PROGRAMOVÉ STRUKTURY: PARALELNÍ PROGRAMOVÁNÍ

Amdahlův zákon, jazykové konstrukce pro paralelní výpočty, paralelismus

21. Února 2009

© jstein

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: $$\frac{1}{\sum_{k=0..n} \frac{P_k}{S_k}}$$

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

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

Amdahlův zákon (2)

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

Př. 50 % kódu lze urychlit libovolně ⇒
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

Urychlení je 1 / (0,11/1 + 0,48/1,6 + 0,23/20 + 0,18/5) ≈ **2,19**

Kromě urychlení zajišťují paralelní konstrukce také spolupráci výpočtů

Paralelismus

- 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 Operačního Systému
- Vývoj multiprocesorových architektur:
  - konec 50. let – 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
    - stejná instrukce současně zpracovávaná na více procesorech
    - na každém s jinými daty
    - vektorové procesory
  - MIMD architektury
    - nezávisle pracující procesory, které mohou být synchronizovány

Paralelismus (2)

- Někdy se rozlišuje
  - Parallel programming = cílem je urychlení výpočtu
  - Concurrent progr.= cílem je správná spolupráce programů
- Paralelismus implicitní (zajistí překladač) nebo **explicitní** (zařizuje programátor)
- Realizace buď konstrukcemi jazyka nebo knihovny (u tradičních jazyků)

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.
- 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)
    - Korutiny – speciální druh podprogramů, kdy volající a volaný jsou si rovni (symetrie), mají více vstupních bodů a zachovávají svůj stav mezi aktivacemi.
- Paralelně prováděné podprogramy musí nějak komunikovat
  - Přes sdílenou paměť (Java, C#), musí se zamykat přístup k paměti
  - Předáváním zpráv (Occam, Ada), vyžaduje potvrzení o přijetí zprávy

Problémy paralelního zpracování

- Nové problémy
  - rychlostní závislost
  - uvíznutí (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

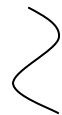
...

Výsledek bude OK

Rychlostní závislost

- Ad paralelní zpracování dvěma procesy

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

- 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) a uvolni(semafor)

Semafory

Proces A		Proces B
Zaber(semafor S)		Zaber(semafor S)
Zjištění stavu konta	odtud	Zjištění stavu konta
Odečtení 500	jsou	Odečtení 200
Uložení nového stavu	kritické	Uložení nového stavu
Převod 500 na konto SIPO	sekce až sem	Převedení 200 na k. OD
Uvolni(semafor S)		Uvolni(semafor S)

- Výsledný stav konta bude (Původní – 700)
- Nebezpečnost semaforů:
  - Opomenutí semaforové operace (tj. ochránění krit. sekce)
  - Možnost skoku do kritické sekce
  - Možnost vzniku deadlocku (viz další), pokud semafor neuvolníme

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:

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

- 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 = pokud nemáš S1, nemůžeš žádat S2, …)

Monitor

- 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)

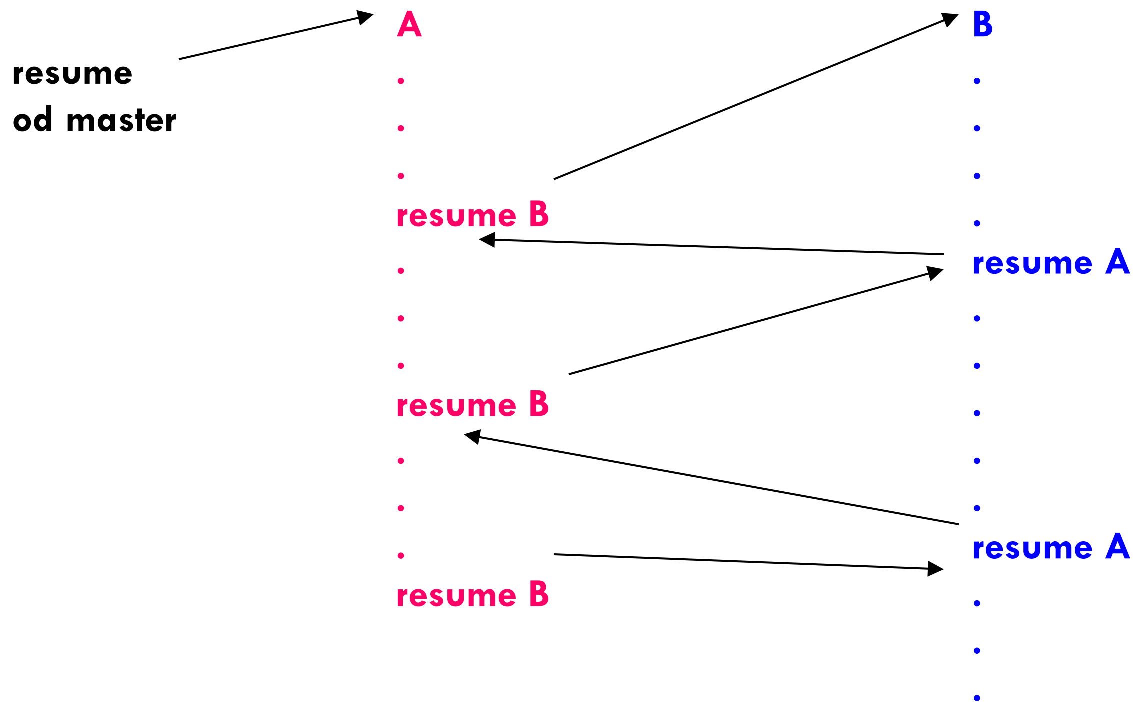
Instance: Mé_konto, SIPO_konto, Obchodům_konto

Proces A	Proces B
Mé_konto.vyber(500)	Mé_konto.vyber(200)
SIPO_konto.vlož(500)	Obchodům_konto(vlož(200)

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

Korutiny (kvaziparalelismus)

- 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



Korutiny (kvaziparalelismus) (2)

- 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
  - 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

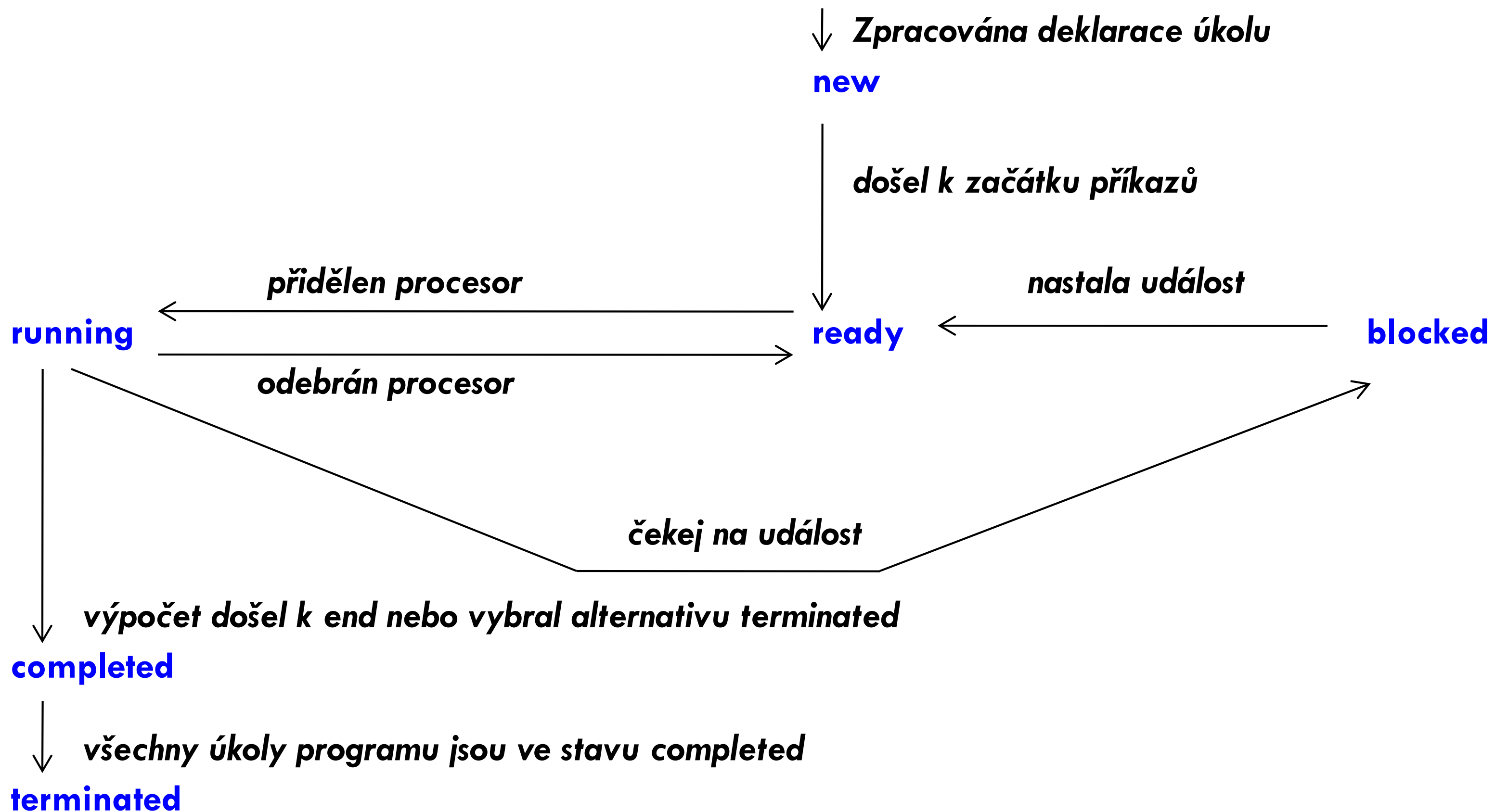
Paralelismus na úrovni podprogramů (2)

- Způsoby jejich komunikace:
  - sdílené nelokální proměnné
  - předávané parametry
  - zasílání zpráv
- Při synchronizaci musí A čekat na B (či naopak) (viz další slajd)
- 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.

Bariéra

- Synchronizační konstrukce
- Bariéra ve zdrojovém kódu skupiny vláken/procesů způsobí, že vlákna/procesy se musí zastavit na tomto místě, a nemohou pokračovat na tomto místě, dokud se ostatní vlákna k bariéře také nedostanou
- Použití – v případě, že potřebujeme výsledek z jiného vlákna abychom mohli pokračovat ve výpočtu
- V případě synchronního posílání zpráv - implicitní bariéra

Stavy úkolů (př. jazyka ADA)



Semafor

- 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

Monitor a zasílání zpráv

- 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 Javu to probereme důkladněji)
- **Zasílání zpráv** (je to synchronní způsob komunikace)
  - Při asynchronní komunikují přes vyrovnávací paměť
  - Vede k principu schůzky (rendezvous Ada)

Paralelní konstrukce v jazyce ADA

- 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;
```

- Primárním prostředkem komunikace úkolů je **schůzka (rendezvous)**

Randezvous


```
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
      accept 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 vstup

          …--pije pivo
          ČIŠNÍK.PLATIT;

          …
end HOST1;
```

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

Randezvous (2)

- 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
```

Randezvous (3)

- Konzument a producent jsou také úkoly a komunikují např.

  Producent: SCHRANKA.PUT(456);

  Konzument: SCHRANKA.GET(I);

- 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;
```

Randezvous (4)

- 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.

Paralelismus na úrovni příkazů jazyka - Occam

- Jazyk Occam je imperativní paralelní jazyk
- SEQ uvozuje sekvenční provádění

```
SEQ
   x := x + 1
      y := x * x
```

- PAR uvozuje paralelní provádění
- V konstrukcích lze kombinovat

```
WHILE next <> eof
   SEQ
       x := next
       PAR
              in  ?  Next
              out  !  x  *  x
```

Paralelismus na úrovni příkazů jazyka - Fortran

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 násobí dělí trojúhelníkovou matici pod hlavní diagonálou příslušným číslem na diagonále

```
REAL DIMENSION (1000, 1000) :: A
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
```

- 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 výsledek seriového 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 kod.
- Po volání fork může být proces ukončen voláním exit.
- Synchronizaci procesů provádí příkaz wait, kterým rodič pozastaví svoji činnost až do ukončení svých dětí.

Paralelismus na úrovni programů (2)


```
#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;
}
```