Amdahlův zákon

Když se něco dělá paralelně, může se získat výsledek rychleji, ale ne vždy
(1 žena porodí dítě za 9 měsíců, ale 9 žen neporodí dítě za 1 měsíc)

Amdahlův Zákon

Určuje urychlení výpočtu při užití více procesorů

Urychlení je limitováno sekvenčními částmi výpočtu

Obecně: $$\frac{1}{\sum_{k=0..n} \left(P_k / S_k \right)}$$

P_k % instrukcí, které lze urychlit

S_k % multiplikátor urychlení

n počet různých úseků programu

k je index úseku

Je-li v programu jeden paralelizovatelý úsek s P % kódu, pak

$$\frac{1}{(1-P) + (P / S)} = \frac{1}{\frac{(1 - P)}{1} + \frac{P}{S}}$$

Amdahlův zákon

Př.Paralelizovatelný úsek zabírá 60% kódu a lze jej urychlit 100 krát. $\Rightarrow$
celkové urychlení je 1/((1-0,6)+(0,6/100)) = 1/(0.4 + 0,006) $\approx$ **2.5**

Př. 50 % kódu lze urychlit libovolně $\Rightarrow$
celkové urychlení je 1/(1 – 0.5) = **2**

Př. úseky

P1 = 11 %,	P2 = 48 %	P3 = 23 %	P4 = 18 %
S1 = 1	S2 = 1,6	S3 = 20	S4 = 5

$$\text{Urychlení je } \frac{1}{\frac{0,11}{1} + \frac{0,48}{1,6} + \frac{0,23}{20} + \frac{0,18}{5}} \approx \mathbf{2,19}$$

Kromě urychlení zajišťují paralelní konstrukce také spolupráci výpočtů

Jazykové konstrukce pro paralelní výpočty

Paralelismus se vyskytuje na:

Úrovni strojových instrukcí	**je záležitostí hardware**
Úrovni příkazů programovacího jazyka	**toho si všimneme**
Úrovni podprogramů	**to probereme**
Úrovni programů	**je záležitostí Oper. Syst.**

Vývoj multiprocesorových architektur:

-konec let 50. Jeden základní procesor a jeden či více speciálních procesorů pro I/O

-polovina 60. víceprocesorové systémy užívané pro paralelní zpracování na úrovni programů

-konec 60. víceprocesorové systémy užívané pro paralelní zpracování na instrukční úrovni

Druhy počítačových architektur pro paralelní výpočty:

-SIMD architektury (simple instruction multiple data = stejná instrukce současně zpracovávaná na více procesorech, na každém s jinými daty) - vektorové procesory

-MIMD architektury (multiple instruction multiple data = nezávisle pracující procesory, které mohou být synchronizovány)

Někdy se rozlišuje	**-Parallel programming**	**= cílem je urychlení výpočtu**
	-Concurrent progr.	**= cílem je správná spolupráce programů**

Také lze dělit na

Paralelismus implicitní (zajistí překladač) nebo explicitní (zařizuje programátor)

Realizace buď konstrukcemi jazyka nebo knihovny (u tradičních jazyků)

Jazykové konstrukce pro paralelní výpočty

Paralelismus na úrovni podprogramů

Sekvenční výpočetní proces je v čase uspořádaná posloupnost operací =vlákno.

Definice vláken a procesů se různí. Obvykle proces obsahuje 1 či více vláken a vlákna uvnitř jednoho procesu sdílí zdroje, zatímco různé procesy ne.

Analogie s divadelním představením:

-Program obsahuje - Procesy (vlákna) provádí je - Procesor(y)

-Scénář divadla obsahuje - Role provádí je - Herec(herci)

Paralelní procesy jsou vykonávány paralelně či pseudoparalelně (pseudo=nepravý)

Kategorie paralelismu:

- **Fyzický paralelismus (má více procesorů pro více procesů)**
- **Logický paralelismus (time-sharing jednoho procesoru, v programu je více procesů)**
- **Kvaziparalelismus (kvazi=zdánlivě, př. korutiny v některých jazycích)**

Paralelně prováděné podprogramy musí nějak komunikovat:

1. **Přes sdílenou paměť (Java, C#), musí se zamykat přístup k paměti**
2. **Předáváním zpráv (Occam, Ada), vyžaduje potvrzení o přijetí zprávy**

Jazykové konstrukce pro paralelní výpočty

Nové problémy: **rychlostní závislost,**
uvíznutí (deadlock = vzájemné neuvolnění prostředků pro jiného),
vyhladovění (obdržení příliš krátkého času k zajištění progresu),
livelock (obdoba uvíznutí, ale nejsou blokovány čekáním, zaměstnávají se navzájem (after you - after you efekt)

Př. **Z konta si vybírá SIPO 500,-Kč a obchodní dům 200,-Kč**
Ad sériové zpracování

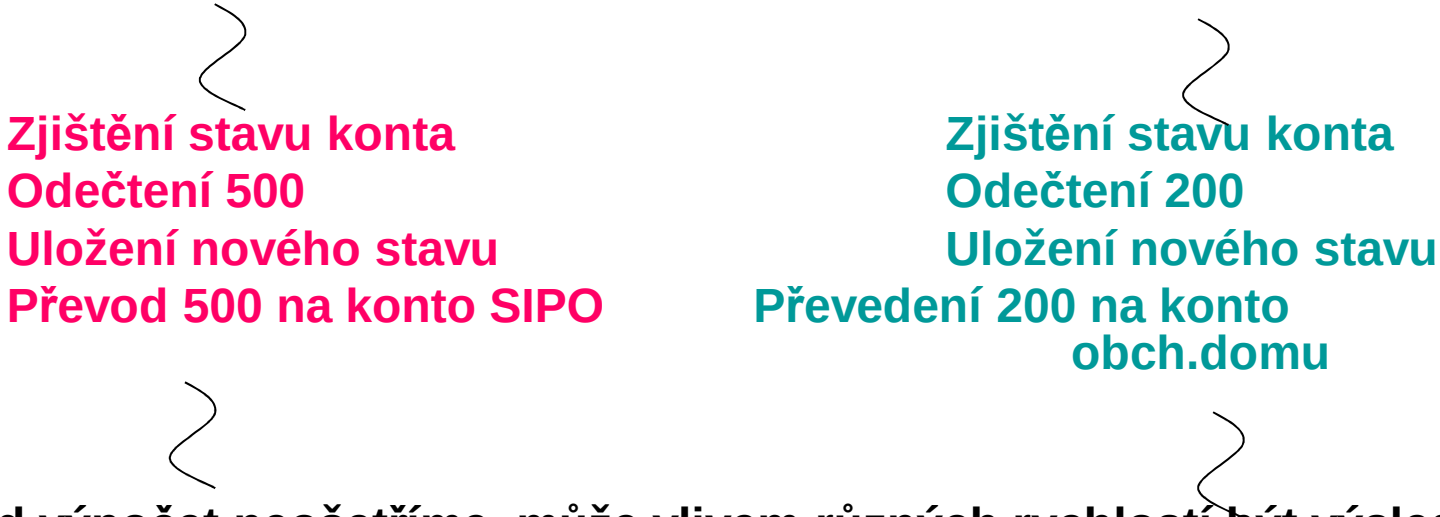
Zjištění stavu konta
Odečtení 500
Uložení nového stavu
Převod 500 na konto SIPO

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

Výsledek bude OK

Jazykové konstrukce pro paralelní výpočty

Ad paralelní zpracování dvěma procesy



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

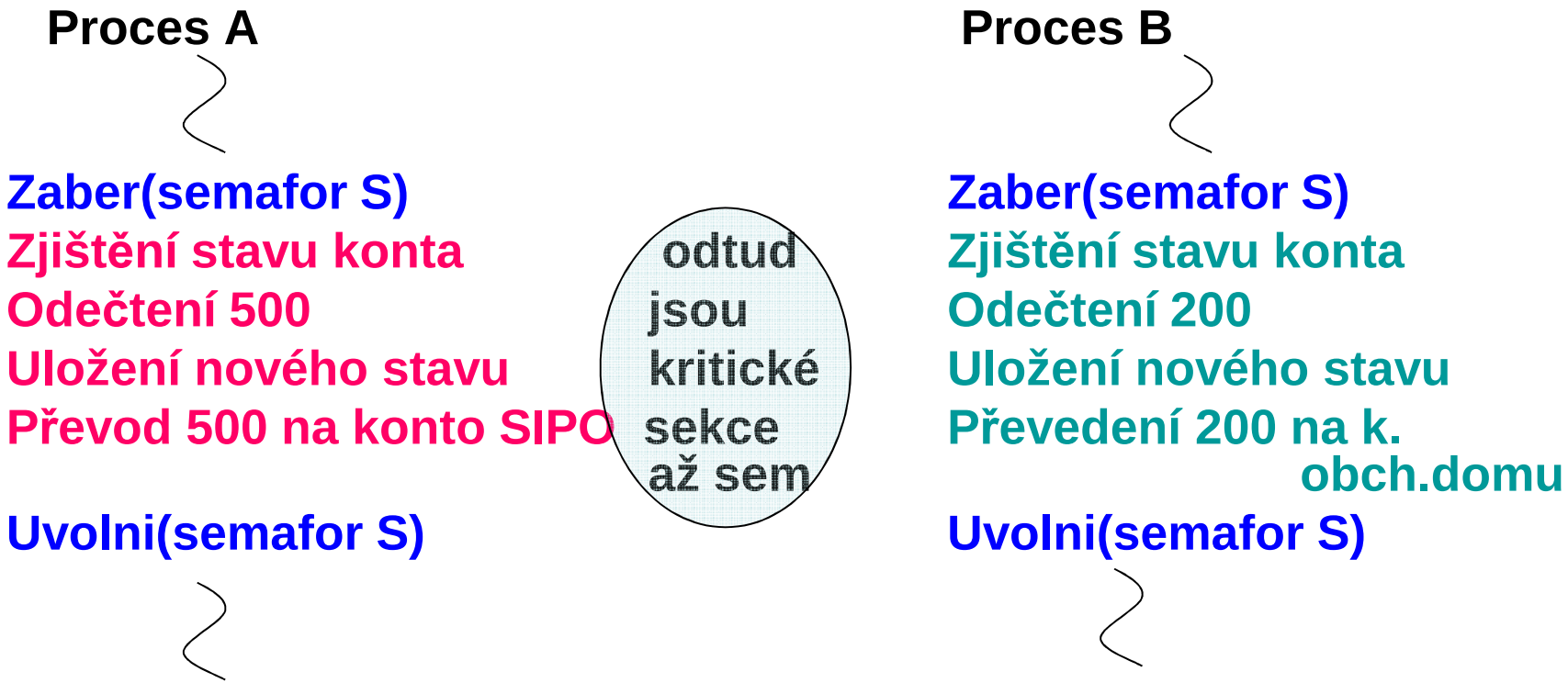
Operace výběrů z konta ale i vložení na konto musí být prováděny ve vzájemném vyloučení. Jsou to tzv. kritické sekce programu

Jak to řešit?

1. řešení Semafor = obdobnou funkci jako klíč od WC nebo návěstidlo železnice (jen jeden může do sdíleného místa).

Operace: zaber(semafor)
uvolni(semafor)

Jazykové konstrukce pro paralelní výpočty



Výsledný stav konta bude (Původní – 700)

Nebezpečnost semaforů:
-Opomenutí semaforové operace (tj. neochránění krit. sekce)
-Možnost skoku do kritické sekce
-Možnost vzniku deadlocku (viz další), pokud semafor neuvolníme

Jazykové konstrukce pro paralelní výpočty

Uvíznutí (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:

Bude docházet k deadlocku. Každý z procesů bude mít jeden ze zdrojů, potřebuje ale oba zdroje. Oba procesy se zastaví.

Jak tomu zabránit? (např. tzv. bankéřův algoritmus nebo přidělování zdrojů v uspořádání tzn. pokud nemáš S1, nemůžeš žádat S2, …)

Jazykové konstrukce pro paralelní výpočty

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)

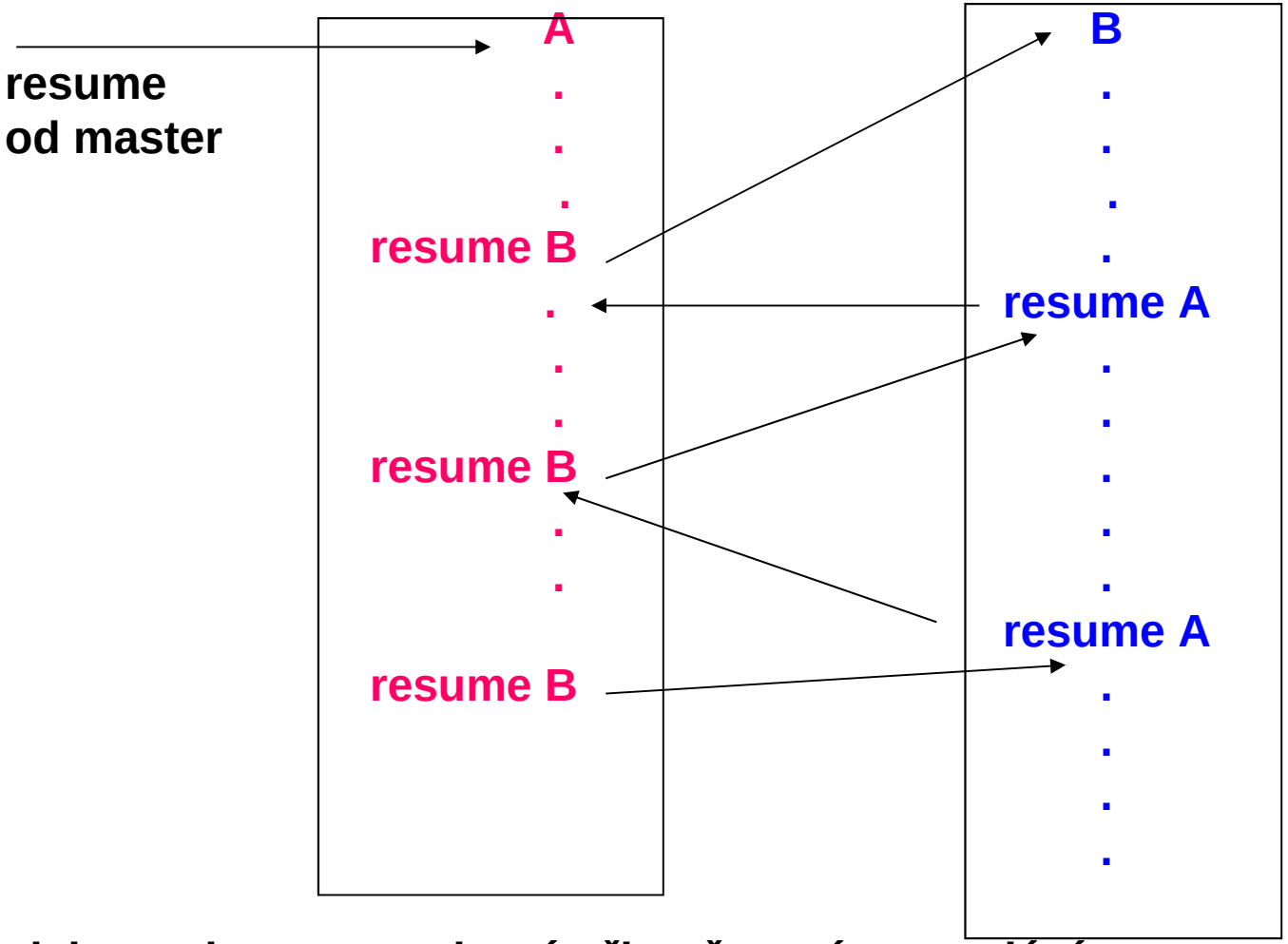
Proces B

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

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

Jazykové konstrukce pro paralelní výpočty

Korutiny =kvaziparalelní prostředek jazyků Modula, Simula, Interlisp



Představte si je jako podprogramy, které při opětovném vyvolání se nespustí od začátku, ale od místa ve kterém příkazem resume předaly řízení druhému

Jazykové konstrukce pro paralelní výpočty

Korutiny (kvaziparalelismus)

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 (trochu poddrobněji)

Procesy mohou být

-nekomunikující (neví o ostatních, navzájem si nepřekáží)

-komunikující (např. producent a konzument)

-soutěžící (např. o sdílený prostředek)

V jazycích nazývány různě:

Vlákno výpočtu v programu (thread Java, C#, Python) 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

-předávané parametry

-zasílání zpráv

Při synchronizaci musí A čekat na B (či naopak)

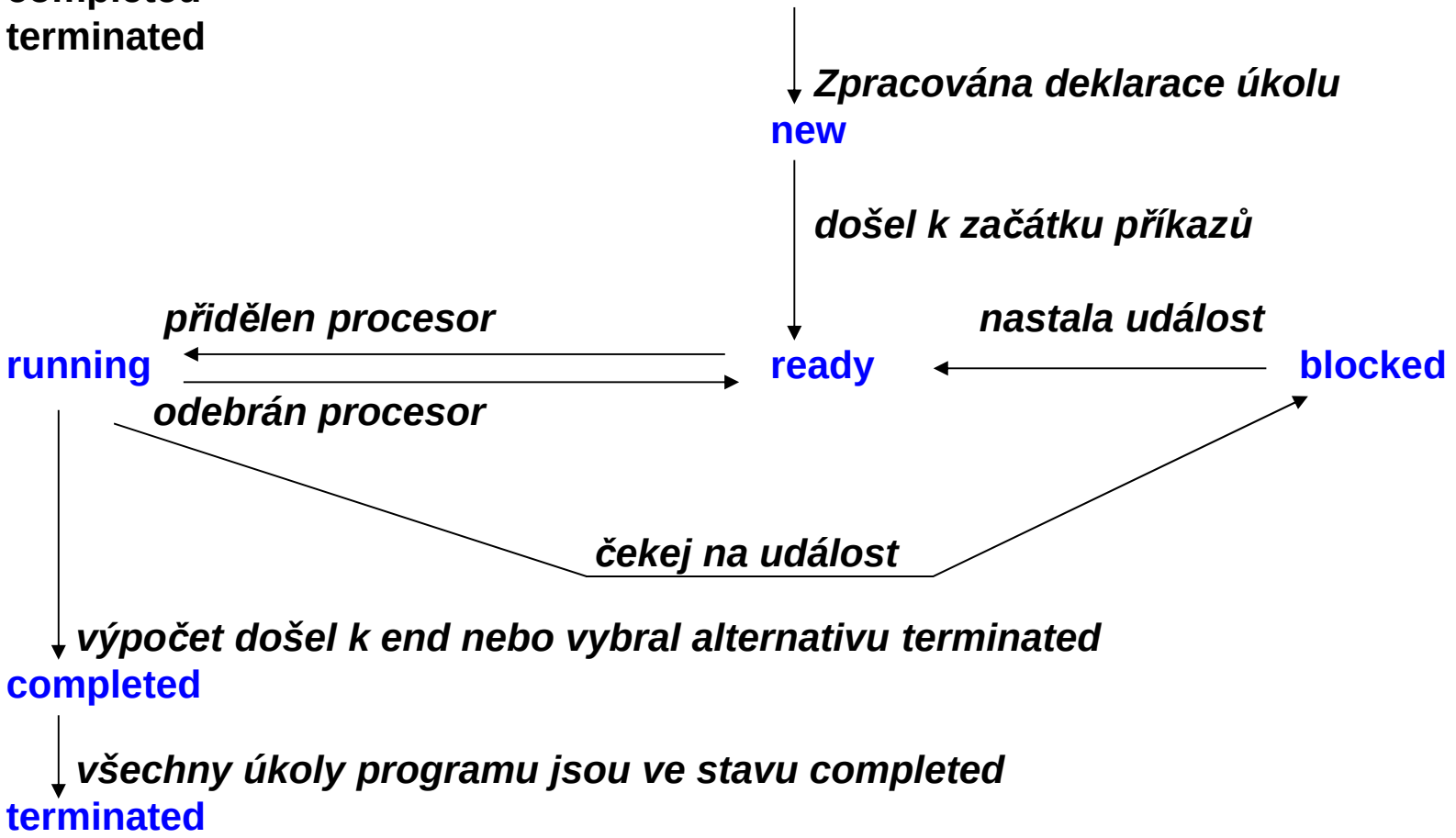
Při soutěžení sdílí A s B zdroj, který není použitelný simultánně (např. sdílený čítač) a vyžaduje přístup ve vzájemném vyloučení.

Části programu, které pracují ve vzájemném vyloučení jsou kritickými sekcemi.

Jazykové konstrukce pro paralelní výpočty

Stavy úkolů (př. systému jazyka ADA):

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



Jazykové konstrukce pro paralelní výpočty

Probereme princip synchronizačních a komunikačních prostředků (semafor, monitor, které pracují se sdílenou pamětí a zasílání zpráv, které lze použít i v distribuovaných výpočtech.

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

P a V jsou atomické operace (tzn.nepřerušitelné)

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

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

Pokud chráníme ne 1 ale n zdrojů, nabývá čítač hodnoty 0..n

Nebezpečnost semaforů -

nelze kontrolovat řádnost použití → může nastat deadlock když neuvolníme

→ „ „ rychlostní závislost když nezabereme

Jazykové konstrukce pro paralelní výpočty

Obdobou je použití signálů

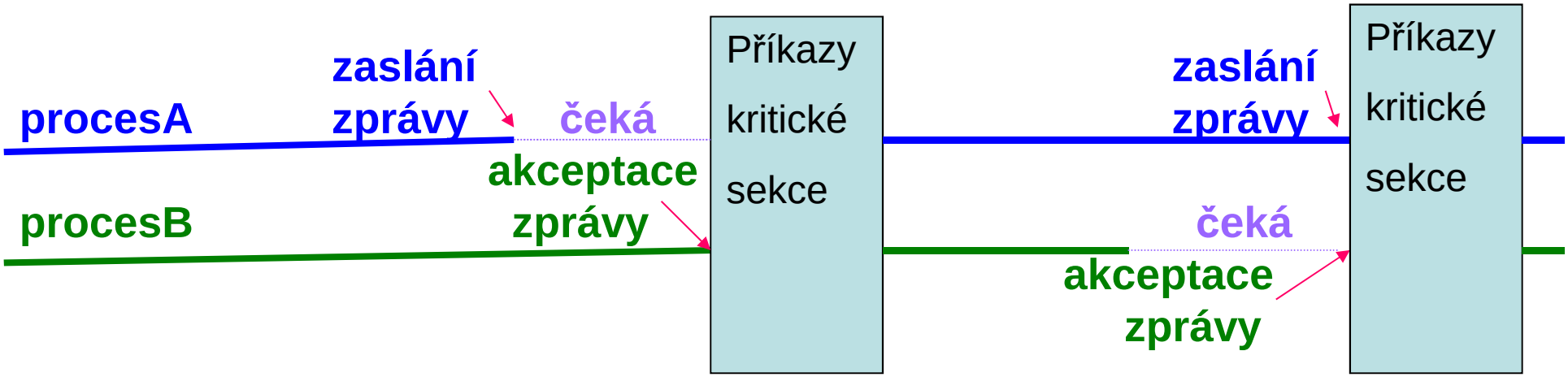
Send(signal) --je akcí procesu 1

Wait(signal) --je akcí procesu 2 (rozdíl proti P a V)

Monitor - 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. (pro Ada a zejména Java si probereme důkladněji)

Zasílání zpráv (je to synchronní způsob komunikace)

(Při asynchronní se komunikuje přes vyrovnávací paměť)



Vede k principu schůzky (rendezvous Ada)

Jazykové konstrukce pro paralelní výpočty

Paralelní konstrukce v jazyce ADA (pouze jako přehled možností**)**
Paralelně proveditelnou jednotkou je task

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

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

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.

Primárním prostředkem komunikace úkolů je schůzka (rendezvous)
Princip rendezvous popíšeme neformálně %

Jazykové konstrukce pro paralelní výpočty

```
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 X piv
        end PIVO;
        …--sbira sklenice
        akcept PLATIT do
        …--inkasuje
        end PLATIT;
     end loop;
  end ČIŠNÍK ;
```

```
task HOST1 is  --nemá zadny vstup
end HOST1;

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

                …--pije pivo
                ČIŠNÍK.PLATIT;

                                …
end HOST1;
```

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

```
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. 0 ukolyADA

nedeterminovanost pořadí provádění úkolů (tj.tisku)

zde nekomunikují

```
with  TEXT_IO; use TEXT_IO;      --Př. 01RychlostniZavislostiADA. Různá zdrzeni daji ruzne vysledky
procedure MAIN is
   package Muj_IO is new Integer_IO(integer);
   use Muj_IO;
   konto: INTEGER := 1000;
   task SIPO is
   end SIPO;
   task body SIPO is
         K: INTEGER;
         begin
            k:= konto;
            delay 1.0;    --(+/-) zpusobi zmenu rychlosti, zkus delay 1.5
            Put_line("platim SIPO ");
            konto:= k-500;
   end SIPO;
   task Obchodak is
   end Obchodak;
   task body Obchodak is
         K: INTEGER;
         begin
            k:= konto;
            delay 1.0;    --(+/-) zpusobi zmenu rychlosti
            Put_line("platim v obchode ");
            konto:= k-200;
   end Obchodak;
begin
   delay 2.0;
         Put("Vysledne konto = ");
         put(konto);
end MAIN;
```

Jazykové konstrukce pro paralelní výpočty

Úkoly mohou mít „vstupy“ (entry), pomocí nichž se mohou synchronizovat a realizovat rendezvous (schůzku)

Př.Schránka pro komunikaci producenta s konzumentem. Schránka je jeden úkol, producent i konzument by byly další (zde nezapsané) úkoly

```
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 –zde čeká, dokud producent nezavolá PUT
                        V := X;
            end PUT;
            accept GET(X: out INTEGER) do –zde čeká, dokud konzument nezavolá GET
                        X := V;
            end GET;
        end loop;
end SCHRANKA; --napred do ni musí vlozit, pak ji muze vybrat
```

Jazykové konstrukce pro paralelní výpočty

Konzument a producent jsou také úkoly a komunikují např.

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.**

To nevyhovuje pro případ sdílené proměnné, do které je možné zapisovat a číst v libovolném pořadí, ne však současně.

Libovolné pořadí volání vstupů dovoluje konstrukce select

```
select
        <příkaz accept neco>
or
        <příkaz accept něco jineho>
or
        ...
or
        terminate
end select;
```

Př. Sdílená proměnná realizovaná úkolem (dovolující 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;        -- nevýhodou je, že sdílená proměnná je také úkolem, takže vyžaduje
                    -- režii s tím spojenou. Proto ADA zavedla tzv. protected proměnné a
                    -- protected typy, které realizují monitor.
```

PROTECTED typ a PROTECTED objekt (je to monitor)

Mechanismus rendezvous : -vedl 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.

Vstupy (entry) s bariérou ≅ accept úkolu s hlídkou (ale pozměněný efekt vyhodnocení bariéry)

```
protected type Bounded_Buffer  is --př. Buffer s omezenou delkou
    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  --vstup s barierou
    begin
            A ( I )  :=  X ;
            I  :=  I  mod  Max + 1 ; Count := Count + 1;
    end  Put;
    entry  Get ( X : out  Item )  when  Count  >  0  is  --vstup s barierou

    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 ) ;
```

Při volání se vyhodnotí bariéra a je-li false je volání frontováno.

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

Př. 02MonitorADA řeší s použitím protected proměnné *konto* úlohu manipulace s kontem tj. možnost simultánních výběrů, příp. vkladů

```
with  TEXT_IO; use TEXT_IO;
procedure MAIN is
        package Muj_IO is new Integer_IO(integer);
        use Muj_IO;
        protected konto is
            function Read return INTEGER;
            procedure Vyber (kolik: INTEGER);
            procedure Vloz  (kolik: INTEGER);
            private
            Stav: INTEGER := 1000;
        end Konto;
        protected body Konto is
            function Read return INTEGER is
            begin return Stav;
            end Read;
            procedure Vyber (kolik:Integer) is
                        k: INTEGER;
            begin
              k:= Stav;
              Put_line("vybiram z konta");
              Stav := k - kolik;
            end Vyber;
            procedure Vloz (kolik:Integer) is
                        k: INTEGER;
            begin
              k:= Stav;
              Put("vkladam na konto");
              Stav := k + kolik;
            end Vloz;
   end Konto;
```

--pokračování př. 02MonitorADA

```
 task SIPO is
   end SIPO;
   task body SIPO is
         k: INTEGER;
         begin
            k:= konto.Read;
            delay 1.0;    --(+/-) zpusobi zmenu rychlosti, vysledek však nezmneni
            Put_line("platim SIPO");
            Konto.Vyber(500);
   end SIPO;
   task Obchodak is
   end Obchodak;
   task body Obchodak is
         k: INTEGER;
         begin
            k:= konto.Read;
            delay 1.0;    --(+/-) zpusobi zmenu rychlosti
            Put_line("platim v obchode");
            Konto.Vyber(200);
         end Obchodak;

begin  -- hlavni program jen vypise vysledek
         delay 2.0;
            Put("Vysledne konto = ");
            put (konto.Read);
end MAIN;
```

Jazykové konstrukce pro paralelní výpočty

Další příklady ADY na generický paralelní zásobník a paralelní násobení si můžete zkusit (prohlédnout), vykládat je nebudeme.

Paralelismus na úrovni příkazů jazyka

Jazyk Occam

Je imperativní paralelní jazyk

```
SEQ  --uvozuje sekvenční provádění
    x := x + 1
    y := x * x

PAR  --uvozuje paralelní provádění
    p()
    q()

WHILE next <> eof      -- da se to v konstrukcích kombinovat
     SEQ
         x := next
         PAR
             in  ?  Next
             out  !  x  *  x
```

Paralelismus na úrovni příkazů jazyka

High performance Fortran

Založen na modelu SIMD:

- **výpočet je popsán jednovláknovým programem**
- **proměnné (obvykle pole) lze distribuovat mezi více procesorů**
- **distribuce, přístup k proměnným a synchronizace procesorů je zabezpečena kompilátorem**

Př. Takto normálně Fortran dělí trojúh. matici diagonálními prvky

```
REAL DIMENSION (1000, 1000) :: A, B
INTEGER  I, J
…
DO I = 2, N
   DO J = 1, I – 1
         A(I, J) = A(I, J) / A(I, I)
   END DO
END DO
```

Paralelismus na úrovni příkazů jazyka

High Performance Fortran to ale umí i paralelně příkazem

FORALL (I = 2 : N, J = 1 : N, J .LT. I) A(I, J) = A(I, J) / A(I, I)

kterým se nahradí ty vnořené cykly

FORALL představuje zobecněný přiřazovací příkaz (a ne smyčku)

Distribuci výpočtu na více procesorů provede překladač.

FORALL lze použít, pokud je zaručeno, že výsledky sériového i paralelního zpracování budou identické.

Paralelismus na úrovni programů

Pouze celý program může v tomto případě být paralelní aktivitou.

Je to věcí operačního systému

Např. příkazem Unixu fork vznikne potomek, přesná kopie volajícího procesu, včetně proměnných

Následující příklad zjednodušeně ukazuje princip na paralelním násobení matic, kde se vytvoří 10 procesů s myid 0,1,…,9.

Procesy vyjadřují zda jsou rodič nebo potomek pomocí návratové hodnoty fork. Na základě testu hodnoty fork() pak procesy rodič a děti mohou provádět odlišný kód.

Po volání fork může být proces ukončen voláním exit.

Synchronizaci procesů provádí příkaz wait, kterým rodič pozastaví svoji činnost až do ukončení svých dětí.

```
#define SIZE 100
#define NUMPROCS 10
int a[SIZE] [SIZE], b[SIZE] [SIZE], c[SIZE] [SIZE];
void multiply(int myid)
{ int i, j, k;
  for (i = myid; i < SIZE; i+= NUMPROCS)
    for (j = 0; j < SIZE; ++j)
    { c[i][j] = 0;
      for (k = 0; k < SIZE; ++k)
        c[i][j] += a[i][k] * b[k][j];
    }
}
main()
{ int myid;
  /* prikazy vstupu a, b */
  for (myid = 0; myid < NUMPROCS; ++myid)
    if (fork() == 0)
    { multiply(myid);
      exit(0);}
  for (myid = 0; myid < NUMPROCS; ++myid)
    wait(0);
  /* prikazy vystupu c */
  return 0;
}
```