Filozof is (I je identifikace filozofa)
begin loop
  Přemýšlení;
  Wait(Volno);
  Wait(Vidlička(I));
  Wait(Vidlička((I+1) mod 5));
  Jídlo;
  Signal(Vidlička(I));
  Signal(Vidlička((I+1) mod 5));
  Signal(Volno);
endloop;
end Filozof;
```

- jednu vidličku nemohou uchopit dva filozofové současně
- nemůže nastat deadlock
- nikdo nestrádá – musíme předpokládat, že semafor Volno má FIFO uspořádání

Asymetrické řešení Aspoň jeden filozof se bude řídit jiným algoritmem než ostatní. Necháme prvé čtyři filozofy provádět původní návrh, pátého necháme napřed čekat na svoji pravou vidličku a pak teprve na levou.

```
Vidlička:array(0..4) of semaphore :=(others=>1);

task body Filozof0123 is (I je identifikace filozofa)
begin loop
  Přemýšlení;
  Wait(Vidlička(I));
  Wait(Vidlička((I+1) mod 5));
  Jídlo;
  Signal(Vidlička(I));
  Signal(Vidlička((I+1) mod 5));
endloop;
end Filozof0123;

task body Filozof4 is
begin loop
  Přemýšlení;
  Wait(Vidlička(0));
  Wait(Vidlička(4));
  Jídlo;
  Signal(Vidlička(4));
  Signal(Vidlička(0));
endloop;
end Filozof4;
```

- jednu vidličku nemohou uchopit dva filozofové současně
- nemůže nastat deadlock
- nikdo nestrádá

Řešení pomocí monitoru

```
monitor Vidl-monitor;

  Fork:array(0..4) of integer range 0..2 :=(others=>2);
  OK_jídlo:array(0..4) of condition;

procedure Uchop_vidličku(I:integer) is
begin if fork(I)<>2 then Wait(OK_jídlo(I); endif;
  fork((I+1) mod 5) := fork((I+1) mod 5)-1;
  fork((I-1) mod 5) := fork((I-1) mod 5)-1;
end;

procedure Vrať_vidličku(I:integer) is
begin fork((I+1) mod 5) := fork((I+1) mod 5)+1;
  fork((I-1) mod 5) := fork((I-1) mod 5)+1;
  if fork((I+1) mod 5)=2 then Signal(OK_jídlo((I+1) mod 5)); endif;
  if fork((I-1) mod 5)=2 then Signal(OK_jídlo((I-1) mod 5)); endif;
end;
end Vidl-monitor;

task body Filozof is
begin loop Přemýšlení;
```

```

        Uchop_vidličku(I);
        Jídlo;
        Vrať_vidličku(I);
    endloop;
end;

```

- jednu vidličku nemohou uchopit dva filozofové současně
- nemůže nastat deadlock
- proces může strádat — dva filozofové se mohou spiknout, že vyhladoví svého společného souseda.

7.8 Synchronizace procesů – Ada

souběh procesů — rendezvous

```

task A;                task B;
task body A is        entry SYNCHRON end B;
begin                task body B is
    repeat            begin
        ...           repeat
    B.SYNCHRON        ...
        ...           ACCEPT SYNCHRON
    until false      until false
end;                  end;

```

- procesy jsou paralelně synchronní
- proces B čeká na synchronizačním místě
- aktivní proces si rande vyžaduje (staví se do fronty na čekání na B)
- po uskutečnění rande asynchronní proces pokračuje

7.8.1 Synchronní výměna dat

```

task A is                task B is
end A;                  entry KOM(x:in D1,y:out D2)
                        end B;
task body A is          task body B is
x1:D1; y1:D2;           x2:D2; y2:D2; z:D1; w:D2;
begin                  begin
    ...                ...
B.KOM(x1,y1);          accept KOM(x2:in D1;y2:out D2);
    ...                do
end A;                  z:=x2;
                        y2:=w;
                        end KOM;
                        ...
end B;

```

7.8.2 Selektivní čekání volaného na rendezvous

řeší problém zatvrdnutí pasivního procesu. p_1, \dots, p_n jsou zámky, proces nabízí n služeb současně, na služby se staví fronty aktivních procesů, u jedné dveří čeká proces maximálně 30 sekund, pak pokračuje. Proces vybírá procesy z front za false zámkem nedeterministicky.


```

select
  when p1 => accept E1 do ... end;
or
  when p2 => accept E2 do ... end;
or
  ...
or
  when pn => accept En do ... end;
or
  delay 30.0sec;
or
  terminate;
end select;

```

7.8.3 Selektivní volání randezvous

```

select          select
  T.E(...)      T.E(...)
  ...           ...
else           or delay 10sec
end select;    end select;

```

```
pack SEMAFORY is
```

```

task type B_SEM is
  entry WAIT;
  entry SIG;
end B_SEM;

```

```

task type SEM is
  entry INIT(N:in int);
  entry WAIT;
  entry SIG;
end SEM;

```

```
pack body SEMAFORY is
```

```

task body B_SEM is
  loop select accept WAIT;
              accept SIG;
              or terminate;
            end select;
  endloop;
end B_SEM;

```

```

task body SEM is
K:int; N1:int;
begin accept INIT(N:in int) do K:=N;N1:=N; end;
  loop select
    when K>0 > accept WAIT do K:=K-1;
  or
    accept SIG do if K<N1 then K:=K+1; end;
  or terminate;
  end select;
end loop;

```

```
end SEM;  
end SEMAFORY;
```

7.8.4 Producent a konzument v Adě

```
task body P is          task body K is  
loop  
  produkce C          loop  
  BUF.WRITE(C)        BUF.READ(C);  
  exit when C="#"     konzumace C  
end loop              exit when C="#"  
end P;                end loop;  
                      end K;  
  
task BUF is  
  entry READ(C:out char);  
  entry WRITE(C:in char);  
end BUF;  
  
task body BUF is  
  B:array(1..100)of char;  
  poč:int:=0; In,Out:range 1..100:=1;  
begin  
  loop  
    select  
      when poč<100 => accept WRITE(C:in char) do B(In):=C; end;  
                      In:=(In mod 100)+1; poč:=poč+1;  
    or when poč>0 => accept READ(C:out char) do C:=B(Out); end;  
                      Out:=(Out mod 100)+1; poč:=poč-1;  
    or terminate;  
  end select;  
end loop;  
end BUF;
```