Jazykové konstrukce pro paralelní výpočty

Paralelismus se vyskytuje na:

1. **Úrovni strojových instrukcí**
2. **Úrovni příkazů programovacího jazyka**
3. **Úrovni podprogramů**
4. **Úrovni programů**

Vývoj multiprocesorových architektur

1. **konec let 50. Jeden základní procesor a jeden či více speciálních procesorů pro I/O**
2. **polovina 60. víceprocesorové systémy užívané pro paralelní zpracování na úrovni programů**
3. **konec 60. víceprocesorové systémy užívané pro paralelní zpracování na instrukční úrovni**
4. **SIMD architektury (stejná instrukce současně zpracovávaná na více procesorech, na každém s jinými daty) - vektorové procesory**
5. **MIMD architektury (nezávisle pracující procesory, které mohou být synchronizovány)¨**

Paralelismus na úrovni podprogramů

Sekvenční výpočetní proces je v čase uspořádaná posloupnost operací

-Program	**- Procesy**	**- Procesor**
-Scénář	**- Role (Představení)**	**- Herec**

Paralelní procesy jsou vykonávány paralelně či pseudoparalelně

Kategorie paralelismu:

1. **Fyzický paralelismus (více procesorů pro více procesů)**
2. **Logický paralelismus (time-sharing jednoho procesoru, v programu je více procesů)**
3. **Kvaziparalelismus (korutiny)**

Nové problémy: **I. rychlostní závislost**
II. zablokování

Př. Z konta si vybírá SIPO 500,-Kč a obchodní dům 200,-Kč

Ad sériové zpracování

Zjištění stavu konta
Odečtení 500
Uložení nového stavu
Převod 500 na konto SIPO

Zjištění stavu konta
Odečtení 200
Uložení nového stavu
Převedení 200 na konto obch. domu

Výsledek bude OK

Ad paralelní zpracování dvěma procesy

Zjištění stavu konta	**Zjištění stavu konta**
Odečtení 500	**Odečtení 200**
Uložení nového stavu	**Uložení nového stavu**
Převod 500 na konto SIPO	**Převedení 200 na konto obch.domu**

Pokud výpočet neošetříme, může vlivem různých rychlostí být výsledný stav konta: (Původní -500) nebo (Původní -200) nebo (Původní - 700)

Operace výběrů z konta ale i vložení na konto musí být prováděny ve vzájemném vyloučení. Jsou to tzv. kritické sekce programu

Jak to řešit?

1. řešení Semafor = obdobnou funkci jako klíč od WC.
Operace: zaber(semafor)
uvolni(semafor)

Proces A	Proces B
Zaber(semafor S)	Zaber(semafor S)
Zjištění stavu konta	Zjištění stavu konta
Odečtení 500	Odečtení 200
Uložení nového stavu	Uložení nového stavu
Převod 500 na konto SIPO	Převedení 200 na k. obch.domu
Uvolni(semafor S)	Uvolni(semafor S)

Výsledný stav konta bude (Původní – 700)

Nebezpečnost semaforů: -Opomenutí semaforové operace (tj. ochránění)
-Možnost skoku do kritické sekce
-Možnost vzniku deadlocku, pokud nešikovně semafory použijete

II. Zablokování (deadlock)

Př. Procesy A, B oba pracují s konty (soubory, obecně zdroji) Z1 a Z2. K vyloučení vzniku nedeterminismu výpočtu, musí požádat o výlučný přístup (např. pomocí semaforů)
Pokud to provedou takto:

Proces A	Proces B
Zaber(semafor S1)	Zaber(semafor S2)
Zaber(semafor S2)	Zaber(semafor S1)

Bude docházet k deadlocku. Oba procesy se zastaví.
Jak tomu zabránit? (bankéřův algoritmus, uspořádání)

2. řešení Monitor

Monitor je modul (v OOP objekt), nad jehož daty mohou být prováděny pouze v něm definované operace. Provádí-li jeden z procesů některou monitorovou operaci, musí se ostatní procesy postavit do fronty, pokud rovněž chtějí provést některou monitorovou operaci . Ve frontě čekají, dokud se monitor neuvolní a přijde na ně řada.

Př. **Typ monitor konto**

-data: stav_ konta -operace: vyber(kolik) vlož(kolik)

Instance: **Mé_konto**
SIPO_konto
Obchodům_konto

Proces A

Mé_konto.vyber(500)
SIPO_konto.vlož(500)

Proces B

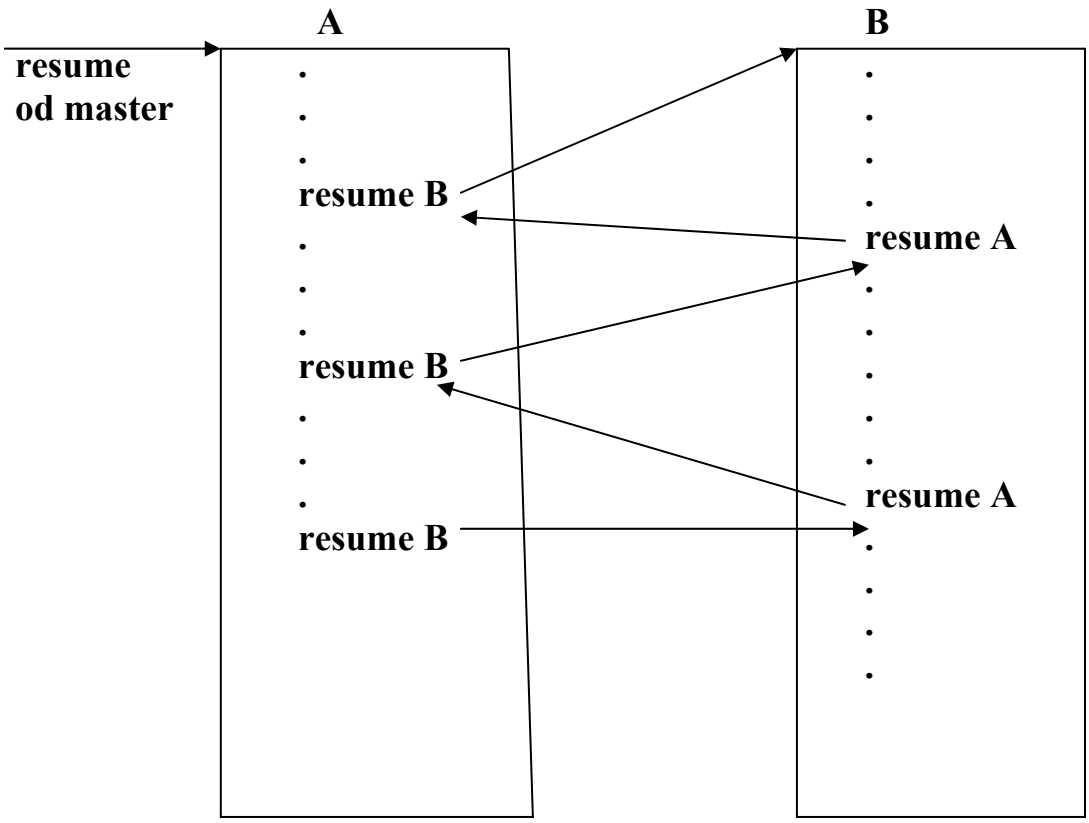
Mé_konto.vyber(200)
Obchodům_konto(vlož(200)

Pozn. V Javě jsou monitorem objekty, které jsou instancí třídy se synchronized metodami (viz později)

Korutiny

Speciální druh podprogramů – volající a volaný nejsou v relaci „master slave“

Jsou si rovni (symetričtí)
Mají více vstupních bodů
Zachovávají svůj stav mezi aktivacemi
V daném okamžiku je prováděna jen jedna



Master (není korutinou) vytvoří deklarací korutiny, ty provedou inicializační kód a vrátí mastru řízení.
Master příkazem resume spustí jednu z korutin
Příkaz resume slouží pro start i pro restart
Pakliže jedna z korutin dojde na konec svého programu, předá řízení mastru

Paralelismus na úrovni podprogramů

Procesy mohou být

1. **nekomunikující (neví o ostatních, navzájem si nepřekáží)**
2. **komunikující (např. producent a konzument)**
3. **soutěžící (např. o sdílený prostředek)**

V jazycích nazývány různě:

Vlákno (thread Java, C#) výpočtu v programu je sekvence míst programu, kterými výpočet prochází.

Úkol (task Ada) je programová jednotka (část programu), která může být prováděna paralelně s ostatními částmi programu. Každý úkol může představovat jedno vlákno.

Odlišnost vláken/úkolů/procesů od podprogramů:

- **mohou být implicitně spuštěny (Ada)**
- **programová jednotka, která je spouští nemusí být pozastavena**
- **po jejich skončení se řízení nemusí vracet do místa odkud byly odstartovány**

Způsoby jejich komunikace:

- **sdílené nelokální proměnné**
- **parametry**
- **zasílání zpráv**

Při synchronizaci musí A čekat na B (či naopak)

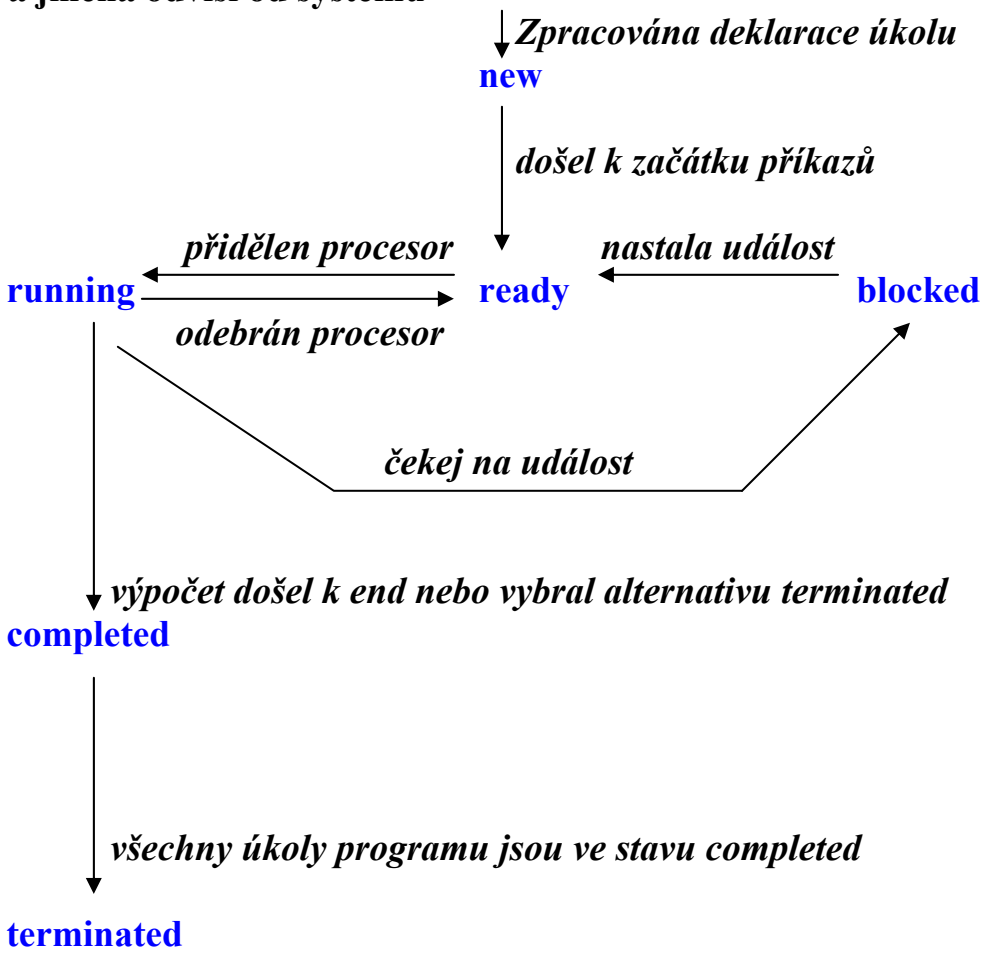
Při soutěžení sdílí A s B zdroj, který není použitelný simultánně (např. sdílený čítač) a vyžaduje přístup ve vzájemném vyloučení.

Části programu, které pracují ve vzájemném vyloučení se nazývají kritickými sekcemi.

Stavy úkolů (př. systému jazyka ADA):

1. **new**
2. **ready**
3. **running**
4. **blocked**
5. **completed**
6. **terminated**

Počet a jména odvisí od systému



Prostředky pro uskutečnění vzájemně výlučného přístupu ke sdíleným zdrojům

1. **Semafory**
2. **Monitory**
3. **Zasílání zpráv**

<u>Semafor</u> - datová struktura obsahující čítač a frontu pro ukládání deskriptorů úkolů. Má dvě operace - zaber a uvolni (P a V). Je použitelný jak pro soutěžící, tak pro spolupracující úkoly.

P a V jsou atomické operace

P(semafor) /* binární*/ /*zaber*/
if semafor = 1 then semafor := 0
else pozastav volající proces a dej ho do fronty na semafor

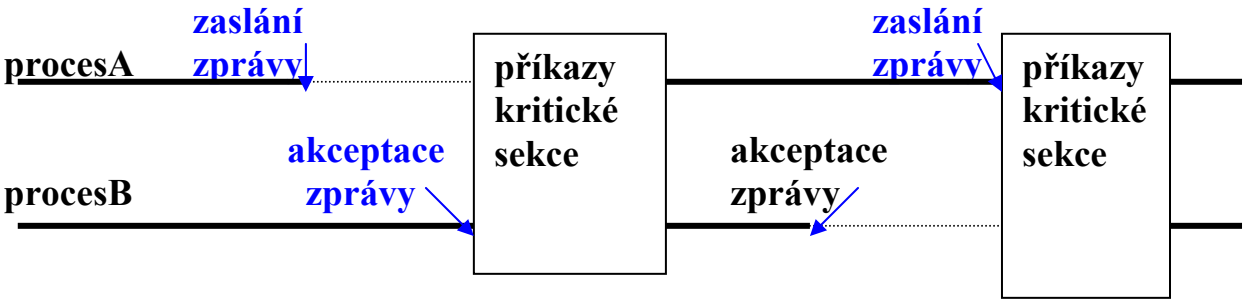
V(semafor) /* binární*/ /*uvolni*/
if fronta na semafor je prázdná then semafor := 1
else vyber prvého z fronty a aktivuj ho

Nebezpečnost semaforů -nelze kontrolovat řádnost použití → <u>může nastat deadlock</u>

Obdobou je použití signálů
Send(signal) --je akcí procesu 1
Wait(signal) --je akcí procesu 2 (rozdíl proti P a V)

<u>Monitor</u> - programový modul zapouzdřující data spolu s procedurami, které s daty pracují. Procedury mají vlastnost, že vždy jen jeden úkol/vlákno může provádět monitorovou proceduru, ostatní čekají ve frontě. (Ada95, Java)

<u>Zasílání zpráv</u> (synchronní způsob komunikace)



Vede k principu schůzky (rendezvous Ada)

Paralelní konstrukce v jazyce ADA

Paralelně proveditelnou jednotkou je task

```
task  T is
        Deklarace tzv. vstupů (entry)
end T;
```

} specifikační část

```
task body T is
        Lokální deklarace a příkazy
end T;
```

} tělo

Lze zavést i typ úkol zápisem *task type T is* ve specifikační části a použít jej k deklaraci úkolů. Oproti proměnným, úkolům nelze přiřazovat a porovnávat je.

Schůzka (rendezvous)

```
task ČIŠNÍK is
        entry PIVO (X: in INTEGER);
        entry PLATIT;
                …
end ČIŠNÍK ;


task body ČIŠNÍK is
 … --lokalni deklarace

  begin
   loop
        …--blouma u pultu
        accept PIVO (X: in INTEGER) do
        …--donese pivo
        end PIVO;
        …--sbira sklenice
        akcept PLATIT do
        …--inkasuje
        end PLATIT;
    end loop;
  end ČIŠNÍK ;
```

```
task HOST1 is
end HOST1;

task body HOST1 is
        …
        ČIŠNÍK.PIVO(2);

        …--pije pivo
        ČIŠNÍK.PLATIT;

        …
end HOST1;
```

Všechny úkoly se spustí současně, jakmile hlavní program dojde k begin své příkazové části (implicitní spuštění).

Př. 0 ukolyADA

```
with  TEXT_IO; use TEXT_IO;
procedure MAIN is
   task Ukol1 is
   end Ukol1;
   task body Ukol1 is
         begin
                  for i in 1 .. 10 loop
                                    delay 1.0;
                                 Put("prvni");
                  end loop;
   end Ukol1;

   task Ukol2 is
   end Ukol2;
   task body Ukol2 is
                 begin
                  for i in 1 .. 10 loop
                                    delay 1.0;
                                    Put("druhy");
                  end loop;
   end Ukol2;

         C: CHARACTER := 'A';
         i: INTEGER := 111;
begin
         for i in 1..20 loop
         delay 1.0;
                 Put("HLAVNI");
         end loop;
end MAIN;
```

Př.Schránka pro komunikaci producenta s konzumentem

```
task SCHRANKA is
        entry PUT(X: in INTEGER);
        entry GET(X: out INTEGER);
end SCHRANKA;

task body SCHRANKA is
        V: INTEGER;
begin
        loop
                accept PUT(X: in INTEGER) do
                        V := X;
                end PUT;
                accept GET(X: out INTEGER) do
                        X := V;
                end GET;
        end loop;
end SCHRANKA;
```

Producent: **SCHRANKA.PUT(456);**

Konzument: **SCHRANKA.GET(I);**

Pozn.: **Pořadí provádění operací PUT a GET je určeno pořadím příkazů. PUT a GET se musí střídat.**

Př. **Sdílená proměnná realizovaná úkolem (dovolí libovolné pořadí zapisování a čtení)**

```
task SDILENA is
        entry PUT(X: in INTEGER);
        entry GET(X: out INTEGER);
end SDILENA;

task body SDILENA is
        V: INTEGER;
begin
        loop
          select                --dovoli alternativni provedeni
                accept PUT(X: in INTEGER) do
                        V := X;
                end PUT;
          or
                accept GET(X: out INTEGER) do
                        X := V;
                end GET;
          or
                terminate;  --umozni ukolu skoncit aniz projde koncovym end
          end select;
        end loop;
end SDILENA;
```

generic -- Př.1 **STACK5ADA** umožňuje komunikaci producenta s konzumentem přes zásob.

```
SIZE: POSITIVE; --generický parametr, délka zásobníku
type POLOZKA is private; --generický parametr, typ hodnot
package STACK5 is
        task ZASOBNIK is
                entry VLOZ(X: in POLOZKA);
                entry UBER(X: out POLOZKA);
        end ZASOBNIK;
end STACK5;

package body STACK5 is
        task body ZASOBNIK is
        S: array (1..SIZE of POLOZKA;
        SP: INTEGER range 0..SIZE;
        begin
                SP := 0;
                loop
                        select
                                when SP < SIZE =>
                                        accept VLOZ(X : in POLOZKA) do
                                                SP := SP + 1;
                                                S(SP) := X;
                                        end VLOZ;
                        or
                                when SP > 0 =>
                                        accept UBER(X : out POLOZKA) do
                                                X := S(SP);
                                                SP := SP - 1;
                                        end UBER;
                        or
                                terminate;
                        end select;
                end loop;
        end ZASOBNIK;
end STACK5;

with STACK5, TEXT_IO; use TEXT_IO;
procedure MAIN is
        package STACKI is new STACK5(20, INTEGER);
        package STACKC is new STACK5(100, CHARACTER);
        C: CHARACTER := 'A';
        I: INTEGER := 111;
begin ...
        STACKC.ZASOBNIK.VLOZ('B'); -- …simulujeme producenta
        STACKI.ZASOBNIK.VLOZ(22);   --…může jím být samostatný úkol
        STACKC.ZASOBNIK.UBER(C);   -- …simulujeme konzumenta
end MAIN;
```

```
generic Size: INTEGER; --2NasobMatricesADA. Paralelni nasobeni matic A a B

package IntMatrices is
  type IntMatrix is array (1..Size,1..Size) OF INTEGER; --Size je generický parametr
  function ParMult(a,b: in IntMatrix) return IntMatrix; --exportuje IntMatrix a
ParMult
end;
package body IntMatrices is
  function ParMult(a,b: in IntMatrix) return IntMatrix is
    c:IntMatrix;
    task type Mult is  -- zavádím typ úkol
      entry DoRow (i: in INTEGER); entry EndRow ;
    end;
    task body Mult is
      iloc: INTEGER;
    begin
      accept DoRow (i: in INTEGER) do  iloc := i; end;
      for j in 1..Size loop
        c(iloc, j) := 0;
        for k in 1..Size loop
          c(iloc, j) := c(iloc,j) + a(iloc,k) * b(k,j);
        end loop;
      end loop;
      accept EndRow;
    end Mult;
  begin -- ParMult
    declare m: array (1..Size) of Mult; --pole úkolů (pro každý řádek matice jeden)
    begin
      for i in 1..Size loop
        m(i).DoRow(i);  --volání vstupů ≡ spuštění úkolů m[1] až m[Size]
      end loop;
      for i in 1..Size loop
        m(i).EndRow;      --volání vstupu EnDRow  ≡ ověření, že úkol m[i] skončil
      end loop;
    end;
    return c;
  end ParMult;                                    --konec funkce
end IntMatrices;                                          --konec balíku IntMatrices
with IntMatrices;  with Text_IO; use Text_IO;   --hlavní program
procedure Zkus is
    package Muj is new IntMatrices(2); --vygenerován balík násobení matic 2x2
    package Muj_IO is new Integer_IO(integer);
    use Muj, Muj_IO;
    A: IntMatrix := ((5,5),(5,5)); B: IntMatrix := ((3,3),(3,3));
    C: IntMatrix;
begin
        C:=ParMult(A,B);
        put(c(1,1)); put(c(1,2));put(c(2,1));put(c(2,2));
end Zkus;
with ADA.Text_IO; use ADA.Text_IO;         -- pr. 3Nasobeni Matice X vektorem U
```

```
procedure NAS_MAT is
  type RADKOVA_MATICE is array (INTEGER range <>) of FLOAT;
  type POINTER is access RADKOVA_MATICE;
  RAD: INTEGER;
  package F_IO is new Float_IO(Float); use F_IO;
  package I_IO is new Integer_IO(Integer); use I_IO;
  task type PARCIALNI_SOUCIN is
    entry PRIJATA_HODNOTA(P : out FLOAT);
    entry VYSLANA_HODNOTA(V_1, V_2: in RADKOVA_MATICE);
  end PARCIALNI_SOUCIN;
  task body PARCIALNI_SOUCIN is
    SOUCIN: FLOAT;
    VEKTOR_1, VEKTOR_2: POINTER;
  begin
    accept VYSLANA_HODNOTA(V_1,V_2: in RADKOVA_MATICE) do
          VEKTOR_1 := new RADKOVA_MATICE'(V_1); --inicializuje VEKTOR_1
          VEKTOR_2 := new RADKOVA_MATICE'(V_2); --hodnotou V_1, pod.V_2
    end VYSLANA_HODNOTA;
    SOUCIN := 0.0;
    for I in VEKTOR_1.all'RANGE   --ukazuje na RADKOVA_MATICE
          loop SOUCIN := SOUCIN + (VEKTOR_1(I)*VEKTOR_2(I));
    end loop;
    accept PRIJATA_HODNOTA(P: out FLOAT) do
          P := SOUCIN;
    end PRIJATA_HODNOTA;
  end PARCIALNI_SOUCIN;
begin
  get(RAD);
  declare
    type MATICE is array(1..RAD) of RADKOVA_MATICE(1..RAD);
    X:MATICE;
    U,P:RADKOVA_MATICE(1..RAD);
    PARALELNI_SOUCIN: array(U'RANGE) of PARCIALNI_SOUCIN;
  begin
    for I in X(1)'RANGE loop
          for J in X(2)'RANGE loop
            GET(X(I)(J));
          end loop;
    end loop;
    for I in U'RANGE loop GET(U(I));
    end loop; --tento cyklus musi byt zvlast od toho nasledujiciho
    for I in U'RANGE loop
          PARALELNI_SOUCIN(I).VYSLANA_HODNOTA(X(I), U);
    end loop;
    for I in P'RANGE loop
          PARALELNI_SOUCIN(I).PRIJATA_HODNOTA(P(I));
          PUT(P(I)); NEW_LINE;
    end loop;
  end;
end NAS_MAT;
```

PROTECTED typ a PROTECTED objekt (je to monitor)

Mechanismus rendezvous : -vede k vytváření dodatečných úkolů pro obsluhu sdílených dat
Typ protected má specifikační část (obsahuje přístupový protokol k objektu) a tělo (obsahuje implementační detaily)
Umožňuje synchronizovaný přístup k privátním datům objektů (bez zavádění dodatečných úkolů) pomocí operací ve tvaru:

1. **funkcí – může je provádět lib. počet klientů, pokud není právě volána procedura nebo vstup**
2. **procedur – pouze 1 volající ji může provádět a nesmí současně být volána funkce či vstup**
3. **vstupů – jako procedury, ale jen při splněné tzv. bariéře**

př.

```
protected Variable is
        function Read return Item ;
        procedure Write ( New_Value : Item ) ;
private
        Data : Item ;
end Variable;

protected body Variable is
        function Read return Item is
        begin
                return Data ;
        end Read ;
        procedure Write ( New_Value : Item ) is
        begin
                Data := New_Value;
        end Write ;
end Variable;
```

Chráněný objekt Variable poskytuje řízený přístup k privátní proměnné Data. Způsob volání prostřednictvím „ .“ notace:

```
X := Variable . Read ;
. . .
Variable . Write ( New_Value => Y ) ;
```

Realizuje monitor = pasivnost
= programový modul zapouzdřující data a procedury s nimi pracující. Procedury mají vlastnost, že vždy jen jeden proces může provádět monitorovou proceduru, ostatní čekají ve frontě na monitor. Funkci může provádět více procesů najednou

Chráněné typy mohou mít kromě proc. a fcí i vstupy (entry) s bariérou ≅ accept úkolu s hlídkou (ale pozměněný efekt vyhodnocení bariéry)

př. Omezený buffer

```
protected type Bounded_Buffer is
        entry Put ( X : in Item ) ;
        entry Get ( X : out Item ) ;
private
        A : Item_Array ( 1 . . Max );
        I , J : Integer range 1 . . Max := 1;
        Count : Integer range 0 . . Max := 0 ;
end Bounded_Buffer;

protected body Bounded_Buffer is
        entry Put ( X : in Item ) when Count < Max is
        begin
                A ( I ) := X ;
                I := I mod Max + 1 ; Count := Count + 1;
        end Put;
        entry Get ( X : out Item ) when Count > 0 is
        begin
                X := A ( J ) ;
                J := J mod Max + 1 ; Count := Count - 1 ;
        end Get;
end Bounded_Buffer ;
```

př. použití:

```
Muj_Buffer : Bounded_buffer ;
Muj_Buffer . Put ( X ) ;
```

1. Při volání se vyhodnotí bariéra a je-li false je volání frontováno.
2. Na konci exekuce každého těla entry či procedury (ale ne funkce - ta nemění stav) se přepočítají všechny bariery a ta volání z front, která mají nyní bariéry true se všechna umožní provést.

Vstupy (stejně jako procedury) jsou prováděny v režimu vzájemného vyloučení

Nebylo potřeba vytvářet úkol

Chráněné proměnné jsou efektivním způsobem realizace synchronizačních prostředků

př. Obecný semafor

```
protected  type  Semafor ( Start : Integer := 1 ) is
        entry  Zaber ;                -- P   (Passeren)
        entry  Uvolni ;               -- V    (Vrijmaken)
        function Citac  return  Integer;
private
        CC : Integer := Start ;
end  Semafor ;

protected  body  Semafor  is
        entry Zaber  when  CC > 0  is
        begin
                CC := CC - 1 ;
        end  Zaber;
        entry  Uvolni  when  CC <  Start  is
        begin
                CC := CC + 1 ;
        end  Uvolni ;
        function  Citac  return  Integer  is
        begin
                return  CC ;
        end  Citac ;
end  Semafor ;
```

Semafor je chráněný typ s diskriminantem Start (t.j. počet chráněných zdrojů). Diskriminant může mít i úkol.

Zdroje informace: http://www.adahome.com/Tutorials/Lovelace/lovelace.htm