Paralelní programování

přednášky

Jan Outrata

únor–duben 2007

Úvod

Paralelní (konkurentní) programování = konstrukce programu obsahujícího 2 a více procesů (vláken) vykonávaných paralelně

- spolupráce procesů na splnění úkolu
- společný přístup procesů ke kritickým zdrojům
- **synchronizace** procesů
  1. čtení a zápis **sdílených proměnných**
  2. posílání a přijímání **zpráv**

Paralelní program = kombinace instancí standardních synchronizačních problémů

Hardware a aplikace

- historický vývoj od ad-hoc řešení ke stěžejním principům a technikám
- hardware
  1. 1 CPU – (rychlé) přepínání procesů, **multiprogramming**
  2. více CPU (jader, MIMD) – paralelní běh procesů, **multiprocessor**, **multicomputer**, **network system**
- aplikace
  1. operační systém – procesy správy HW zdrojů a uživatelských úloh
  2. uživ. úlohy – kdykoliv je potřeba paralelního vykonávání: okenní systémy, transakční zpracování (DBS), vědecké výpočty, ...
- paralelní vs. distribuovaný program

Synchronizace

- potřeba při jakékoliv komunikaci procesů
- 2 druhy
  1. **vzájemné vyloučení (mutual exclusion)** – **kritické sekce** za sebou
  2. **podmíněná (condition)** – čekání na splnění podmínky
- např. producent-konzument pomocí sdíleného bufferu
- **stav** programu = aktuální hodnoty všech proměnných, každý proces mění neznámou rychlostí vykonáváním **atomických akcí** (implementovaných HW)
- **historie** programu = (prolínající se) sekvence atom. akcí jednotlivých procesů, obrovský počet
- synchronizace = omezení počtu historií na žádoucí historie, omezení možností následující atom. akce
- kritická sekce = sekvence atom. akcí přistupujících ke sdíleným prostředkům chovající se atomicky

Vlastnosti programu

= tvrzení platná pro všechny možné historie

- lze vyjádřit ve smyslu
  1. **bezpečnosti (safety)** – nikdy špatný stav
  2. **živosti (liveness)** – někdy dobrý stav
- např. vzájemné vyloučení, absence **deadlocku** = vlastnosti bezpečnosti, vstup do kritické sekce = vlastnost živosti
- demonstrace splnění
  1. testování a debugování – kontrola jen některých (!) historií, v případě paralelních programů extrémně náročné
  2. operační analýza (případů) – průzkum všech historií, obrovský počet
  3. **aserce** – abstraktní analýza pomocí formulí pred. logiky charakterizujících množiny stavů (**programová logika**), uměrné počtu atom. akcí

Programová logika

Lineární hledání: pole $a[0:n]$, $n > 0$, hodnota x je v a, index i prvního výskytu x?

```
(setf i 0)
(loop while (not (= (aref a i) x)) do
      (incf i)
      )
```

Důkaz správnosti?

Programová logika je formální systém pro vývoj a analýzu programů pomocí asercí (podmínek)

- predikáty vyjadřující stav programu
- relace vyjadřující vykonávání programu

Výroková (Boole) a predikátová logika

- symboly, formule, axiomy, odvozovací pravidla $\frac{H_i}{C}$, důkaz, teorém, interpretace (vyhodnocení) symbolů a formulí, model, korektnost a úplnost (vzhledem k interpretaci)
- vlastnosti programů nemohou mít úplnou axiomatizaci (aritmetika + Gödelova věta o neúplnosti) $\rightarrow$ relativní úplnost
- výroky: true/false, proměnné, log. spojky ($\neg, \vee, \wedge, \Rightarrow, =$), splnitelný výrok, tautologie/axiomy (zákony negace, kontradikce, implikace, komutativity, asociativity, distributivity, De Morganovy, ...)
- predikáty: relace ($<, >, =, \neq$), kvantifikátory ($\exists, \forall$), vázané proměnné, volný a vázaný výskyt proměnné ve formuli, tautologie/axiomy (zákony De Morganovy pro kvantifikátory, konjunkce, disjunkce, prázdný rozsah), substituce P_e^x výrazu e za proměnnou x

Sekvenční program (Hoare 1969)

- symboly: predikáty (aserce/podmínky) $P, Q, \ldots$, výrazy programovacího jazyka S, závorky $\{\}$
- formule: **trojice** $\{P\}\ S\ \{Q\}$, $P = pre(S)$ je prekondice S, $Q = post(S)$ je postkondice S, S bez vedlejších efektů

Vyhodnocení trojice: Trojice $\{P\}\ S\ \{Q\}$, P je pravdivá, jestliže ze stavu splňujícího P po vykonání S splňuje výsledný stav Q.

- částečná korektnost (předpoklad dokončení S)
- např. $\{x = 0\}$ (incf x) $\{x = 1\}$

Axiom přiřazení: $\{P\}$ (setf x e) $\{P, ?\}$?

$$\{P_e^x\}\ \text{(setf x e)}\ \{P\}$$

Např. $\{y = 1\}$ (setf x 5) $\{y = 1 \wedge x = 5\}$.
$\{P_8^{a[3]}\}$ (setf (aref a 3) 8) $\{P : i = 3 \wedge a[i] = 8\}$?

Odvozovací pravidla

Pravidlo důsledku: zesílení prekondice, zeslabení postkondice

$$\frac{P' \Rightarrow P, \{P\}\ S\ \{Q\}, Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Např. $\frac{x{=}3 \Rightarrow \mathit{true}, \{\mathit{true}\}\ (\text{setf x 5})\ \{x{=}5\}}{\{x{=}3\}\ (\text{setf x 5})\ \{x{=}5\}}$.

Pravidlo složení:

$$\frac{\{P\}\ S_1\ \{Q\}, \{Q\}\ S_2\ \{R\}}{\{P\}\ (\text{progn } S_1\ S_2)\ \{R\}}$$

Pravidlo podmínky: (if B S)

$$\frac{P \wedge \neg B \Rightarrow Q, \{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{if}\ \{Q\}}$$

Odvozovací pravidla

Pravidlo cyklu: (loop while B do S)

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{loop}\ \{I \wedge \neg B\}}$$

I = invariant cyklu

```
(setf fact 1 i 0)
(loop while (not (= i n)) do
      (progn
        (incf i)
        (setf fact (* fact i))
        )
      )
```

I: fact $= i!$

Důkaz (částečné správnosti)

= sekvence axiomů nebo formulí plynoucích z předchozích formulí

Např. pro lineární hledání

- prekondice $P : n > 0 \wedge (\exists j : 0 \leq j < n : a[j] = x)$
- postkondice $LS : a[i] = x \wedge (\forall j : 0 \leq j < i : a[j] \neq x)$
- invariant cyklu $I : \forall j : 0 \leq j < i : a[j] \neq x$
- 11 formulí (kroků)

Nástin/idea důkazu = program s asercemi/podmínkami v poznámkách

- úplný - podmínky před a po každém výrazu
- v praxi podmínky jen na kritických místech

Důkaz správnosti lineárního hledání

; $P : n > 0 \wedge (\exists j : 0 \leq j < n : a[j] = x)$
```
(setf i 0)
```
; $i = 0, I : \forall j : 0 \leq j < i : a[j] \neq x$
```
(loop while (not (= (aref a i) x)) do
```
; $I \wedge a[i] \neq x$
```
      (incf i)
```
; I
```
      )
```
; $I \wedge a[i] = x$
; $LS : a[i] = x \wedge (\forall j : 0 \leq j < i : a[j] \neq x)$

Programový kalkul

- pomocí programové logiky důkaz (částečné) správnosti programu
- častěji zadán cíl (poskondice) a předpoklady (prekondice) a úkol zkonstruovat program

Programový kalkul (Dijkstra 1975) pro konstrukci (úplně) správných programů

- výrazy jako transformace predikátů
- vývoj programu spolu s konstrukcí důkazu

Paralelní program

- dva a více komunikujících sekvenčních programů, které je potřeba synchronizovat
- komunikace pomocí čtení a zápisu sdílených proměnných
- problém **interference (ovlivnění)**: jeden proces zneplatní předpoklady jiného procesu

Paralelní vykonávání procesů:

- vlákno vykonávání pro každý proces
- (**co-progn** S_1 S_2 ...) = prolnutí atom. akcí sekvenčních programů S_i
- globální a lokální proměnné
- (**co-dotimes** (i n) S) pro stejných n procesů S

Vzájemné vyloučení = kombinace (jemných, fine-grained) atom. akcí do složených (hrubých, coarse-grained) akcí

Příklady co-progn a co-dotimes

```
(setf x 0 y 0)
(co-progn (incf x) (incf y))
(setf z (+ x y))

(co-dotimes (i n)
  (setf (aref a i) 0)
)
```

Jemná atomicita

- **atomická akce** nedělitelná, dočasné stavy neviditelné ostatním procesům
- jemná atom. akce = implementována HW (instrukce) nebo OS
- každé přiřazení musí být atomické!
- výpočet = více instrukcí

```
(setf y 0 z 0)
(co-progn
  (setf x (+ y z))
  (progn (setf y 1) (setf z 2))
  )
```

x může být 0, 1, 2 i 3!

Charakteristiky stroje

- hodnoty základních typů (např. čísel) jsou čteny a zapisovány atomicky
- hodnoty se načítají z paměti do registrů, tam se s nimi pracuje a pak se uloží zpět do paměti
- každý proces má vlastní sadu registrů
- dočasné výsledky v privátních registrech nebo paměti (zásobník)

Pak, jesliže výraz nebo přiřazení neodkazuje proměnnou měněnou jiným prosesem, je atomický.

Charakteristiky stroje

Vlastnost „jednou a dost” (JD): Výraz *e* ji splňuje, jestliže se odkazuje nejvýše jedenkrát na nejvýše jednu jednoduchou proměnnou, která může být měněna jiným procesem. Přiřazení (setf x e) ji splňuje, jestliže ji splňuje výraz *e* a proměnnou *x* nečte jiný proces nebo *x* je jednoduchá proměnná a *e* neodkazuje žádnou proměnnou, která může být měněna jiným procesem.

Jesliže výraz nebo přiřazení splňuje vlastnost JD, je atomický.

Nesplňuje:

```
(co-progn
  (setf x (+ y 1))
  (setf y (+ x 1))
  )
```

Synchronizace

- i když výraz nebo přiřazení nesplňuje JD, chceme jej provést atomicky
- chceme několik výrazů nebo přiřazení provést atomicky (hrubé atom. akce), např. vkládání a mazání do/ze seznamu
- potřeba synchronizačních mechanizmů
- (**await** B S) - čeká na splnění podmínky B, pak atomicky provede S, se zárukou platnosti B!!
- await velice silné (jakákoliv hrubá atom. akce), velice obtížné implementovat
- speciální případy, např. (await ($>$ s 0) (decf s))
- vzájemné vyloučení: (await t S) = (**atomically** S)
- podmíněná synchronizace: (await B), pokud B splňuje JD, pak lze implementovat jako (loop while (not B))

Např. (await ($>$ count 0)) $\rightarrow$ (loop while (not ($>$ count 0))).

Pravidlo synchronizace

; $x = 0 \wedge y = 0$

```
(co-progn (incf x) (incf y))
```

; $x = 1 \wedge y = 1$

- pro důkaz potřeba odvozovací pravidlo pro co-progn
- každý proces = sekvenční + synchronizace

Pravidlo synchronizace: (await B S)

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \text{await}\ \{Q\}}$$

Interference

$; x = 0 \wedge x = 0$

```
(co-progn (atomically (incf x)) (atomically (incf x)))
```

$; x = 1 \wedge x = 1$

?? Výsledek má být 2!

Problém **interference (ovlivnění)** procesu s podmínkou v jiném procesu, tj. zneplatnění podmínky.

1. před provedením atom. akce musí platit prekondice a nesmí ji porušit jiná atom. akce
2. prekondice je **kritická podmínka**
3. kritické podmínky nesmí interferovat s atom. akcemi paralelně vykonávaných procesů, tj. atom. akce nesmí zneplatnit kritické podmínky

Vyloučení interference

Kritická podmínka: V nástinu/idei důkazu poslední postkondice a každá (nejslabší) prekondice, která není v await, popř. atomically.

Např. v $\{P\}$ (atomically S_1 $\{Q\}$ S_2) $\{R\}$ není Q kritickou podmínkou.

Přiřazovací akce = atom. akce obsahující jedno nebo více přiřazení

Vyloučení interference: Přiřazovací akce a neinterferuje s kritickou podmínkou C, jestliže platí:

$$NI(a, C) : \{C \wedge pre(a)\}\ a\ \{C\},$$

tj. C je invariantní vzhledem k provedení a.

Pravidlo paralelizace

- více procesů neinterferuje, jestliže neinterferuje žádná přiřazovací akce s žádnou kritickou podmínkou
- pak jsou důkazy pro jednotlivé procesy platné i při paralelním vykonávání

Pravidlo paralelizace: (co-progn S_1 S_2 ...)

$$\frac{\{P_i\}\ S_i\ \{Q_i\}\ \text{jsou bez interferencí}, 1 \leq i \leq n}{\{P_1 \wedge \ldots \wedge P_n\}\ \text{co}-\text{progn}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

Vyloučení interference

1. důkazy pro jednotlivé procesy + ukázání neinterference
2. počet historií je exponenciální s počtem vykonaných atom. akcí, ale počet interferencí je lineární s počtem přiřazovacích akcí
3. pro n procesů s a přiřazovacími akcemi a C kritickými podmínkami: $n(n-1)aC$ interferencí
4. mnoho interferencí je stejných, protože procesy jsou si podobné či symetrické
5. vyloučení interference = úpravy kritických podmínek a přiřazovacích akcí tak, že neinterferují

Různé proměnné

Jestliže množina proměnných, do kterých se v jednom procesu přiřazuje, a množina proměnných v podmínkách důkazu druhého procesu jsou různé, procesy neinterferují.

```
(co-progn (incf x) (incf y))
```

$\{x = 0\}$ (incf x) $\{x = 1\}$ a $\{y = 0\}$ (incf y) $\{y = 1\}$

1 přiřazení, 2 podmínky = 4 možné interference, např.
$NI((\text{incf y}), x = 0) : x = 0 \wedge y = 0$ (incf y) $x = 0$ platí, ostatní stejné.

Prvky pole a struktur jako samostatné proměnné.

Zeslabení podmínek

I když se množiny překrývají, lze interferenci vyloučit zeslabením podmínek zahrnutím efektů paralelizace, tj. výsledků jiných procesů.

```
(co-progn (atomically (incf x)) (atomically (incf x 2)))
```

$\{x = 0\}$ (incf x) $\{x = 1\}$ a $\{x = 0\}$ (incf x 2) $\{x = 2\}$

- každé přiřazení interferuje s oběma podmínkami a navíc $x = 1 \wedge x = 2$ nedává správný výsledek $x = 3$.
- pokud se provede první proces před druhým, pak $\{x = 0 \vee x = 1\}$ (incf x 2) $\{x = 2 \vee x = 3\}$
- pokud se provede druhý proces před prvním, pak $\{x = 0 \vee x = 2\}$ (incf x) $\{x = 1 \vee x = 3\}$
- neinterferují, $\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2) \Rightarrow x = 0\}$ a $\{(x = 2 \vee x = 3) \wedge (x = 1 \vee x = 3) \Rightarrow x = 3\}$

Globální invariant

- predikát I s globálními proměnnými je **globální invariant**, jestliže (1) je platný před spuštěním procesů a (2) je invariantní vzhledem ke každé přiřazovací akci a, tj. platí $\{I \wedge pre(a)\}\ a\ \{I\}$
- jestliže je každá kritická podmínka C tvaru $I \wedge L$, kde L je predikát nad lokálními proměnnými (nebo globálními, do kterých přiřazuje jen proces), jsou důkazy procesů bez interferencí

Producent-konzument: kopie pole $a[0 : n]$ do pole $b[0 : n]$ pomocí sdíleného bufferu.

Producent-konzument

```
; a = A
(setf p 0 c 0)
(co-progn
  (loop while (< p n) do
        (await (= p c))
        (setf buf (aref a p))
        (incf p)
        )
  (loop while (< c n) do
        (await (> p c))
        (setf (aref b c) buf)
        (incf c)
        )
  )
; b = A
```

Producent: důkaz

Globální invariant $I : c \leq p \leq c + 1 \wedge p > c \Rightarrow (buf = A[p-1]) \wedge a = A$

; $IP : I$

```
(loop while (< p n) do
```

; $I \wedge p < n$

```
      (await (= p c))
```

; $I \wedge p < n \wedge p = c$

```
      (setf buf (aref a p))
```

; $I \wedge p < n \wedge p = c \wedge buf = A[p]$

```
      (incf p)
```

; IP

```
      )
```

; $IP \wedge p = n$

Konzument: důkaz

Globální invariant $I : c \leq p \leq c + 1 \wedge p > c \Rightarrow (buf = A[p-1]) \wedge a = A$

$; IC : I \wedge b[0:c] = A[0:c]$

```
(loop while (< c n) do
```

$; I \wedge c < n$

```
      (await (> p c))
```

$; I \wedge c < n \wedge p > c$

```
      (setf (aref b c) buf)
```

$; I \wedge c < n \wedge p > c \wedge b[c] = A[p-1] = A[c]$

```
      (incf c)
```

$; IC$

```
      )
```

$; IC \wedge c = n$

Producent-konzument: interference

- všechny podmínky jsou ve tvaru $I \wedge L$, kromě těch obsahujících $p = c$ a $p > c$, protože obsahují nelokální proměnné
- např. $A_1 : I \wedge p < n \wedge p = c$ obsahuje c, kterou mění (incf c) s prekondicí $A_2 : I \wedge c < n \wedge p > c \wedge b[c] = A[c]$
- $NI((\text{incf c}), A_1) : A_1 \wedge A_2$ (incf c) A_1 platí, protože $A_1 \wedge A_2$ neplatí, protože $p = c \wedge p > c \Rightarrow \mathit{false}$
- např. $A_2' : I \wedge c < n \wedge p > c$ obsahuje p, kterou mění (incf p) s prekondicí $A_1' : I \wedge p < n \wedge p = c \wedge \mathit{buf} = A[p]$
- $NI((\text{incf p}), A_2') : A_2' \wedge A_1'$ (incf p) A_2' platí, protože $A_2' \wedge A_1'$ neplatí, protože $p > c \wedge p = c \Rightarrow \mathit{false}$

Výrazy await zajišťují střídání procesů v přístupu k bufferu.

Producent-konzument: aktivní čekání

- výrazy await splňují JD, mohou být tedy implementovány cyklem
- (await (= p c)) $\rightarrow$ (loop while (not (= p c)))
- (await (> p c)) $\rightarrow$ (loop while (not (> p c)))
- **aktivní čekání (busy-waiting)**

Synchronizace

- atomická akce je nedělitelná, proto stačí ověřovat interference celé akce, ne např. jednotlivá přiřazení nebo podmínky o vnitřním stavu
- vzájemné vyloučení a podmíněná synchronizace pro vyloučení interference

P1: $\ldots\{pre(a)\}a\ldots$, P2: $\ldots S_1\{C\}S_2\ldots$ a přiřazení a interferuje s kritickou podmínkou C

1. "skrytí" C před a: (atomically (progn S_1 S_2))
2. zesílení $pre(a)$: $NI(a, C)$ bude splněno, jestliže:
   a) C neplatí, když se má vykonat a a nemůže se tedy vykonat S_2
   b) vykonání a způsobí platnost C

   a nahradíme za (await (or (not C) B) a), kde B charakterizuje stavy, kdy vykonání a způsobí platnost C

Bankovní systém: převod z účtu na účet a kontrola zpronevěry.

Bankovní systém

```
(let* ((ucty (make-array '(10) :initial-contents
                         '(10 20 30 40 50 60 70 80 90 100)))
       (n (length ucty))
       (zpronevera nil))
  (labels ((prevod (x y castka)
             (atomically
               (decf (aref ucty x) castka)
               (incf (aref ucty y) castka)
               ))
           (kontrola (celkem)
             (let ((total 0))
               (dotimes (i n)
                 (incf total (aref ucty i))
                 )
               (if (not (= total celkem))
                   (setf zpronevera t)
                 ))))
    (co-progn (prevod 7 1 50) (kontrola 550))
    zpronevera
    ))
```

Bankovní systém: převod

```
(prevod (x y castka)
; ucty[x] = X ∧ ucty[y] = Y
  (atomically
    (decf (aref ucty x) castka)
    (incf (aref ucty y) castka)
    ))
; ucty[x] = X − CASTKA ∧ ucty[y] = Y + CASTKA
```

Bankovní systém: kontrola

```
(kontrola (celkem)
  (let ((total 0))
    ; total = ucty[0] + ... + ucty[i]
    (dotimes (i n)
      ; C1 : total = ucty[0] + ... + ucty[i] ∧ i < n
      (incf total (aref ucty i))
      ; C2 : total = ucty[0] + ... + ucty[i]
      )
    ; total = ucty[0] + ... + ucty[n − 1] ∧ i = n
    (if (not (= total celkem))
        (setf zpronevera t)
      ))))
```

Bankovní systém

Převod je sice atomický, ale interferuje s podmínkami C_1 a C_2 – pokud se provede, když je cyklus kontroly mezi účty x a y, částka se přičte dvakrát.

Řešení:

1. vzájemné vyloučení pro skrytí C_1 a C_2: atomický celý cyklus kontroly
2. podmíněná synchronizace pro zabránění převodu, pokud je cyklus kontroly mezi účty:

```
(await (or (and (<  x i) (<  y i))
           (and (>  x i) (>  y i)))
    převod)
```

Pomocné proměnné (Clint 1973)

- programová logika i s pravidlem paralelizace není (relativně) úplná
- někdy potřeba uvažovat (např.) pořadí vykonávání procesů = **skrytý stav**

```
(co-progn (atomically (incf x) (atomically (incf x))))
```

oba procesy: $\{x = 0\}$ (incf x) $\{x = 1\}$

- interference a $x = 1 \wedge x = 1 \neq x = 2 \rightarrow$ zeslabení podmínek: $\{x = 0 \vee x = 1\}$ (incf x) $\{x = 1 \vee x = 2\}$
- interference i špatný výsledek zůstávají
- explicitní rozlišení procesů

```
(setf x 0 t1 0 t2 0)
(co-progn
  (progn (atomically (incf x)) (setf t1 1))
  (progn (atomically (incf x)) (setf t2 1))
  )
```

Pomocné proměnné

Globální invariant $x = t1 + t2$? Neplatí po incf $\rightarrow$ skrýt do atomické akce:

```
; I : x = t1 + t2
(setf x 0 t1 0 t2 0)
; I ∧ t1 = 0 ∧ t2 = 0
(co-progn
  ; I ∧ t1 = 0
  (atomically (progn (incf x) (setf t1 1)))
  ; I ∧ t1 = 1
  ; I ∧ t2 = 0
  (atomically (progn (incf x) (setf t2 1)))
  ; I ∧ t2 = 1
  )
; I ∧ t1 = 1 ∧ t2 = 1
```

Každá podmínka tvaru $I \wedge L \rightarrow$ neinterferují.

Pomocné proměnné

- programy různé, ale přidané proměnné jsou jen **pomocné**

Omezení pomocné proměnné: Pomocná proměnná x je jen v přiřazovacích výrazech (setf x e).

- pomocné proměnné nemohou program ovlivnit, stavy jsou bez nich stejné

Pravidlo pomocných proměnných:

$$\frac{\{P\}\ S'\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

P, Q bez pomocných proměnných, S vznikne z S' odstraněním výrazů s pomocnými proměnnými.

Vlatnosti bezpečnosti a živosti (Lamport 1977)

- vlastnosti živosti jsou dány plánovací politikou plánovače procesů OS, který rozhoduje, která atomická akce bude následující
- důkaz vlastnosti bezpečnosti: predikát charakterizující špatný stav
- formální logika pro dokazování vlastností živosti závisí na plánovací politice

Dokazování vlatností bezpečnosti

- jestliže program nesplňuje vlastnost, nesplňuje ji nějaká historie
- spec. případ: nesplňuje ji nějaký jeden špatný stav
- např. částečná správnost, vzájemné vyloučení, absence deadlocku, NE např. "hodnota x je neklesající"
- vlastnost jako absence špatnéhoh stavu charakterizovaného predikátem BAD

Důkaz vlastnosti bezpečnosti: Jestliže $\{P\}\ S\ \{Q\}$ je důkaz, P je počáteční stav, pak S splňuje vlastnost bezpečnosti specifikovanou jako $\neg BAD$, jestliže pro každou kritickou podmínku C platí $C \Rightarrow \neg BAD$.

- stačí uvažovat globální invariant I

Dokazování vlatností bezpečnosti

Vzájemné vyloučení: podmínky čekání P a Q dvou procesů nemohou být současně pravdivé, tj. $BAD : P \wedge Q = false$.

Vyloučení konfigurací: Jeden proces nemůže být ve stavu splňujícím P zatímco jiný proces je ve stavu splňujícím Q, jestliže $P \wedge Q = false$.

Producent-konzument (kopírování pole): deadlock, jestliže oba čekají: $(PC \wedge p < n \wedge p \neq c) \wedge (IC \wedge c < n \wedge p \leq c) = false$.

Plánovací politiky a férovost (Lehman 1981)

- většina vlastností živosti závisí na **férovosti**, garanci toho, že proces dostane šanci běžet, nezávisle na ostatních
- uvažujme 1 CPU: paralelní vykonávání = "prolnuté" sériové
- plánovací politika se týká výkonnosti a využití HW, ale důležité jsou také globální atributy a jejich efekt na vlastnosti živosti procesů

Loop-Stop:

```
(setf continue t)
(co-progn (loop while continue) (setf continue nil))
```

Předp., že politika předá CPU až proces skončí nebo čeká. Pak Loop-Stop na 1 CPU neskončí, pokud je Loop naplánován jako první. Stop musí dostat šanci:

Nepodmíněná férovost: Plánovací politika je nepodmíněně férová, jestliže každá nepodmíněná atom. akce, která se může vykonat, se někdy vykoná.

Např. round-robin na 1 CPU, paralelní vykonávání na více CPU.

Plánovací politiky a férovost

Podmíněná atom. akce (await B S) se může provést, až platí podmínka B (která do provedení akce není zneplatněna):

Slabá férovost: Plánovací politika je slabě férová, jestliže je 1) nepodmíněně férová a 2) každá podmíněná atom. akce, která se může vykonat, se někdy vykoná, za předpokladu že podmínka nabyde platnosti a není zneplatněna (kromě procesem samotným).

Např. round-robin s časovými kvanty.

Slabá férovost nestačí, pokud se podmínka rychle mění:

Silná férovost: Plánovací politika je silně férová, jestliže je 1) nepodmíněně férová a 2) každá podmíněná atom. akce, která se může vykonat, se někdy vykoná, za předpokladu že podmínka platí nekonečněkrát často (v každé historii nekončícího programu).

Plánovací politiky a férovost

Silná plánovací politika musí zaručovat naplánování podmíněné atom. akce, když je podmínka platná:

Loop-Try-Stop:

```
(setf continue t try nil)
(co-progn
  (loop while continue do
        (progn (setf try t) (setf try nil))
        )
  (await try (setf continue nil))
  )
```

Se silně férovou politikou skončí, protože try je nekonečněkrát platné, se slabě férovou nemusí skončit.

Plánovací politiky a férovost

- nelze navrhnout obecnou praktickou a zároveň silně férovou politiku
- na 1 CPU je politika přepínající procesy po každé atom. akci silně férová, ale nepraktická, round-robin s časovými kvanty je praktická, ale není silně férová (ani na více CPU)
- speciální při čekání procesů: "první přijde, první mele" – naplánován je proces, který čeká nejdéle

Producent-konzument (kopírování pole): díky střídání (podmínka zůstává platná) stačí slabě férová politika, při aktivním čekání stačí nepodmíněně férová

Obecně nepodmíněně férová politika pro nepodmíněné atom. akce nemusí stačit, např. v Loop-Try-Stop s aktivním čekáním může plánovat Loop vždy, když continue platí = **livelock** – analogie deadlocku při aktivním čekání.

Sdílené proměnné

- řešení cyklů: invariant cyklu
- řešení synchronizace: zabránění interference pomocí await = **hrubá synchronizace**
- await těžké implementovat v obecné podobě → aktivní čekání s využitím nízkoúrovňových **synchronizačních primitiv** implementovaných HW nebo OS + techniky vyloučení interference = **jemná synchronizace**
- nejčastěji globální invariant (lze využít i pro vlastnosti bezpečnosti charakterizované predikátem *BAD*) a různé proměnné

Postup řešení (Andrews 1989):

1. Definice problému: procesy, synchronizační problémy, invariant
2. Návrh řešení: proměnné, atomické akce
3. Zajištění invariantu: podmíněné atom. akce
4. Implementace atom. akcí: sekvenční výrazy + dostupná synchronizační primitiva

Jemná synchronizace

- (téměř) všechny CPU obsahují instrukce, které atomicky testují (vrací) a mění sdílenou proměnnou a dají se tak použít pro implementaci await:
  - **Test-And-Set**: nastav proměnnou a vrať její původní hodnotu
  - **Fetch-And-Add**: inkrementuj proměnnou a vrať její původní hodnotu
  - **Swap**: prohoď obsah dvou proměnných
  - **Compare-And-Swap**: jestliže je hodnota proměnné rovna dané hodnotě, nastav obsah proměnné na jinou hodnotu, a vrať srovnání hodnot
  - ...
- každou instrukci lze (většinou) implementovat pomocí jiné, tj. pro implementaci await lze využít libovolnou z nich
- aktivní čekání je na 1 CPU je neefektivní, na více CPU efektivnější, samotný HW jej používá (např. přesuny dat v paměti nebo po síti)

Problém kritické sekce (Dijkstra 1965)

- klasický synchronizační problém, řešení se dají použít pro implementaci await
- více procesů opakovaně vykonává **kritickou sekci** kódu, ve které se přistupuje ke sdílenému prostředku, a nekritickou sekci kódu
- kritická sekce je ohraničená vstupním a výstupním protokolem
- předp., že proces, který vstoupí do kritické sekce, z ní i vystoupí, tj. může skončit jen mimo kritickou sekci

```
(co-dotimes (i n)
  (loop while t do
        (progn
          ; vstupní protokol
          ; kritická sekce
          ; výstupní protokol
          ; nekritická sekce
          )
        )
  )
```

Řešení kritické sekce

Úkol: navrhnout vstupní a výstupní protokol, které splňují:

1. vzájemné vyloučení: v kritické sekci je v daném čase nejvýše jeden proces
2. absence deadlocku: pokud se více procesů snaží zároveň vstoupit do kritické sekce, alespoň jeden uspěje
3. absence zbytečného čekání: pokud se proces snaží vstoupit do kritické sekce a žádný jiný proces v ní není, není mu bráněno
4. zaručený vstup: proces snažící se vstoupit do kritické sekce do ní někdy vstoupí

- vlastnosti bezpečnosti: 1., 2. při (pasivním, pomocí await) čekání, 3. (špatný stav = procesu je bráněno)
- vlastnosti živosti: 2. při aktivním čekání (procesy nečekají) = absence livelocku, 4.

Řešení kritické sekce

Triviální "řešení": celou kritickou sekci provést atomicky (atomically).

Pak stačí nepodmíněně férové plánování, ale

1. neefektivní – vylučuje paralelní vykonávání nekritických sekcí jiných procesů
2. implementace – CPU neposkytuje atomically!

Kritická sekce: hrubé řešení

2 procesy

Vzájemné vyloučení: potřeba indikovat, že proces je v kritické sekci $\rightarrow$ sdílené proměnné in1 a in2

Vlastnost bezpečnosti a globální invariant:
$MUTEX : \neg(in1 \wedge in2) = \neg in1 \vee \neg in2$

Kritická sekce: hrubé řešení – návrh

```
(setf in1 nil in2 nil)   ; MUTEX : ¬ (in1 ∧ in2)  =  ¬ in1 ∨ ¬ in2
  (labels ((P1 ()
             (loop while t do
                   (progn
                     (setf in1 t)          ⟵ PROBLÉM!
                     ; kritická sekce
                     (setf in1 nil)
                     ; nekritická sekce
                     )))
           (P2 ()
             (loop while t do
                   (progn
                     (setf in2 t)          ⟵ PROBLÉM!
                     ; kritická sekce
                     (setf in2 nil)
                     ; nekritická sekce
                     ))))
    (co-progn (P1) (P2))
    )
```

Kritická sekce: hrubé řešení

Zajištění invariantu:

- inicializace proměnných
- platnost po vstupním protokolu (setf in1 t):
  $\neg(true \wedge in2) = false \vee \neg in2 = \neg in2 \rightarrow$ zesílení vstupního protokolu:

  ```
  (await (not in2) (setf in1 t))
  ```

- platnost po výstupním protokolu (setf in1 nil):
  $\neg(false \wedge in2) = true \vee \neg in2 = true \rightarrow$ není potřeba čekat (a je atomické), obecně platné pro výstupní protokol

Kritická sekce: hrubé řešení

```
P1 ()
 (loop while t do
       (progn
         ; MUTEX ∧ ¬ in1
         (await (not in2) (setf in1 t))
         ; MUTEX ∧ in1
         ; kritická sekce
         (setf in1 nil)
         ; MUTEX ∧ ¬ in1
         ; nekritická sekce
         )))
```

P2 analogicky

Kritická sekce: hrubé řešení

Absence deadlocku: vyloučením konfigurací: procesy se snaží vstoupit do kritické sekce, ale nemohou, tzn. prekondice vstupu platí, ale podmínka vstupu neplatí, tzn. $(\neg in1 \wedge in2) \wedge (\neg in2 \wedge in1) = false$

Absence zbytečného čekání: vyloučením konfigurací: proces P1 se snaží vstoupit do kritické sekce, P2 je mimo kritickou sekci, ale P1 nemůže, tzn. postkondice výstupu P2 platí, ale podmínka vstupu P1 neplatí, tzn. $(\neg in2 \wedge in2) = false$

Zaručený vstup: jestliže se proces P1 snaží vstoupit do kritické sekce, ale nemůže, musí tam být proces P2, tzn. $in2 = true$, až P2 vystoupí, je $in2 = false$ a podmínka vstupu P1 platí, jestliže stále nemůže vstoupit, pak plánování není fér nebo vstoupil P2 a situace se opakuje, tzn. nekonečněkrát $in2 = false$ a platí podmínka vstupu P1 $\rightarrow$ potřeba silně férové plánování

Kritická sekce: hrubé řešení pomocí zámku

Více (N) procesů: N sdílených proměnných inX

Potřeba rozeznat jen 2 stavy:

1. nějaký proces je v kritické sekci
2. žádný proces není v kritické sekci

1 sdílená proměnná **lock** $= in1 \vee in2 \vee \ldots$, **zámek** kritické sekce

Kritická sekce: hrubé řešení pomocí zámku

```
(setf lock nil)
; pomocná proměnná in
; MUTEX : (P[i] je v kritické sekci) ⇒ (lock ∧ in[i] ∧
; (∀ j : 0 ≤ j < N, i ≠ j : ¬ in[j]))
(co-dotimes (i n)
  (let (in)
    (loop while t do
          (progn
            (await (not lock)
              (progn
                (setf lock t)
                (setf in t))
              )
            ; kritická sekce
            (setf lock nil)
            ; nekritická sekce
            ))))
```

Kritická sekce: jemné řešení – spin lock

Spin lock = cyklus, dokud sdílená proměnná lock neobsahuje hodnotu false

Pomocí instrukce Test-And-Set (TS):

```
(ts lock L)
```

=

```
(atomically (prog1 (setf L lock) (setf lock t)))
```

L je lokální proměnná

TS bývá implementována tak, že vrací *L*:
`(ts lock)` = `(atomically (prog1 lock (setf lock t)))`,
což ale při použití v aktivním čekání vytváří vedlejší efekt při vyhodnocování podmínky.

Kritická sekce: jemné řešení – spin lock

```
(setf lock nil)
; MUTEX : (P[i] je v kritické sekci) ⇒ (lock ∧ ¬ Li ∧
; (∀ j : 0 ≤ j < N, i ≠ j : Lj))
(co-dotimes (i n)
  (let (L)
    (loop while t do
          (progn
            (ts lock L)
            (loop while L do (ts lock L))
            ; kritická sekce
            (setf lock nil)
            ; nekritická sekce
            ))))
```

Kritická sekce: jemné řešení – spin lock

Vzájemné vyloučení: jestliže chce více procesů vstoupit do kritické sekce, jen jeden může "vidět" $lock = false$ (v atomické akci) a vstoupit, tj. nastavit $lock = true \wedge L = false$

Absence deadlocku/livelocku: před vstupem procesů do kritické sekce je $lock = false$ a jeden z nich může vstoupit

Absence zbytečného čekání: jestliže jsou všechny procesy mimo kritickou sekci, je $lock = false$ a procesu nic nebrání vstoupit

Zaručený vstup: nekonečněkrát $lock = false \rightarrow$ potřeba silně férové plánování

Kritická sekce: jemné řešení – spin lock

Je pravidlem, že výstupní protokol kritické sekce řešené aktivním čekáním vždy vrací sdílené proměnné do stavu před vstupním protokolem.

Na více CPU systémech instrukce TS zbytečně zatěžuje paměť: každý proces v cyklu aktivního čekání zapisuje do sdílené proměnné lock i když se nemění, CPU mají cache, která se tímto zneplatňuje a proměnná se musí znovu číst

Řešení: Zámek (sdílenou proměnnou lock) v cyklu jen opakovaně číst a až se změní, provést TTS = Test-And-Test-And-Set:

```
(tts lock L) = (progn (loop while lock) (ts lock L))
```

Toto jen snižuje zatížení paměti: při změně zámku mohou všechny procesy provést TS, ale jen jeden vstoupí do kritické sekce, ostatní opět "čekají".

Kritická sekce: implementace await (a atomically)

Libovolné řešení kritické sekce lze použít pro implementaci (`atomically` S):

```
CSenter S CSexit
```

CSenter a CSexit jsou vstupní a výstupní protokoly kritické sekce.

Předpoklad! Všechny (kritické) sekce všech procesů, ve kterých se přistupuje ke sdíleným proměnným v S, jsou chráněny stejnými protokoly.

Implementace (`await` B S)? B musí být zaručeně platná před vykonáním S:

```
CSenter
(loop while (not B) do ? )
S
CSexit
```

Předpoklad! Předchozí předpoklad platí pro B i S.

Kritická sekce: implementace await

Tělo "čekacího" cyklu?

- pro ukončení cyklu musí $B(= \mathit{false})$ nabýt platnosti, což může udělat jen jiný proces, ale k B se přistupuje jen v kritické sekci $\longrightarrow$ vystoupit z kritické sekce
- vyhodnocení B a vykonání S musí být atomické $\longrightarrow$ znovu vstoupit do kritické sekce

```
CSenter
(loop while (not B) do (progn CSexit CSenter))
S
CSexit
```

Pokud po nabytí platnosti B toto platí, stačí slabě férové plánování, jinak, jestliže se platnost B mění (a B platí nekonečněkrát), je potřeba silně férové plánování.

Kritická sekce: implementace await

- neefektivní (i když pomineme aktivní čekání): v cyklu se neustále vystupuje z a vstupuje do kritické sekce dokud B nenabyde platnosti $\longrightarrow$ počkat před znovuvstoupením a dát tak jiným procesům šanci změnit B:
  `CSexit Delay CSenter`
- lze použít i v aktivním čekání samotného CSenter
- S prázdné a B splňuje podmínku JD $\longrightarrow$
  `(loop while (not B))`
- použití např. v Ethernetu: rozhraní (síťová karta) pošle paket a detekuje případnou kolizi, čeká a pošle paket znovu; pro zabránění dalším kolizím (při čekání zhruba stejnou dobu) se doba mezi vysláním paketu volí náhodně z intervalu, který se po každé iteraci zdvojnásobuje = "kompromis binární exploze"

Kritické sekce: Algoritmus Tie-Breaker (Prolomení remízy) (Peterson 1981)

- řešení kritické sekce pomocí instrukce TS vyžaduje silně férové plánování pro zaručení vstupu, jinak se proces nemusí dočkat
- to je silný požadavek, důvod: nebere se v úvahu pořadí, ve kterém se více procesů snaží vstoupit

Petersonův algoritmus Tie-Breaker:

- nepotřebuje žádné speciální HW instrukce jako TS
- stačí mu nepodmíněně férové plánování pro zaručení vstupu procesu do kritické sekce

Tie-Breaker: hrubé řešení

- jak implementovat `(await (not in2) (setf in1 t))` z předchozího řešení?

  ```
  (loop while in2)
  (setf in1 t)
  ```

- *PROBLÉM*: aktivní čekání a akce nejsou atomické, není zaručena neplatnost *in*2 před vykonáním `(setf in1 t)`, tj. není zaručeno vzájemné vyloučení: postkondice cyklu P1 ($\neg in2$) interferuje s akcí P2 `((setf in2 t))`
- každý proces vyžaduje aby po ukončení čekacího cyklu nebyl žádný jiný proces v kritické sekci:

  ```
  (setf in1 t)
  (loop while in2)
  ```

- PROBLÉM?

Tie-Breaker: hrubé řešení

- *PROBLÉM*: deadlock: oba procesy se pokusí vstoupit ve stejnou dobu, $in1 \wedge in2 = true$ před čekacími cykly, které nikdy neskončí
- další jednoduchá proměnná umožňující vpuštění jednoho procesu a bránění vstupu druhého (prolomení remízy)
- *last*, indikující, který proces začal vykonávat vstupní protokol jako poslední a bude (dál) čekat

Tie-Breaker: hrubé řešení

```
(setf in1 nil in2 nil last 1)
; pomocné proměnné mid1, mid2
; TIE − BREAKER : ¬ (in1 ∧ ¬ mid1 ∧ in2 ∧ ¬ mid2) =
; ¬ in1 ∨ mid1 ∨ ¬ in2 ∨ mid2
(labels ((P1 ()
           (loop while t do
                 (progn
                   (atomically (progn (setf in1 t) (setf mid1 t)))
                   (atomically (progn (setf last 1) (setf mid1 nil)))
                   (await (or (not in2) (not (= last 1))))
                   ; kritická sekce
                   (setf in1 nil)
                   ; nekritická sekce
                   )))
         (P2 ()
           (loop while t do
                 (progn
                   (setf in2 t last 2)
                   (await (or (not in1) (not (= last 2))))
                   ; kritická sekce
                   (setf in2 nil)
                   ; nekritická sekce
                   ))))
  (co-progn (P1) (P2))
  )
```

Tie-Breaker: hrubé řešení

Vzájemné vyloučení: vyloučením konfigurací: když je P1 v kritické sekci, platí $in1 \wedge \neg mid1 \wedge (\neg in2 \vee last = 2 \vee mid2)$, podobně pro P2, spojení neplatí

Absence deadlocku: vyloučením konfigurací: procesy se snaží vstoupit do kritické sekce, ale nemohou, tzn. prekondice čekání platí, ale podmínka vstupu neplatí, tzn. $(in1 \wedge last = 1 \wedge in2) \wedge (in2 \wedge last = 2 \wedge in1) = false$

Absence zbytečného čekání: vyloučením konfigurací: proces P1 se snaží vstoupit do kritické sekce, P2 je mimo kritickou sekci, ale P1 nemůže, tzn. postkondice výstupu P2 platí, ale podmínka vstupu P1 neplatí, tzn. $(\neg in2 \wedge in2 \wedge last = 1) = false$

Zaručený vstup: jestliže se proces P1 snaží vstoupit do kritické sekce, ale nemůže, musí tam být proces P2, tzn. $in1 = true \wedge last = 1 \wedge in2 = true$, až P2 vystoupí, je $in2 = false$ a podmínka vstupu P1 platí, jestliže stále nemůže vstoupit, pak plánování není fér, protože i kdyby se P2 snažil vstoupit, nemohl, protože $in1 = true \wedge last = 2 \rightarrow$ stačí nepodmíněně férové plánování

Tie-Breaker: jemné řešení

- await nesplňuje JD (používá 2 sdílené proměnné měněné jiným procesem), teoreticky nelze implementovat aktivním čekáním
- v tomto případě ale atomicita await není potřeba:
  1. pokud podmínka await v P1 platí (a P1 může vstoupit do kritické sekce), jedniná možnost jejího zneplatnění je `(setf in2 t)` ve vstupním protokolu P2, tj. když P2 není v kritické sekci
  2. následně, vykonáním `(setf last 2)`, podmínka opět začne platit, ale P2 stále nemůže vstoupit do kritické sekce (protože $in1 = true$)
  3. během celého neatomického vstupního protokolu P1 nemůže být P2 v kritické sekci

Tie-Breaker: jemné řešení

```
(setf in1 nil in2 nil last 1)
(labels ((P1 ()
           (loop while t do
                 (progn
                   (setf in1 t last 1)
                   (loop while (and in2 (= last 1)))
                   ; kritická sekce
                   (setf in1 nil)
                   ; nekritická sekce
                   )))
         (P2 ()
           (loop while t do
                 (progn
                   (setf in2 t last 2)
                   (loop while (and in1 (= last 2)))
                   ; kritická sekce
                   (setf in2 nil)
                   ; nekritická sekce
                   ))))
  (co-progn (P1) (P2))
  )
```

Tie-Breaker: N procesů

Vstupní protokol:

- $N-1$ úrovní, proces postupuje po úrovních nahoru
- na každé úrovni algoritmus Tie-Breaker pro 2 procesy rozhodující, který proces postoupí do další úrovně
- proces čeká, pokud (1) je jiný proces na vyšší nebo stejné úrovni a (2) je poslední proces, který postoupil na tuto úroveň
- za první úrovní je nejvýše $N-1$ procesů, ..., za všemi $N-1$ úrovněmi je nejvýše 1 proces, tj. nejvýše 1 proces je v v daném čase kritické sekci

Výstupní protokol:

- proces "spadne" na nejnižší úroveň

Z výše popsaného protokolu plyne vzájemné vyloučení; absence livelocku, zbytečného čekání a postačující nepodmíněně férové plánování plynou z verze pro 2 procesy, z podmínek čekáná procesu a toho, že proces někdy ukončí kritickou sekci.

Tie-Breaker: N procesů

```
(setf in (make-array n :initial-element -1)
      last (make-array n :initial-element -1))
(co-dotimes (i n)
  (loop while t do
        (progn
          (loop for j from 0 to (- n 2) do
                (setf (aref in i) j (aref last j) i)
                (loop for k from 0 to (- n 1)
                      when (not (= i k)) do
                      (loop while (and (>= (aref in k)
                                           (aref in i))
                                       (= (aref last j) i)))))
          ; kritická sekce
          (setf (aref in i) -1)
          ; nekritická sekce
          )))
```

Kritické sekce: Algoritmus Ticket (Tahání lístků)

- algoritmus Tie-Breaker pro N procesů je relativně složitý (zobecnění z $N = 2$ není přímočaré)
- jednodušší řešení: čítač (**ticket**, lístek) pro uspořádání procesů a čekání ve frontě, dokud proces nebude na se svým lístkem na řadě

Hrubé řešení

- vytažený lístek procesu má číslo (o 1) větší než číslo lístku předchozího procesu, který začal vstupní protokol
- proces čeká, až všechny předchozí procesy ukončí kritické sekce, a pak (jako jediný) vstoupí do kritické sekce
- zajištění unikátnosti čísel lístků a nejvýše 1 procesu v kritické sekci: vytažení lístku a zvýšení čísla dalšího lístku a dalšího procesu na řadě musí být atomické

Ticket: hrubé řešení

```
(setf number 1 next 1 turn (make-array n :initial-element 0))
; TICKET : (P[i] je v kritické sekci) ⇒ (turn[i] = next) ∧
; (∀ i,j : 0 ≤ i,j < N, i ≠ j :
;          turn[i] = 0 ∨ turn[i] ≠ turn[j])
(co-dotimes (i n)
  (loop while t do
        (progn
          (atomically
            (progn
              (setf (aref turn i) number)
              (incf number)))
          (await (= (aref turn i) next))
          ; kritická sekce
          (atomically (incf next))
          ; nekritická sekce
          )))
```

Ticket: hrubé řešení

Vzájemné vyloučení: globální invariant
$TICKET : (P[i] \text{ je v kritické sekci}) \Rightarrow (turn[i] = next) \wedge (\forall i, j : 0 \leq i, j < N, i \neq j : turn[i] = 0 \vee turn[i] \neq turn[j])$

Absence deadlocku a zbytečného čekání: plyne z unikátnosti nenulových lístků

Zaručený vstup: stačí slabě férové plánování, protože podmínka vstupu zůstává po nabytí platnosti platná (další na řadě se nemění před vstupem procesu do kritické sekce)

Potenciální nedostatek (společný všem algoritmům používajícím inkrementální čítač): čísla lístků a další na řadě jsou neomezená $\rightarrow$ modulo hodnota $\geq N$ (pro zajištění unikátnosti čísel)

Ticket: jemné řešení

- await používá jedinkrát jedinou sdílenou proměnnou next, splňuje tedy JD a lze jej implementovat aktivním čekáním
- výstupní protokol může být neatomický, protože jej (celý) může provést nejvýše 1 proces (až poslední atom. akce vpustí jiný proces do kritické sekce)
- vytažení lístku a inkrementace čísla, tj. přečtení a inkrementace proměnné: potřeba HW atomické instrukce Fetch-And-Add (FA):

```
(fa var addend)
```

=

```
(atomically (prog1 var (incf var addend)))
```

+ modulo N

Ticket: jemné řešení

```
(setf number 1 next 1 turn (make-array n :initial-element 0))
; TICKET : (P[i] je v kritické sekci) ⇒ (turn[i] = next) ∧
; (∀ i,j : 0 ≤ i,j < N, i ≠ j :
;          turn[i] = 0 ∨ turn[i] ≠ turn[j])
(co-dotimes (i n)
  (loop while t do
        (progn
          (setf (aref turn i) (fa number 1))
          (loop while (not (= (aref turn i) next)))
          ; kritická sekce
          (incf next)
          ; nekritická sekce
          )))
```

Ticket: jemné řešení

Pokud instrukce FA (nebo srovnatelná, ale nestačí jen atomická inkrementace!) není k dispozici, lze použít jiné řešení kritické sekce:

```
CSenter
(setf (aref turn i) number)
(incf number)
CSexit
```

Nemusí být zaručeno pořadí čísel lístů v jakém procesy začaly provádět vstupní protokol (např. při použití řešení s instrukcí TS) a teoreticky může proces čekat donekonečna (jiný proces pořád "předbíhá") – nepravděpodobné.

Kritické sekce: Algoritmus Bakery (Pekařský) (Lamport 1974)

- algoritmus Ticket potřebuje instrukci FA nebo jiné řešení kritické sekce, při kterém ale nemusí být spravedlivý
- podobný **bakery (pekařský)** je spravedlivý a nepotřebuje žádnou speciální HW instrukci pro zajištění vzájemného vyloučení, poskytuje prolomení remízy při vytažení lístků se stejným číslem

Hrubé řešení

- vstupní protokol: nastavení čísla lístku na větší než čísla lístků všech ostatních procesů
- proces čeká, až bude číslo jeho lístku menší než čísla všech lístků všech ostatních procesů
- stejně jako u algoritmu Ticket pro zajištění unikátnosti čísel lístků a nejvýše 1 procesu v kritické sekci musí být vytažení lístku atomické
- na rozdíl od algoritmu Ticket si procesy porovnávají čísla lístků mezi sebou, atomicky

Bakery: hrubé řešení

```
(setf turn (make-array n :initial-element 0))
; BAKERY : (P[i] je v kritické sekci) ⇒ (turn[i] ≠ 0 ∧
; (∀ j : 0 ≤ j < N, i ≠ j : turn[j] = 0 ∨ turn[i] < turn[j]))
(co-dotimes (i n)
  (loop while t do
        (progn
          (atomically
            (setf (aref turn i) (+ (max-array turn) 1)))
          (loop for j from 0 to (- n 1)
                when (not (= j i)) do
                (await (or (= (aref turn j) 0)
                           (< (aref turn i) (aref turn j)))))
          ; kritická sekce
          (setf (aref turn i) 0)
          ; nekritická sekce
          )))
```

Bakery: hrubé řešení

Vzájemné vyloučení: globální invariant
$BAKERY : (P[i] \text{ je v kritické sekci}) \Rightarrow (turn[i] \neq 0 \wedge (\forall j : 0 \leq j < N, i \neq j : turn[j] = 0 \vee turn[i] < turn[j]))$

Absence deadlocku: plyne z unikátnosti nenulových lístků

Absence zbytečného čekání: plyne z toho, že $turn[i] = 0$ platí jen mimo kritickou sekci

Zaručený vstup: stačí slabě férové plánování, protože podmínka vstupu zůstává po nabytí platnosti platná

Potenciální nedostatek: čísla lístků mohou příliš narůstat (a nelze to řešit modulem), ale to znamená, že vždy alespoň 1 proces čeká příliš dlouho na vstup do kritické sekce – nepravděpodobné

Bakery: jemné řešení – návrh

```
(setf turn1 0 turn2 0)
(labels ((P1 ()
           (loop while t do
                 (progn
                   (setf turn1 (+ turn2 1))     ⟵ PROBLÉM!
                   (loop while (and (not (= turn2 0)) (> turn1 turn2)))
                   ; kritická sekce
                   (setf turn1 0)
                   ; nekritická sekce
                   )))
         (P2 ()
           (loop while t do
                 (progn
                   (setf turn2 (+ turn1 1))     ⟵ PROBLÉM!
                   (loop while (and (not (= turn1 0)) (> turn2 turn1)))
                   ; kritická sekce
                   (setf turn2 0)
                   ; nekritická sekce
                   ))))
  (co-progn (P1) (P2))
  )
```

Bakery: jemné řešení

- vstupní protokol: ani přiřazení ani aktivní čekání nesplňují JD (a neatomické) – vzájemné vyloučení není zaručeno (oba procesy provedou zároveň přiřazení a $turn1 = 1 \wedge turn2 = 1$)
- řešení: jeden proces bude čekat (podobně jako v algoritmu Tie-Breaker) $\rightarrow$ zesílení čekací podmínky P2: `(>= turn2 turn1)`
- race condition (chyba souběhu): P1 přečte $turn2 = 0$, P2 vstoupí, P1 vstoupí (má přednost), oba v kritické sekci, protože P2 předběhl P1
- řešení: na začátku vstupních protokolů: `(setf turn1 1)` a `(setf turn2 1)`
- neatomické čekání: proces čeká, až druhý proces dokončí nastavení proměnné turn

Bakery: jemné řešení

```
(setf turn1 0 turn2 0)
(labels ((P1 ()
           (loop while t do
                 (progn
                   (setf turn1 1 turn1 (+ turn2 1))
                   (loop while (and (not (= turn2 0)) (> turn1 turn2)))
                   ; kritická sekce
                   (setf turn1 0)
                   ; nekritická sekce
                   )))
         (P2 ()
           (loop while t do
                 (progn
                   (setf turn2 1 turn2 (+ turn1 1))
                   (loop while (and (not (= turn1 0)) (>= turn2 turn1)))
                   ; kritická sekce
                   (setf turn2 0)
                   ; nekritická sekce
                   ))))
  (co-progn (P1) (P2))
  )
```

Bakery: jemné řešení

Vzájemné vyloučení: vyloučením konfigurací: když je P1 v kritické sekci, platí $turn1 > 0 \wedge (turn2 = 0 \vee turn1 \leq turn2)$, podobně pro P2 platí $turn2 > 0 \wedge (turn1 = 0 \vee turn2 < turn1)$, spojení neplatí

Absence deadlocku: vyloučením konfigurací: procesy se snaží vstoupit do kritické sekce, ale nemohou, tzn. prekondice čekání platí, ale podmínka vstupu neplatí, tzn. $(turn1 > 0 \wedge (turn2 \neq 0 \wedge turn1 > turn2)) \wedge (turn2 > 0 \wedge (turn1 \neq 0 \wedge turn2 \geq turn1)) = false$

Absence zbytečného čekání: vyloučením konfigurací: proces P1 se snaží vstoupit do kritické sekce, P2 je mimo kritickou sekci, ale P1 nemůže, tzn. postkondice výstupu P2 platí, ale podmínka vstupu P1 neplatí, tzn. $turn2 = 0 \wedge (turn2 \neq 0 \wedge turn1 > turn2) = false$

Zaručený vstup: jestliže se proces P1 snaží vstoupit do kritické sekce, ale nemůže, musí tam být proces P2, tzn. $turn2 > 0 \wedge turn1 > turn2$, až P2 vystoupí, je $turn2 = 0$ a podmínka vstupu P1 platí, jestliže stále nemůže vstoupit, pak plánování není slabě férové, protože i kdyby se P2 snažil vstoupit, nemohl, protože $turn1 > 0 \wedge turn2 \geq turn1$, a podmínka vstupu P1 stále platí $\rightarrow$ stačí slabě férové plánování

Bakery: N procesů

Řešení se dvěma procesy není zcela symetrické (podmínky). Symetrie:

$$\begin{array}{lcll} (a,b) > (c,d) & = & \mathit{true} & \text{jestliže } a > c \text{ nebo jestliže } a = c \text{ a } b > d \\ & = & \mathit{false} & \text{jinak} \end{array}$$

Podmínky:
P1: `(>-list (list turn1 1) (list turn2 2))`
P2: `(>-list (list turn2 2) (list turn1 1))`
P[i]: `(>-list (list (aref turn i) i) (list (aref turn j) j))`

- proces čeká, dokud nemá přednost před všemi ostatními
- globální invariant *BAKERY* je téměř stejný, jen s tím rozdílem, že procesy mohou mít stejné číslo lístku, v tom případě má přednost ten s menším číslem

Bakery: N procesů

```
(setf turn (make-array n :initial-element 0))
; BAKERY : (P[i] je v kritické sekci) ⇒ (turn[i] ≠ 0 ∧
; (∀ j : 0 ≤ j < N, i ≠ j : turn[j] = 0 ∨ turn[i] < turn[j]))
(co-dotimes (i n)
  (loop while t do
        (progn
          (setf (aref turn i) 1)
          (setf (aref turn i) (+ (max-array turn) 1)))
          (loop for j from 0 to (- n 1)
                when (not (= j i)) do
                (loop while (and
                             (not (= (aref turn j) 0))
                             (>-list (list (aref turn i) i)
                                     (list (aref turn j) j)))))
          ; kritická sekce
          (setf (aref turn i) 0)
          ; nekritická sekce
          )))
```

Bariéry (Jordan 1978)

- mnoho problémů je iterativního rázu, tj. opakovaný výpočet lepších a lepších aproximací řešení a ukončení při dosažení výsledku nebo konvergenci k výsledku (typicky numerické úlohy)
- části dat lze zpracovávat paralelně více procesy (vykonávajícími stejný algoritmus)
- většinou iterace závisí na výsledcích předchozích iterací – **klíčové**

```
(loop while t do
      (co-dotimes (i n) (implementace procesu i)))
```

neefektivní: vytváření nových procesů při každé iteraci

Bariéry

```
(co-dotimes (i n)
  (loop while t do
        (progn
          (implementace procesu i)
          (čekání na dokončení všech n procesů))))
```

čekání = synchronizace **bariérou**: bod čekání (typicky na konci každé iterace), do kterého musí všechny procesy dojít, než budou vpuštěny dál

Bariéry: sdílený čítač

- nejjednodušší: čítač, inicializován na 0, proces jej před bariérou inkrementuje a čeká, až bude roven počtu procesů (= k bariéře došly všechny procesy)
- globální invariant: $COUNT : \forall i : 0 \leq i < N : passed[i] \Rightarrow count = N$

```
(setf count 0 passed (make-array n :initial-element nil))
(co-dotimes (i n)
  (loop while t do
        (progn                                    ← PROBLÉM!
          ; implementace procesu i
          (incf count)
          (await (= count n) (setf (aref passed i) t)))))
```

Bariéry: sdílený čítač

- await pomocí aktivního čekání, inkrementace pomocí Fetch-And-Add
- *PROBLÉM*: čítač je potřeba resetovat na 0 po každém průchodu bariérou a musí být resetován před inkrementováním lib. procesem
- možné řešení: 2 čítače, jeden inkrementovat od 0 do N, druhý dekrementovat od N do 0, za bariérou prohodit
- praktické problémy:
  1. inkrementace (a dekrementace) atomicky (FA nemusí být dostupná)
  2. čekající procesy v cyklu testují čítač $\rightarrow$ zatěžování paměti (u N CPU kvůli cache)

Bariéry: vlajky a koordinátor

- řešení problémů sdíleného čítače: rozdistribuovat čítač mezi procesy
- procesy Pracovník[i]: místo inkrementace čítače nastavení proměnné, čekání na jinou proměnnou:

```
(setf (aref arrive i) 1)
(await (= (aref continue i) 1))
```

- proces Koordinátor: čeká na nastavení prvních proměnných a nastaví druhé proměnné všech procesů:

```
(dotimes (i n) (await (= (aref arrive i) 1)))
(dotimes (i n) (setf (aref continue i) 1))
```

- await pomocí aktivního čekání (1 sdílená proměnná)
- interpretace proměnných arrive a continue:
  $(\forall i : 0 \leq i < N : (arrive[i] = 1) \Rightarrow$
  Pracovník[i] dosáhl bariéry$) \wedge (\forall i : 0 \leq i < N : (continue[i] = 1) \Rightarrow$
  Pracovník[i] může skrz bariéru$)$

Bariéry: vlajky a koordinátor

Proměnné arrive a continue = **vlajky** – nastaveny pro signalizaci platnosti synchronizační podmínky

Zbývá: resetování vlajek před další iterací procesu

Principy synchronizace vlajkou: (1) Proces, který čeká na nastavení vlajky, je ten, který ji také resetuje.
(2) Vlajka není nastavena, dokud není resetována.

(1) zajišťuje, že vlajka není resetována dřív než je nastavena:
Pracovník[i] resetuje continue[i], Koordinátor resetuje všechny arrive[i]

(2) zajišťuje, že jiný proces nenastaví vlajku dřív než je resetována, což by mohlo způsobit deadlock (první proces potom marně čeká na nastavení):
Koordinátor resetuje všechny arrive[i] před nastavením všech continue[i]

Bariéry: vlajky a koordinátor

```
(setf arrive (make-array n :initial-element 0)
      continue (make-array n :initial-element 0))
(labels ((Pracovnik (i)
           (loop while t do
                 ; implementace procesu i
                 (setf (aref arrive i) 1)
                 (await (= (aref continue i) 1))
                 (setf (aref continue i) 0)))
         (Koordinator ()
           (loop while t do
                 (dotimes (i n)
                   (progn
                     (await (= (aref arrive i) 1))
                     (setf (aref arrive i) 0)))
                 (dotimes (i n)
                   (setf (aref continue i) 1)))))
  (co-progn
    (co-dotimes (i n) (Pracovnik i))
    (Koordinator)))
```

Bariéry: stromově kombinovaná bariéra

Nepříjemné vlastnosti bariéry pomocí vlajek a koordinátora:

- extra proces (Koordinátor), nejlépe na samostatném CPU kvůli aktivnímu čekání
- čas každé iterace Koordinátora (a tedy bariéry) je lineárně závislý na počtu procesů: u iterativních algoritmů se dá očekávat, že procesy dorazí k bariéře zhruba ve stejnou dobu, tzn. že Koordinátor většinu času aktivně čeká

Řešení:

- kombinace (pracovního) procesu a Koordinátora: proces je také koordinátorem
- procesy organizované do stromu: proces čeká na signál arrive zdola od potomků a posílá jej nahoru rodiči a až dostane signál continue shora od rodiče, pošle jej dolů potomkům
- **stromově kombinovaná bariéra**: proces kombinuje výsledky potomků a posílá je rodiči, logaritmicky závislá na počtu procesů
- vylepšení: 1 signál (proměnná) continue, resetování: 1. 2 proměnné a střídání, 2. střídání hodnoty, na kterou se čeká (0,1)

Bariéry: stromově kombinovaná bariéra

listový proces *l*:

```
(setf (aref arrive l) 1)
(await (= (aref continue l) 1)) (setf (aref continue l) 0)
```

vnitřní proces *i*:

```
(await (= (aref arrive left) 1)) (setf (aref arrive left) 0)
(await (= (aref arrive right) 1)) (setf (aref arrive right) 0)
(setf (aref arrive i) 1)
(await (= (aref continue i) 1)) (setf (aref continue i) 0)
(setf (aref continue left) 1 (aref continue right) 1)
```

kořenový proces *r*:

```
(await (= (aref arrive left) 1)) (setf (aref arrive left) 0)
(await (= (aref arrive right) 1)) (setf (aref arrive right) 0)
(setf (aref continue left) 1 (aref continue right) 1)
```

Bariéry: symetrické bariéry

- ve stromově kombinované bariéře mají procesy různé role (kořenový, vnitřní, listový), vnitřní procesy vykonávají nejvíc práce, kořenový čeká na všechny
- pokud by každý proces běžel na svém CPU, měly by k bariéře dorazit všechny ve stejnou dobu a paralelně projít
- **symetrická bariéra**:
  - tvořena z párů bariér pro 2 procesy
  - technika pracovník/koordinátor, ale každý proces nastaví svoji vlajku a čeká na vlajku druhého, kterou pak resetuje

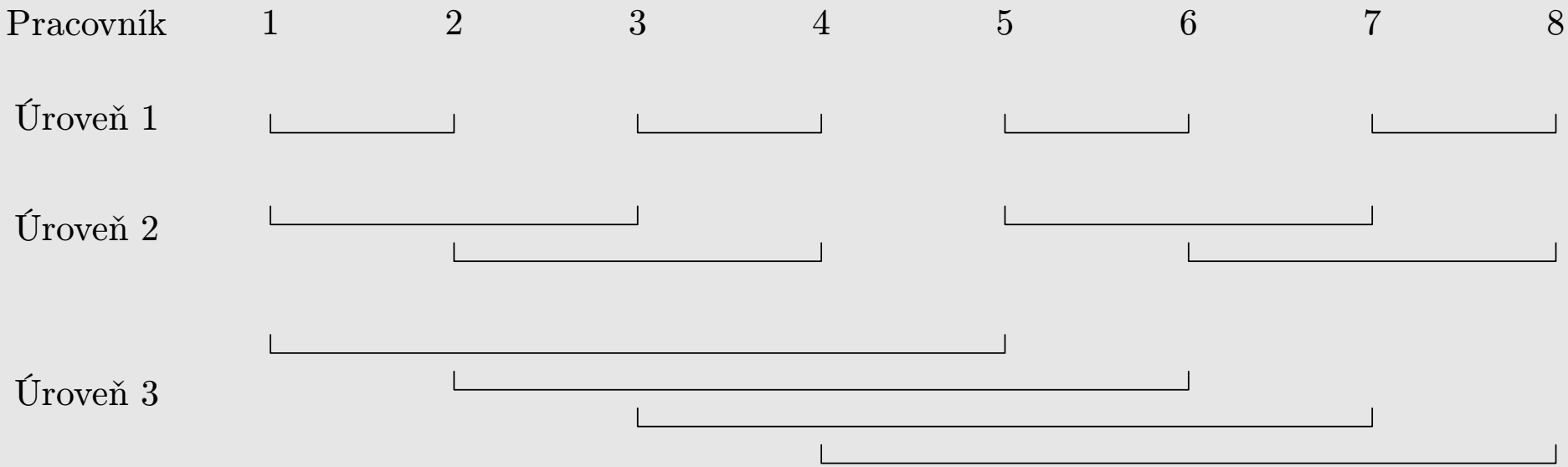
```
(await (= (aref arrive i) 0))
(setf (aref arrive i) 1)
(await (= (aref arrive j) 1))
(setf (aref arrive j) 0)
```

```
(await (= (aref arrive j) 0))
(setf (aref arrive j) 1)
(await (= (aref arrive i) 1))
(setf (aref arrive i) 0)
```

2., 3. a 4. řádek z principů synchronizace vlajkami, 1. proti předběhnutí k bariéře a nastavení vlajky dřív než ji druhý proces resetuje

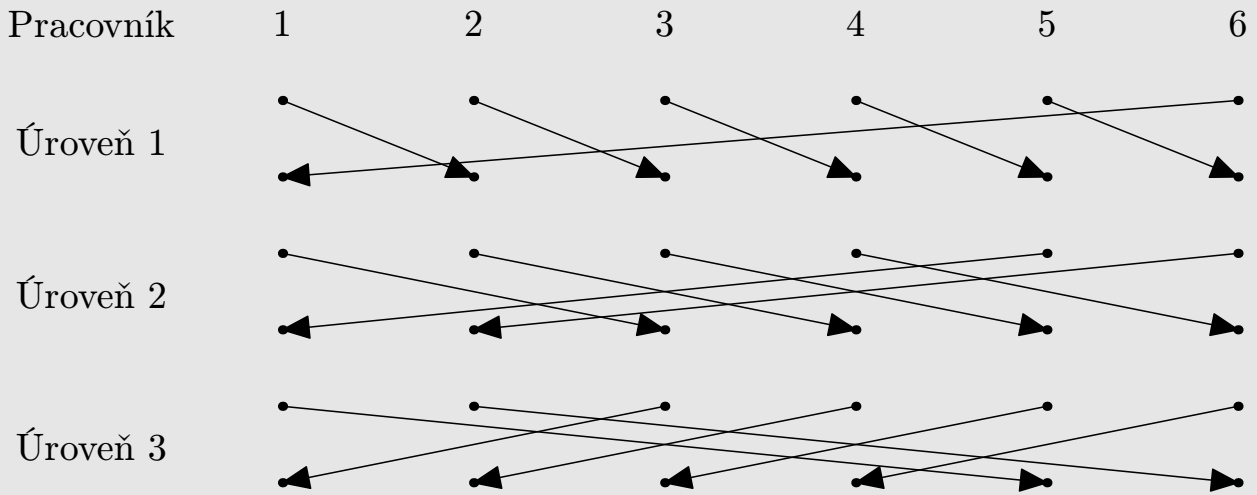
Bariéry: Butterfly (motýlovitá) bariéra

- pro počet procesů $N = 2^X$ (pojmenování podle tvaru kombinace párů procesů)
- $log_2 N$ úrovní, na úrovni $0 \leq s < X$ se navzájem synchronizují procesy ve vzdálenosti 2^s
- po projití všech úrovní se proces přímo nebo nepřímo synchronizoval s každým jiným procesem



Bariéry: Diseminační bariéra

- pro počet procesů $N \neq 2^X$ lze použít butterfly bariéru pro N rovno nejbližšímu většímu 2^Y
- lepší: diseminační, distribuční (pojmenování podle způsobu distribuce informace na N cílů v logaritmickém čase)
- $log_2 N$ úrovní, na úrovni $0 \leq s < X$ se proces i synchronizuje s procesem ve vzdálenosti 2^s modulo N:
  - nastaví vlajku procesu $(i + 2^s)\ mod\ N$
  - čeká na a resetuje vlajku procesu $(i - 2^s)\ mod\ N$



Datově paralelní algoritmy

= iterativní algoritmy, které opakovaně a paralelně operují nad sdíleným datovým polem

- pro synchronizaci použity bariéry

Synchronní multiprocesory (SIMD)

- na asynchronních multiprocesorech (MIMD) mohou procesy běžet různě rychle
- SIMD: více stejných procesů (dat), 1 proud vykonávání $\Rightarrow$ všechny procesy běží synchronně stejnou rychlostí, implicitní bariéra za každou atomickou instrukcí
- vhodné pro datově paralelní algoritmy – nejsou potřeba (explicitní) bariéry

Datově paralelní algoritmy: prefixy pole

- často potřeba aplikovat operaci na všechny prvky pole (např. průměr)
- iterativně: operace na rostoucí prefixy pole $\longrightarrow$ paralelní prefixy
- aplikace: zpracování lineárních datových struktur, matic, obrazu, formálních jazyků, ...

Př. **Součet prvků prefixů pole**

- lze použít na libovolný asociativní operátor ($+, \times, \wedge, \vee, max, min, \ldots$)
- pole obsahuje alespoň 1 prvek

Sekvenčně:

```
(setf (aref sum 0) (aref a 0))
; SUM : sum[i] = a[0] + ... + a[i]
(loop for i from 1 to (- n 1) do
      (setf (aref sum i) (+ (aref sum (- i 1)) (aref a i))))
```

Datově paralelní algoritmy: prefixy pole

Paralelně:

- pro každý prvek (nebo kus) pole proces
- inicializovat pole součtů na pole prvků a přičítat ke každému součtu předchozí součet ve zdvojnásobující se vzdálenosti
- v každé iteraci se zdvojnásobí počet sčítaných prvků $\longrightarrow log_2 N$ kroků
- potřeba bariér pro zabránění interferenci

prvky pole	1	2	3	4	5	6
součty do vzdálenosti 1	1	3	5	7	9	11
součty do vzdálenosti 2	1	3	6	10	14	18
součty do vzdálenosti 4	1	3	6	10	15	21

Datově paralelní algoritmy: prefixy pole

```
(setf sum (make-array n) old (make-array n))
(co-dotimes (i n)
  (let ((d 1))
    (setf (aref sum i) (aref a i))
    (barrier)
    ; SUM : sum[i] = a[i - d + 1] + ... + a[i]
    (loop while (< d n) do
          (progn
            (setf (aref old i) (aref sum i))
            (barrier)
            (if (>= (- i d) 0)
                (setf (aref sum i)
                      (+ (aref old (- i d)) (aref sum i))))
            (barrier)
            (setf d (* 2 d))))))
```

Datově paralelní algoritmy: konec seznamu

- operace ve spojovaných datových strukturách v logaritmickém čase → (např.) vyvážené stromy
- s datově paralelním algoritmy i pro lineární datové struktury, např. seznamy

Př. **Nalezení konce jednosměrného seznamu**

- seznam prvků reprezentován polem indexů dalšího prvku seznamu (pro poslední prvek je index dalšího roven -1)
- seznam není cyklický a obsahuje alespoň 2 prvky

Sekvenčně:

```
(setf end (aref links 0))
(loop while (not (= (aref links end) -1)) do
      (setf end (aref links end)))
```

Datově paralelní algoritmy: konec seznamu

Paralelně:

- pro každý prvek (nebo kus) seznamu proces
- inicializovat pole indexů konce seznamu na pole indexů dalšího prvku seznamu a nastavovat každý index konce na index konce prvku na indexu konce
- po každé iteraci se zdvojnásobí vzdálenost konce od prvku $\longrightarrow log_2 N$ kroků
- potřeba bariér pro zabránění interferenci

indexy dalšího prvku seznamu	1	2	3	4	5	−1
indexy konce seznamu po 1. iteraci	2	3	4	5	5	−1
indexy konce seznamu po 2. iteraci	4	5	5	5	5	−1
indexy konce seznamu po 3. iteraci	5	5	5	5	5	−1

Datově paralelní algoritmy: konec seznamu

Datově paralelní algoritmy: další

Př. **Konvergenční (numerické) metody**

```
inicializace
(loop while (není řešení nebo nekonverguje) do
      (výpočet nové hodnoty pro každý bod)
      (test nelezení řešení nebo konvergence))
```

- pro každý bod (nebo blok bodů) proces
- bariéry pro synchronizaci: po inicializaci, před aktualizací bodu a testem, po iteraci
- např. diferenciální rovnice nebo zpracování obrazu

Semafory (Dijkstra 1968)

- synchronizační protokoly používající jen aktivní čekání mohou být složité návrhem, implementací i dokazováním
- aktivní čekání je neefektivní na 1 CPU, ale i na N CPU (více procesů než CPU) – "čekací" cyklus
- speciální nástroj (pasivně) blokující procesy?

Semafor

- jednoduché použití na kritické sekce i na podmíněnou synchronizaci
- implementace aktivním čekáním i pasivním blokováním procesů ve spolupráci s plánovačem procesů OS
- motivace: semafor na silniční křižovatce nebo železnici – signál indikující volno nebo obsazeno, nastavován a rušen, vyjadřuje podmínku vzájemného vyloučení vícenásobného obsazení kritického místa

Semafory: notace a sémantika

- semafor = instance abstraktního datového typu + operace **P** a **V**
- **V** signalizuje událost, **P** blokuje proces, dokud nenastane událost
- implementace musí zachovávat invariant semaforu *SEM*:

Invariant semaforu: Nechť nP je počet ukončených operací **P**, nV je počet ukončených operací **V** a *init* je počáteční hodnota semaforu. Pak ve všech viditelných stavech programu platí $SEM : nP \leq nV + init$.
Tj. **P** potenciálně blokuje, dokud nebude proveden odpovídající počet **V**.

- nejjednodušší: semafor = nezáporné číslo $s = init + nV - nP$, potom $SEM : s \geq 0$
- **P** zvyšuje nP, tj. snižuje s, pro zajištění SEM musí být podmíněna $s > 0$
- **V** zvyšuje nV, tj. zvyšuje s, SEM platí, nemusí být podmíněna

Semafory: notace a sémantika

P(s): `(await (> s 0) (decf s))`
V(s): `(atomically (incf s))`

Obojí atomické a jediné operace na semaforu, tzn. nelze zjišťovat hodnotu semaforu.

- **obecný semafor**: s nezáporné číslo
- **binární semafor**: $b \in \{0, 1\}$, invariant $BSEM : 0 \leq b \leq 1 \rightarrow$ **V** podmíněná $b < 1$
  P(b): `(await (> b 0) (decf b))`
  V(b): `(await (< b 1) (incf b))`
- místo binárního semaforu lze použít obecný, ale musí být explicitně zaručeno, že **V** bude vykonávána jen když platí $s = 0$, jinak není zajištěn invariant $BSEM$

Struktury `semaphore` a `binary-semaphore`.

Semafory: notace a sémantika

- operace pomocí await $\Rightarrow$ sémantika podle Pravidla synchronizace
- substituce $s > 0$ za B a `(decf s)` za S: $\{P \wedge s > 0\}$ `(decf s)` $\{Q\}$
- $+$ Axiom přiřazení: $(P \wedge s > 0) \Rightarrow Q^s_{s-1}$

Pravidla obecného semaforu:

$$\frac{(P \wedge s > 0) \Rightarrow Q^s_{s-1}}{\{P\}\ \mathbf{P}(s)\ \{Q\}} \qquad \frac{P \Rightarrow Q^s_{s+1}}{\{P\}\ \mathbf{V}(s)\ \{Q\}}$$

Pravidla binárního semaforu:

$$\frac{(P \wedge b > 0) \Rightarrow Q^b_{b-1}}{\{P\}\ \mathbf{P}(b)\ \{Q\}} \qquad \frac{(P \wedge b < 1) \Rightarrow Q^b_{b+1}}{\{P\}\ \mathbf{V}(b)\ \{Q\}}$$

- jestliže $s > 0$ nabyde platnosti a platí, pro ukončení $\mathbf{P}(s)$ stačí slabě férové plánování
- jestliže $s > 0$ platí nekonečněkrát, pro ukončení $\mathbf{P}(s)$ je potřeba silně férové plánování
- podobně pro $\mathbf{V}(b)$, pro ukončení $\mathbf{V}(s)$ stačí nepodmíněně férové plánování

Semafory: základní použití

- přímá podpora protokolů kritické sekce
- jednoduchá podmíněná synchronizace (podmínka = výskyt události)
- techniky programování se semafory

Semafory: Kritické sekce: Výměna proměnných (Andrews 1989)

- mnoho řešení kritické sekce představuje mnoho globálních invariantů specifikujících vzájemné vyloučení
- semafory = čísla $\rightarrow$ invarianty pomocí čísel, atomické akce lze pak přímo změnit na operace se semafory
- $in[i] = 1$ když $P[i]$ je v kritické sekci (vstupní protokol), jinak $in[i] = 0$ (výstupní protokol), nejvýše 1 proces v kritické sekci: $CS : in[0] + \ldots + in[N-1] \leq 1$
- zajištění CS:
  - platnost po vstupním protokolu: $in[0] + \ldots + 1 + \ldots + in[N-1] \leq 1$, všechny $in[i] = 0$ nebo $1 \Rightarrow \forall i : 0 \leq i < N : in[i] = 0$ (i $P[i]$ není před vstupem v kritické sekci) $\rightarrow$ zesílení podmínky protokolu
  - výstupní protokol nemůže porušit CS
- platí vzájemné vyloučení, absence deadlocku i zbytečného čekání (pomocí vyloučení konfigurací), pro zaručení vstupu je potřeba silně férové plánování

Semafory: Kritické sekce: hrubé řešení

```
(setf in (make-array n :initial-element 0))
; CS : in[0] + ... + in[N-1] ≤ 1
(co-dotimes (i n)
  (loop while t do
        (progn
          (await (every #'zerop in) (setf (aref in i) 1))
          ; kritická sekce
          (setf (aref in i) 0)
          ; nekritická sekce
          )))
```

Semafory: Kritické sekce: Výměna proměnných

Semafory implementující atomické akce?

- **výměna proměnných** tak, že atomické akce = operace na semaforu: $mutex = 1 - (in[0] + \ldots + in[N-1])$, $CS' : mutex \geq 0$

```
(await (> mutex 0)
       (progn (decf mutex) (setf (aref in i) 1)))

(atomically (progn (incf mutex) (setf (aref in i) 0)))
```

- *in* pomocná proměnná, atomické akce = operace **P** a **V** na semaforu *mutex*
- řešení pro N procesů, jednodušší než řešení pomocí aktivního čekání

Semafory: Kritické sekce: řešení se semaforem

```
(setf mutex (make-semaphore 1))
; CS' :  mutex  ≥  0
(co-dotimes (i n)
  (loop while t do
        (progn
          (semaphore-P mutex)
          ; kritická sekce
          (semaphore-V mutex)
          ; nekritická sekce
          )))
```

Semafory: Výměna proměnných

Podmínky výměny proměnných: Atomické akce lze implementovat pomocí operací na semaforu jestliže:

1. v různých podmínkách jsou použity různé proměnné a jsou použity jen v atomických akcích
2. každou podmínku lze převést na tvar $vyraz > 0$, kde *vyraz* je číselný
3. každá podmíněná atomická akce obsahuje jednu dekrementaci hodnoty výrazu transformované podmínky
4. každá podmíněná atomická akce inkrementuje hodnotu výrazu transformované podmínky

Pak lze použít jeden semafor pro každou různou podmínku, původní proměnné v podmínkách se stanou pomocnými a atomické akce operacemi na semaforu.

Semafory: Bariéry: Signalizace události

- řešení aktivním čekáním pomocí vlajek
- semafory: jeden pro každou vlajku, nastavení vlajky = **V**, čekání na a zrušení vlajky = **P**
- 2 procesy, potřeba: 1. žádný proces nesmí za bariéru, dokud k ní nedorazí oba procesy, 2. znovupoužitelnost bariéry pro synchronizaci po každé iteraci procesů
- potřeba zaznamenat příchod k i odchod z bariéry $\Rightarrow$ čítače projití kritickým místem
- globální invariant: $BARRIER : depart1 \leq arrive2 \wedge depart2 \leq arrive1$, tj. P1 nemůže za bariéru vícekrát než P2 dorazí k bariéře a naopak
- zajištění *BARRIER*: zesílení podmínek před odchodem z bariéry, $depart1 < arrive2$ a $depart2 < arrive1$

Semafory: Bariéry: hrubé řešení

```
(setf arrive1 0 depart1 0 arrive2 0 depart2 0)
; BARRIER : depart1 ≤ arrive2 ∧ depart2 ≤ arrive1
(labels ((P1 ()
           (loop while t do
                 (progn
                   ; implementace procesu P1
                   (atomically (incf arrive1))
                   (await (< depart1 arrive2) (incf depart1))))
         (P2 ()
           (loop while t do
                 (progn
                   ; implementace procesu P2
                   (atomically (incf arrive2))
                   (await (< depart2 arrive1) (incf depart2))))
  (co-progn (P1) (P2)))
```

Semafory: Bariéry: výměna proměnných

Semafory implementující atomické akce?

- výměna proměnných: $barrier1 = arrive1 - depart2$, $barrier2 = arrive2 - depart1$, $BARRIER' : barrier1 \geq 0 \wedge barrier2 \geq 0$

```
(atomically (progn (incf barrier1) (incf arrive1)))

(await (> barrier2 0)
       (progn (decf barrier2) (incf depart1)))
```

- *arrive* a *depart* pomocné proměnné, atomické akce = operace **P** a **V** na semaforech $barrier1$ a $barrier2$

Semafory: Bariéry: řešení pomocí semaforů

```
(setf barrier1 (make-semaphore) barrier2 (make-semaphore))
; BARRIER' : barrier1 ≥ 0 ∧ barrier2 ≥ 0
(labels ((P1 ()
           (loop while t do
                 (progn
                   ; implementace procesu P1
                   (semaphore-V barrier1)
                   (semaphore-P barrier2))))
         (P2 ()
           (loop while t do
                 (progn
                   ; implementace procesu P2
                   (semaphore-V barrier2)
                   (semaphore-P barrier1)))))
  (co-progn (P1) (P2)))
```

Semafory: Bariéry: Signalizace události

- semafory jako signály indikující událost příchodu k bariéře
- více procesů:
  - přímé použití řešení pro 2 procesy v butterfly (motýlovité) nebo diseminační bariéře: proces signalizuje semafor jiného procesu a čeká na signalizaci svého semaforu
  - použití semaforů jako vlajek v bariéře vlajkami-koordinátorem nebo stromově kombinované bariéře

Semafory: Producenti a konzumenti: Dělené (Split) binární semafory (Hoare 1974, Dijkstra 1979)

- obecněji více producentů a konzumentů
- operace uložení a přečtení, potřeba zajistit nepřepisování před přečtením a přečtení jedenkrát $\longrightarrow$ střídání operací
- inkrementující čítače pro indikaci projití kritickým bodem: započetí a ukončení operací uložení (deposit) a přečtení (fetch)
- globální invariant: $PC : inD \leq afterF + 1 \wedge inF \leq afterD$, tj. uložení může začít maximálně o 1 vícekrát než ukončení čtení a čtení nemůže začít vícekrát než ukončení uložení (špatný stav: 2 a více stejných operací za sebou)
- zajištění PC: zesílení podmínek započetí operací, $inD \leq afterF$ a $inF < afterD$

Semafory: Producenti a konzumenti: hrubé řešení

```
(setf inD 0 afterD 0 inF 0 afterF 0)
; PC : inD ≤ afterF + 1 ∧ inF ≤ afterD
(labels ((Producenti ()
           (co-dotimes (i M)
             (loop while t do
                   ; vyprodukuj data m
                   (await (<= inD afterF) (incf inD))
                   (setf buf m)
                   (atomically (incf afterD)))))
         (Konzumenti ()
           (co-dotimes (i N)
             (loop while t do
                   (await (< inF afterD) (incf inF))
                   (setf m buf)
                   (atomically (incf afterF))
                   ; zkonzumuj data m
                   ))))
  (co-progn (Producenti) (Konzumenti)))
```

Semafory: Producenti a konzumenti: výměna proměnných

Semafory implementující atomické akce?

- výměna proměnných: $empty = afterF - inD + +$, $full = afterD - inF$, $PC' : 0 \leq empty + full \leq 1$

```
(await (> empty 0)
       (progn (decf empty) (incf inD)))

(atomically (progn (incf full) (incf afterD)))
```

- *in* a *after* pomocné proměnné, atomické akce = operace **P** a **V** na semaforech *empty* a *full*

Semafory: Producenti a konzumenti: řešení pomocí semaforů

```
(setf empty (make-semaphore 1) full (make-semaphore))
; PC' : 0 ≤ empty + full ≤ 1
(labels ((Producenti ()
           (co-dotimes (i M)
             (loop while t do
                   ; vyprodukuj data m
                   (semaphore-P empty)
                   (setf buf m)
                   (semaphore-V full))))
         (Konzumenti ()
           (co-dotimes (i N)
             (loop while t do
                   (semaphore-P full)
                   (setf m buf)
                   (semaphore-V empty)
                   ; zkonzumuj data m
                   ))))
  (co-progn (Producenti) (Konzumenti)))
```

Semafory: Producenti a konzumenti: Dělené (Split) binární semafory

- *empty* a *full* jsou binární semafory
- dohromady tvoří **dělený (split) binární semafor**: nejvýše jeden z nich je v daném čase roven 1 (PC')
  Dělený (split) binární semafor: Binární semafory $b_1, \ldots, b_n$ tvoří dělený (split) binární semafor, jestliže platí globální invariant: $SPLIT : 0 \leq b_1 + \ldots + b_n \leq 1$.
- jednotlivé binární semafory lze nahlížet jako jeden binární semafor rozdělený do více binárních semaforů
- lze pomocí něj implementovat vzájemné vyloučení: proces vykonává **P** na jednom ze semaforů a **V** na jiném, mezi **P** a **V** je vzájemně vyloučený kód

Semafory: Omezené buffery: Počítání zdrojů

- 1 položka bufferu stačí pro zhruba stejně rychlé procesy producenta a konzumenta, tj. stejnou frekvenci produkce a konzumace
- běžně je ale rychlost různá (např. výpočet + zápis výsledků a čtení výsledků) $\rightarrow$ potřeba větší buffer pro redukci čekání = **omezený buffer**

Selektivní vzájemné vyloučení

= vzájemné vyloučení procesů

1. přistupujících k překrývajícím se množinám sdílených prostředků
2. kombinujících paralelní a exkluzivní přístup různých skupin procesů ke sdíleným prostředkům
3. ...

Klasické příklady:

1. večeřící filozofové
2. čtenáři a písaři
3. ...

Večeřící filozofové (Dijkstra 1968)

- i když spíše učebnicový příklad, je podobný skutečným problémům, kdy procesy vyžadují současný přístup k více prostředkům

Problém: N filozofů sedí u kulatého stolu, přemýšlí nebo jedí pomocí dvou vidliček, kterých je mezi nimi celkem N, každý z nich používá jednu vidličku nalevo a jednu napravo o něj. Simulace? Zabránění situaci (deadlock), kdy všichni chtějí jíst, ale žádný nemůže vzít obě vidličky, protože každý má jednu.

- dva sousedící filozofové nemohou jíst zároveň a zároveň může jíst nejvýše $\frac{N}{2}$ z nich

Návrh:

```
(labels ((Filozof (i)
           (loop while t do
                 ; filozof i premýšlí
                 ; zvednutí vidliček
                 ; filozof i jí
                 ; položení vidliček
                 )))
  (co-dotimes (i n) (Filozof i)))
```

Večeřící filozofové: synchronizace

- potřeba implementovat zvedání a pokládání vidliček, což je sdílený prostředek
- můžeme pro každého filozofa a vidličku zaznamenávat, zda ji má nebo ne, podobně jako když proces je v kritické sekci nebo ne
- nebo můžeme použít čítače zvednutí ($up[i]$) a položení ($down[i]$) vidličky
- vidlička nemůže být položena vícekrát než je zvednuta a nemůže být zvednuta vícekrát (více filozofy) za sebou bez položení:
  $FORKS : \forall i : 0 \leq i < N : down[i] \leq up[i] \leq down[i] + 1$
- filozof vezme obě vidličky i a $(i + 1) \mod N$, jí, pak obě vidličky položí = inkrementace a dekrementace up a $down$
- zajištění globálního invariantu $FORKS$: podmínky zvedání vidliček

Večeřící filozofové: hrubé řešení

```
(setf up (make-array n :initial-element 0))
(setf down (make-array n :initial-element 0))
; FORKS : ∀ i : 0 ≤ i < N : down[i] ≤ up[i] ≤ down[i] + 1
(labels ((Filozof (i)
           (loop while t do
                 ; filozof i premýšlí
                 (await (= (aref up i) (aref down i))
                        (incf (aref up i)))
                 (await (= (mod (+ (aref up i) 1) n)
                           (mod (+ (aref down i) 1) n))
                        (incf (mod (+ (aref up i) 1) n)))
                 ; filozof i jí
                 (incf (aref down i))
                 (incf (mod (+ (aref down i) 1) n)))))
  (co-dotimes (i n) (Filozof i)))
```

Večeřící filozofové: výměna proměnných

Semafory implementující atomické akce?

- výměna proměnných: $forks[i] = 1 - (up[i] - down[i])$, proměnné *up* a *down* pomocné, $forks[i]$ semafor, akce:

```
(semaphore-P (aref forks i))
(semaphore-P (aref forks (mod (+ i 1) n)))

(semaphore-V (aref forks i))
(semaphore-V (aref forks (mod (+ i 1) n)))
```

- *PROBLÉM*: všichni filozofové vezmou 1 vidličku a nemohou vzít druhou = deadlock, cyklické čekání
- řešení: jeden z filozofů vezme vidličky v opačném pořadí (formálně pomocí proměnných indikujících, kteří filozofové zvedli které vidličky, a vyloučení konfigurací)
- jiná řešení: ?

Večeřící filozofové: řešení pomocí semaforů

```
(setf forks (make-array n))
(loop for i from 0 to (- n 1) do (setf (aref forks i) (make-semaphore 1)))
(labels ((Filozof (i)
           (loop while t do
                 ; filozof i premýšlí
                 (semaphore-P (aref forks i))
                 (semaphore-P (aref forks (mod (+ i 1) n)))
                 ; filozof i jí
                 (semaphore-V (aref forks i))
                 (semaphore-V (aref forks (mod (+ i 1) n)))))
         (Opacny-Filozof (i)
           (loop while t do
                 ; filozof i premýšlí
                 (semaphore-P (aref forks (mod (+ i 1))))
                 (semaphore-P (aref forks i))
                 ; filozof i jí
                 (semaphore-V (aref forks (mod (+ i 1))))
                 (semaphore-V (aref forks i)))))
  (co-progn (co-dotimes (i (- n 1)) (Filozof i)) (Opacny-Filozof (- n 1))))
```

Čtenáři a písaři (Courtois, Heymans, Parnas 1971)

- klasický a zároveň praktický synchronizační problém

Problém: 2 druhy procesů, čtenáři a písaři, sdílejí data, čtenáři je čtou, písaři čtou i zapisují. Pro zabránění interferencí mezi zápisy a čteními vedoucím k nekonzistentnímu stavu dat musí mít písaři výlučný přístup k datům, pokud žádný písař s daty nepracuje, může číst libovolný počet čtenářů.

- třídy procesů soutěží o přístup k datům: písaři mezi sebou, čtenáři jako třída s písaři
- písaři se vzájemně vylučují $\Rightarrow$ kritická sekce: proměnné $writing[j]$ a $\sum_j writing[j] \leq 1$
- čtenáři jako třída vylučují písaře: proměnná $reading$ a $RW : reading + \sum_j writing[j] = SUM \leq 1$
- třída čtenářů: jen první čtenář písaře vylučuje a jen poslední je povoluje $\rightarrow$ čítač čtenářů nr, nastavit $reading$, když $nr = 1$, zrušit, když $nr = 0$
- změna čítače čtenářů a změna proměnné $reading$ atomicky!

Čtenáři a písaři: návrh řešení

```
(setf reading 0 nr 0 writing (make-array n :initial-element 0))
; RW : reading + ∑_j writing[j] = SUM ≤ 1
(labels ((Ctenar (i)
           (loop while t do
                 (atomically (progn
                               (incf nr)
                               (if (= nr 1) (incf reading))))
                 ; (nr > 0 ∧ reading = 1) ∨ (nr = 0 ∧ reading = 0)
                 ; čtenář i čte
                 (atomically (progn
                               (decf nr)
                               (if (= nr 0) (decf reading))))))
                 ; (nr > 0 ∧ reading = 1) ∨ (nr = 0 ∧ reading = 0)
         (Pisar (i)
           (loop while t do
                 (atomically (incf (aref writing i)))
                 ; písař i píše
                 (atomically (decf (aref writing i))))))
  (co-progn (co-dotimes (i m) (Ctenar i))
    (co-dotimes (i n) (Pisar i))))
```

Čtenáři a písaři: synchronizace

- zajištění globálního invariantu *RW*: při nastavení *reading* musí být $\sum_j writing[j] = 0$ a při nastavení $writing[j]$ musí být $reading + \sum_{k \neq j} writing[k] = 0$:

  ```
  (await (= SUM 0) inkrementace)
  ```

- *PROBLÉM*: await uprostřed atomické akce
- řešení: jetliže dvě atomické akce používají rozdílné proměnné, mohou být vykonány paralelně
- *nr* používají jen čtenáři, *reading* čtenáři i písaři $\longrightarrow$ rozdělení složené atomické akce na část kritickou jen pro čtenáře a část kritickou pro čtenáře i písaře
- část kritická jen pro čtenáře $\Rightarrow$ kritická sekce pomocí semaforu *mutexR*
- zůstávají atomické akce s částmi kritickými pro čtenáře i písaře

Čtenáři a písaři: výměna proměnných

```
(semaphore-P mutexR)
(incf nr)
(if (= nr 1) (await (= SUM 0) (incf reading)))
(semaphore-V mutexR)

(semaphore-P mutexR)
(decf nr)
(if (= nr 0) (atomically (decf reading)))
(semaphore-V mutexR)
```

Semafory implementující atomické akce?

- výměna proměnných: $rw = 1 - SUM$, atomické akce = operace na semaforu *rw*

PROBLÉM?: Preference čtenářů: jestliže čtenář čte data a jiný čtenář a písař chtějí pracovat s daty, přednost má čtenář. To ale není fér, protože čtenáři tak mohou neustále blokovat písaře.

Čtenáři a písaři: řešení pomocí semaforů

```
(setf nr 0 mutexR (make-semaphore 1) rw (make-semaphore 1))
(labels ((Ctenar (i)
           (loop while t do
                 (semaphore-P mutexR)
                 (incf nr)
                 (if (= nr 1) (semaphore-P rw))
                 (semaphore-V mutexR)
                 ; čtenář i čte
                 (semaphore-P mutexR)
                 (decf nr)
                 (if (= nr 0) (semaphore-V rw))
                 (semaphore-V mutexR)))
         (Pisar (i)
           (loop while t do
                 (semaphore-P rw)
                 ; písař i píše
                 (semaphore-V rw))))
  (co-progn (co-dotimes (i m) (Ctenar i))
    (co-dotimes (i n) (Pisar i))))
```

Obecná podmíněná synchronizace

- předchozí řešení problému čtenářů a písařů je pomocí vzájemného vyloučení, obsahuje překrývající se kritické sekce čtenárů a písařů
- jednodušší řešení: evidovat počty čtenářů (nr) a písařů (nw) pracujících se sdílenými daty a omezit počty
- špatný stav: $(nr > 0 \wedge nw > 0) \vee nw > 1 \Rightarrow$
  $RW : (nr = 0 \vee nw = 0) \wedge nw \leq 1$
- zajištění RW: inkrementaci nr omezit $nw = 0$, inkrementaci nw omezit $nr = 0 \wedge nw = 0$
- dekrementace neomezovat, obecně není potřeba zdržovat proces snažící se ukončit práci se sdílenými daty

Čtenáři a písaři: hrubé řešení

```
(setf nr 0 nw 0)
; RW : (nr = 0 ∨ nw = 0) ∧ nw ≤ 1
(labels ((Ctenar (i)
           (loop while t do
                 (await (= nw 0) (incf nr))
                 ; čtenář i čte
                 (atomically (decf nr))))
         (Pisar (i)
           (loop while t do
                 (await (and (= nr 0) (= nw 0)) (incf nw))
                 ; písař i píše
                 (atomically (decf nw)))))
  (co-progn (co-dotimes (i m) (Ctenar i))
    (co-dotimes (i n) (Pisar i))))
```

Předávání peška (Andrews 1989)

- *PROBLÉM*: různé podmínky v await se překrývají (obsahují společné proměnné) $\Rightarrow$ nelze použít výměnu proměnných pro implementaci pomocí semaforu (nesplněny podmínky výměny)
- jeden semafor nemůže blokovat na základě dvou různých podmínek
- řešení: technika **předávání peška (passing the baton)**
  - používá dělené binární semafory pro vzájemné vyloučení a pro kontrolu, který proces bude odblokován
  - lze pommocí ní implementovat await (a atomically), tj. libovolnou podmíněnou (a nepodmíněnou) synchronizaci
- atomické akce tvaru:
  F_1: `(atomically (`S_j`))` F_2: `(await (`B_j`) (`S_j`))`
- dělený binární semafor pro vzájemné vyloučení atomických akcí (e) a každou podmíněnou synchronizaci (b_j, blokuje dokud není splněna podmínka B_j) + čítače (d_j) procesů čekajících (blokovaných) na splnění podmínky

Předávání peška: implementace

F_1:

```
(binary-semaphore-P e) ; I
(Si)                   ; I
(SIGNAL)
```

F_2:

```
(binary-semaphore-P e)                      ; I
(if (not Bj) (progn
               (incf dj)
               (binary-semaphore-V e)
               (binary-semaphore-P bj))) ; I ∧ Bj
(Sj)                                        ; I
(SIGNAL)
```

SIGNAL:

```
(cond ((and (B1) (> d1 0)) (progn
                             ; I ∧ B1
                             (decf d1)
                             (binary-semaphore-V b1)))
      ...
      (t (binary-semaphore-V e))) ; I
```

Předávání peška: implementace

- dělený binární semafor: nejvýše jeden z binárních semaforů e a b_j je signalizován ($e + \sum_j b_j \leq 1$), části mezi **P** a **V** jsou vzájemně vyloučené
- podmínky B_j garantovaně platí před akcemi S_j (jsou v kritických sekcích implementovaných binárními semafory b_j)
- nemůže nastat deadlock, protože binární semafory b_j jsou signalizovány jen když na nich nějaký proces čeká nebo bude následující akcí čekat (signál se neztratí) a tyto semafory jsou součástí děleného binárního semaforu (může být signalizován nejvýše jeden)
- předávání peška = způsob signalizace procesů: když je proces v kritické sekci (mezi **P** a **V** binárního semaforu b_j), má peška, kterého v části *SIGNAL* předá jinému procesu, který má nyní splněnu podmínku B_j vstupu, nebo procesu, který čeká na binárním semaforu e na vzájemné vyloučení atomických akcí

Čtenáři a písaři (Dijkstra 1979)

- 2 různé podmínky podmíněné synchronizace $\longrightarrow$ 2 binární semafory (r a w) blokující do splnění podmínek ($nw = 0$ a $nr = 0 \wedge nw = 0$) a a čítače (dr a dw) blokovaných procesů
- $SIGNAL_1$ u čtenářů v await, $SIGNAL_2$ u čtenářů v atomically, $SIGNAL_3$ u písařů v await, $SIGNAL_4$ u písařů v atomically
- (obecně) z prekondicí podmínek v části $SIGNAL$ lze podmínky zjednodušit nebo odstranit:
  - čtenáři: platí $nr > 0 \wedge nw = 0$ před $SIGNAL_1$ a $nw = 0 \wedge dr = 0$ před $SIGNAL_2$
  - písaři: platí $nr = 0 \wedge nw > 0$ před $SIGNAL_3$ a $nr = 0 \wedge nw = 0$ před $SIGNAL_4$
- v části $SIGNAL_4$ lze zaměnit pořadí podmíněných signalizací blokovaného čtenáře a písaře
- signalizovaný čtenář kaskádovitě signalizuje případné další blokované čtenáře (inkrementuje nr a případně signalizuje binární semafor r, který stále blokuje jiné čtenáře)

Čtenáři a písaři: řešení pomocí semaforů – čtenář

```
(setf nr 0 nw 0) ; RW : (nr = 0 ∨ nw = 0) ∧ nw ≤ 1
(setf e (make-binary-semaphore 1) r (make-binary-semaphore)
      w (make-binary-semaphore))
; SPLIT : 0 ≤ e + r + w ≤ 1
(setf dr 0 dw 0) ; COUNTERS : dr ≥ 0 ∧ dw ≥ 0
(labels ((Ctenar (i)
           (loop while t do
                 (binary-semaphore-P e)
                 (if (> nw 0) (progn
                                (incf dr)
                                (binary-semaphore-V e)
                                (binary-semaphore-P r)))
                 (incf nr)
                 (cond ((> dr 0) (progn
                                   (decf dr)
                                   (binary-semaphore-V r)))
                       (t (binary-semaphore-V e)))
                 ; čtenář i čte
                 (binary-semaphore-P e)
                 (decf nr)
                 (cond ((and (= nr 0) (> dw 0)) (progn
                                                  (decf dw)
                                                  (binary-semaphore-V w)))
                       (t (binary-semaphore-V e)))))
```

Čtenáři a písaři: řešení pomocí semaforů – písař

```
        (Pisar (i)
          (loop while t do
                (binary-semaphore-P e)
                (if (or (> nr 0) (> nw 0)) (progn
                                             (incf dw)
                                             (binary-semaphore-V e)
                                             (binary-semaphore-P w)))
                (incf nw)
                (binary-semaphore-V e)
                ; písař i píše
                (binary-semaphore-P e)
                (decf nw)
                (cond ((> dr 0) (progn
                                  (decf dr)
                                  (binary-semaphore-V r)))
                      ((> dw 0) (progn
                                  (decf dw)
                                  (binary-semaphore-V w)))
                      (t (binary-semaphore-V e))))))
(co-progn (co-dotimes (i m) (Ctenar i))
  (co-dotimes (i n) (Pisar i))))
```

Čtenáři a písaři: alternativní plánování

- stále preference čtenářů, ale lze jednoduše modifikovat změnou podmínek pro plánování v jiném pořadí
- neporuší se struktura řešení = vlastnost techniky předávání peška

Preference písařů:

1. nový čtenáři čekají, jestliže čeká písař $\rightarrow$ zesílení podmínky ve vstupním protokolu čtenářů: $nw > 0 \wedge dw > 0$
2. čtenář je signalizován jen když nečeká písař $\rightarrow$ zesílení podmínky signalizace čtenáře ve výstupním protokolu písařů: $dr > 0 \wedge dw = 0$

Střídání třídy čtenářů a písařů (když obojí čeká): ?

- pořadí různých procesů lze kontrolovat i jemněji, co nelze kontrolovat?
- techniku lze použít i pro řešení obecného problému **alokace zdrojů** – rozhodování, kdy a jakému procesu může být přidělen sdílený zdroj

Implementace semaforů

- operace **P** a **V** definovány pomocí await a atomically $\rightarrow$ implementace pomocí kritických sekcí pro podmíněnou synchronizaci a vzájemné vyloučení s aktivním čekáním:
  1. jedna společná kritická sekce:
     P(s):

     ```
     CSenter
     (loop while (not (> s 0)) do (progn CSexit Delay CSenter))
     (decf s)
     CSexit
     ```

     V(s): `CSenter (incf s) CSexit`
  2. dvě kritické sekce:
     P(s):

     ```
     CSenter1
     (loop while (not (> s 0)))
     CSenter2 (decf s) CSexit2
     CSexit1
     ```

     V(s): `CSenter2 (incf s) CSexit2`

Implementace semaforů

- pasivní čekání: potřeba podpory pasivních kritických sekcí a čekání na splnění podmínky operačním systémem
- nejčastěji je přímo dostupná (pasivní) implementace semaforů v operačním systému

I když jsou jednodušší na použití než řešení s HW atomickými instrukcemi a aktivním čekáním a umožňují pasivní (blokující) čekání,

semafory jsou nízkoúrovňové synchronizační primitivum nejčastěji používané v operačním systému a programovacích jazycích pro implementaci synchronizačních primitiv na vyšší úrovni (podmíněné kritické regiony, monitory, aj.).

Synchronizace v operačních systémech (OS)

- uživatelské procesy a vlákna běží paralelně a asynchronně vedle sebe (přepínaně na 1 CPU nebo paralelně na více CPU)
- vlákna jádra OS (jaderná vlákna) – více částí jádra vykonáváno paralelně a asynchronně vedle sebe
- chyby souběhu (race conditions) – procesy a vlákna přistupují zároveň ke stejným sdíleným datům a výsledek je závislý na pořadí přístupu
- v OS je mnoho částí běžících paralelně a přistupujících zároveň ke stejným datům (zdrojům), např. seznam otevřených souborů, procesů, atd. $\rightarrow$ potřeba **synchronizace**
- většina dnešních OS (včetně M$ Windows a GNU/Linuxu) neobsahuje prevenci, zabránění vzniku ani detekci a vyřešení deadlocku, protože nenastávají tak často, aby se vyplatilo je řešit

Synchronizace v OS – Kritické sekce

Preemtivnost jádra OS

- když proces volá službu OS (*system call*), běží po dobu vyřízení služby v *režimu jádra*
- nepreemtivní jádro = proces běžící v režimu jádra nebo jaderné vlákno není přepnuto na jiné, proces musí sám opustit režim jádra, být blokován nebo předat řízení CPU (zavolat plánovač procesů) $\Rightarrow$ datové struktury jádra OS není třeba chránit před současným vícenásobným přístupem, př. M$ Windows, některé UNIXy, Linux $<$ 2.6
- preemtivní jádro = proces běžící v režimu jádra nebo jaderné vlákno může být přepnuto na jiné $\Rightarrow$ potřeba chránit datové struktury jádra OS, př. Solaris, Linux $\geq$ 2.6

Synchronizace v OS – Kritické sekce

- programové (softwarové) řešení $\rightarrow$ aktivní čekání, př. Petersonův Tie-Breaker nebo Bakery algoritmus
- v jádře OS pro delší kritické sekce nepřijatelné
- pasivní čekání (blokování) – vyžaduje podporu HW
- na 1 CPU: po dobu kritické sekce zákaz HW i SW přerušení nebo preemtivnosti jádra
- na více CPU: zákaz přerušení na všech CPU by byl náročný (časově i koncepčně) a snižoval by výkon $\Rightarrow$ zákaz preemtivnosti jádra nebo aktivní čekání
- podpora HW: atomické instrukce Test-And-Set, Fetch-And-Add, Swap, aj., př. spin lock (**mutex**), Ticket algoritmus

Synchronizační primitiva v OS

Spin lock

- aktivní čekání s využitím HW instrukce Test-And-Set
- výhodné pro krátké kritické sekce na více CPU – blokování znamená přepínání procesů

Read-Write (RW) zámek

- zobecnění problému čtenáři-písaři na synchronizační primitivum pro synchronizaci procesů, které jen čtou a které i zapisují sdílený prostředek
- čtecí a zapisovací režim zámku – zámek ve čtecím režimu může mít více procesů zároveň
- vhodný pro situace, kdy lze procesy rozlišit na čtenáře a písaře a kdy je čtenářů víc

Synchronizační primitiva v OS

Semafor

- pomocí aktivního čekání: spin lock
- pomocí pasivního čekání:

  P(s):

```
(decf s)
(if (< s 0)
    (progn
      přidej proces do seznamu procesů čekajících na $s$
      blokuj proces = změň stav procesu na čekající a zavolej plánovač procesů
))
```

  V(s):

```
(incf s)
(if (<= s 0)
    (progn
      vyber proces ze seznamu procesů čekajících na $s$
      probuď tento proces = změn jeho stav na aktivní a zavolej plánovač proce
))
```

- semafor může mít zápornou hodnotu = počet čekajících procesů
- pro zaručení omezeného čekání (ukončení **P**) je seznam čekajících procesů organizován jako FIFO
- operace **atomicky** = kritická sekce

Další synchronizační primitiva

Synchronizační primitiva na vyšší úrovni

- spíše v prog. jazycích (např. C#, Java) a prog. knihovnách než v samotném OS
- **monitor** = ATD s privátními daty a operacemi nad nimi vykonávanými v rámci monitoru vzájemně vyloučeně, tj. uvnitř monitoru je vždy aktivní max. jeden proces
- **podmínka (podmíněná proměnná)** = proměnná s operacemi wait (čekání) a signal (signalizace, nezávislá na wait)
- **monitor s podmínkami** = ...
- implementace pomocí semaforů

Synchronizace jader OS M$ Windows XP a Linux

M$ Windows XP

- 1 CPU: zákaz přerušení (která mohou přistupovat k chráněnému prostředku)
- více CPU: aktivní čekání (spin lock) na krátké kritické sekce, jinak blokování (synchronizační primitivum?)

Linux (UNIX)

- 1 CPU: zákaz přerušení (Linux ¡ 2.6 a UNIX) nebo preemtivnosti jádra (Linux ¿= 2.6)
- více CPU: aktivní čekání (spin lock) pro krátké kritické sekce, jinak blokování semafory nebo RW zámky

Synchronizace procesů a vláken poskytovaná OS

API poskytované programům pro synchronizaci

- (téměř výhradně) v C – jádro OS je v C
- obsahuje:
  - funkce pro vytváření a rušení procesů a vláken
  - funkce pro komunikaci a synchronizaci procesů a vláken (pomocí sdílené paměti i posíláním zpráv), funkce pro **synchronizační primitiva**
  - ...
- uživatelské procesy – každý má svoji, nesdílenou (virtuální) paměť $\Rightarrow$ meziprocesová synchronizace pomocí sdílené paměti, posíláním zpráv nebo synchronizačních primitiv pro procesy
- uživatelská vlákna (= „odlehčené" procesy) – všechna vlákna jednoho procesu sdílí paměť procesu $\Rightarrow$ synchronizace pomocí sdílených proměnných nebo synchronizačních primitiv pro vlákna

API pro synchronizaci: M$ Windows

Win32 API

- synchronizační objekty: mutex, semafor, událost (podobná podmíněné proměnné), časovač (singalizace po uplynutí zadaného času), ...
- signalizovaný (objekt je dostupný, proces není blokován při požadování objektu) nebo nesignalizovaný (objekt není dostupný, proces je blokován) stav
- funkce pro požadování objektu: `WaitForSingleObject`, `WaitForMultipleObjects`, ...
- funkce pro signalizaci objektu: `ReleaseMutex`, `ReleaseSemaphore`, `SetEvent`, ...
- funkce pro kritickou sekci: `EnterCriticalSection`, `LeaveCriticalSection`, ...
- funkce pro atomicke operace: `InterlockedIncrement`, `InterlockedExchange`, ...
- ...

API pro synchronizaci: Linux (UNIX)

SYSTEM V IPC, Pthreads, POSIX

- Pthreads: funkce pro vytváření, rušení a operace s mutexy, podmíněnými proměnnými a RW zámky:
  `pthread_mutex_lock`, `pthread_mutex_unlock`, ...
  `pthread_cond_wait`, `pthread_cond_signal`, ...
- SYSTEM V IPC, POSIX: funkce pro vytváření, rušení a operace se semafory:
  `sem_wait`, `sem_post`, ...
  `semop`, ...
- jádro: funkce pro atomicke operace:
  `atomic_read`, `atomic_set`, `atomic_add`, ...
- ...