KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 4 ◇ poznámky k přednášce

5. Semafory

verze z 12. března 2025

1 Semafory

Vraťme se k Petersonovu algoritmu, jenž řeší problém kritické sekce pomocí aktivního čekání:

```
let ((in1 nil) (in2 nil) (last 1))
```

loop
  nekritická sekce
  (set! in1 t)
  (set! last 1)
  (await (or (not in2)
  (= last 2)))
  kritická sekce
  (set! in1 nil)
	loop
	  nekritická sekce
	  (set! in2 t)
	  (set! last 2)
	  (await (or (not in1)
	  (= last 1)))
	  kritická sekce
	  (set! in2 nil)

Nepohodlné je, že proces čekající ve vstupním protokolu vytěžuje procesor opakovaným vyhodnocováním podmínky, přestože je jasné, že podmínka bude splněna až poté, co druhý proces opustí kritickou sekci. Poznamenejme, že v případě, že je procesů nejvýše stejně jako procesorů, aktivní čekání stroj nijak nezpomaluje. V této přednášce si představíme *semafory*, což je synchronizační prostředek, který umožňuje mimo jiné vyřešit kritickou sekci bez aktivního čekání.

Semafory objevil nizozemský informatik Edsger Wybes Dijkstra kolem roku 1962 během programování operačního systému pro stroj Electrologica X8, který si můžete prohlédnout na obrázku 1. **Semafor** je abstraktní datová struktura, které při vytvoření zadáme celé nezáporné číslo `k` udávající **počet zdrojů**. Semafor vytvoříme konstruktorem `sem`. Tedy `(sem k)` je semafor s `k` zdroji. Počet zdrojů semaforu musí být vždy nezáporný. Počet zdrojů semaforu nemůžeme přímo číst ani měnit. Máme k dispozici jen dvě atomické operace `p` a `v`. Jejich název je odvozen od nizozemských slov *passering* a *vrijgave* vycházejících ze signalizace provozu na železnici. Pro zapamatování jmen budeme používat česká slova *počkat* a *vyhlásit*. Následuje definice operací.

- `p`
  Vem zdroj. Případně čekej, než je zdroj k dispozici.
- `v`
  Přidej zdroj.

Obrázek 1: Electrologica X8

Přestože počet zdrojů semaforu nemůžeme přímo číst, můžeme si zavést následující pomocné proměnné, které nám umožní se k počtu zdrojů dopočítat.

- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Fakt, že počet zdrojů semaforu je nezáporný vyjádříme invariantem:

```
(<= cp (+ cv k))
```

Podmínkou `(= s (+ k (- cv cp)))` definujeme aktuální počet zdrojů `s` nazývaný **hodnota semaforu**. Pro další úvahy ztotožníme semafor s jeho hodnotou. Pamatujme jen, že uživatel nemůže hodnotu semaforu přímo získat ani nastavit.

Invariant semaforu můžeme přímo vyjádřit pomocí jeho hodnoty: `(>= s 0)`. Podívejme se, jak bychom mohli definovat operace `p` a `v` nahrubo. Tedy za použití složených atomických příkazů.

Operace `p` snižuje o jedna hodnotu semaforu. Tedy:

```
(set! s (- s 1))
```

Aby operace zachovala podmínku `(>= s 0)` musel by být program

```
{(>= s 0)}
(set! s (- s 1))
{(>= s 0)}
```

pravdivý. Ten ovšem pravdivý není, protože pro `s` rovno nule skončí chybou. Požadovaný předpoklad příkazu obdržíme z pravidla přiřazení:

```
{(>= (- s 1) 0)}
(set! s (- s 1))
{(>= s 0)}
```

Což je po úpravě:

```
{(> s 0)}
(set! s (- s 1))
{(>= s 0)}
```

Použitím pravidla sladění obdržíme příkaz zachovávající podmínku `(>= s 0)`:

```
{(>= s 0)}
[(await (> s 0))
 (set! s (- s 1))]
{(>= s 0)}
```

Vychází nám, že operace `(p s)` je:

```
[(await (> s 0))
 (set! s (- s 1))]
```

Operace `v` zvyšuje o jedna hodnotu semaforu. Tedy:

```
(set! s (+ s 1))
```

Operace musí zachovávat podmínku semaforu. Tedy program:

```
{(>= s 0)}
(set! s (+ s 1))
{(>= s 0)}
```

musí být pravdivý. Pro důkaz pravdivosti použijeme pravidlo přiřazení:

```
{(>= (+ s 1) 0)}
(set! s (+ s 1))
{(>= s 0)}
```

Po úpravě obdržíme:

```
{(>= s -1)}
(set! s (+ s 1))
{(>= s 0)}
```

Protože podmínka `(>= s 0)` je silnější než podmínka `(>= s -1)`, můžeme použít pravidlo zesílení předpokladu a obdržet dokazovaný program:

```
{(>= s 0)}
(set! s (+ s 1))
{(>= s 0)}
```

Vychází nám, že operace `v` může být jen:

```
[(set! s (+ s 1))]
```

2 Semafory řeší kritickou sekci

V této kapitole si představíme obecný postup, jak převést hrubé řešení na jemné za použití semaforů. Postup si ukážeme na problému kritické sekce. Nejprve je potřeba vypracovat hrubé řešení problému.

V našem příkladu kritické sekce, bychom mohli vzít řešení z minulé přednášky. Pro opakování zkusíme řešení vypracovat znovu, ale za použití číselných proměnných. Důvodem je, že číselné proměnné se lépe převádějí na semafory. Poznamenejme, že se spokojíme s částečným řešením problému kritické sekce, kde není zaručen vstup.

Začneme náčrtkem problému:

loop	loop
nekritická sekce	*nekritická sekce*
vstupní protokol	*vstupní protokol*
kritická sekce	*kritická sekce*
výstupní protokol	*výstupní protokol*

Do náčrtku doplníme proměnné umožňující formulování podmínky, která má být invariantem:

let ((in1 0) (in2 0))	

loop
  nekritická sekce
  (set! in1 1)
  kritická sekce
  (set! in1 0)
	loop
	  nekritická sekce
	  (set! in2 1)
	  kritická sekce
	  (set! in2 0)

Podmínka *CS* = `(<= (+ in1 in2) 1)` vyjadřuje vzájemné vyloučení. Do programu doplníme tvrzení, které mimo jiné určují předpoklady příkazů:

let ((in1 0) (in2 0))	

loop
  nekritická sekce
  {(= in1 0)}
  (set! in1 1)
  {(= in1 1)}
  kritická sekce
  {(= in1 1)}
  (set! in1 0)
  {(= in1 0)}
	loop
	  nekritická sekce
	  {(= in2 0)}
	  (set! in2 1)
	  {(= in2 1)}
	  kritická sekce
	  {(= in2 1)}
	  (set! in2 0)
	  {(= in2 0)}

Podmínka *CS* je jistě zachována příkazy `(set! in1 0)` a `(set! in2 0)`. Příkazy `(set! in1 1)` a `(set! in2 1)` podmínku nezachovávají a z pravidla přiřazení dostaneme podmínky pro aktivní čekání:

```
{(<= in2 0)}
(set! in1 1)
{(<= (+ in1 in2) 1)}
```

a

```
{(<= in1 0)}
(set! in2 1)
{(<= (+ in1 in2) 1)}
```

Vykonání právě uvedených příkazů musíme hlídat. Přidáme do programu aktivní čekání. Všimněte si, že podmínky v aktivním čekání jsou v jiném tvaru než ty, které jsem odvodily. Důvodem je, že za chvíli bude výhodné, že obě podmínky v čekání jsou stejné.

let ((in1 0) (in2 0))	

loop
  nekritická sekce
  {(= in1 0)}
  [(await (= (+ in1 in2) 0))
  (set! in1 1)]
  {(= in1 1)}
  kritická sekce
  {(= in1 1)}
  (set! in1 0)
  {(= in1 0)}
	loop
	  nekritická sekce
	  {(= in2 0)}
	  [(await (= (+ in1 in2) 0))
	  (set! in2 1)]
	  {(= in2 1)}
	  kritická sekce
	  {(= in2 1)}
	  (set! in2 0)
	  {(= in2 0)}

Protože podmínka *CS* je nyní invariantem, můžeme ji doplnit do všech tvrzení:

let ((in1 0) (in2 0))	

loop
  nekritická sekce
  {(and (= in1 0) *CS*)}
  [(await (= (+ in1 in2) 0))
  (set! in1 1)]
  {(and (= in1 1) *CS*)}
  kritická sekce
  {(and (= in1 1) *CS*)}
  (set! in1 0)
  {(and (= in1 0) *CS*)}
	loop
	  nekritická sekce
	  {(and (= in2 0) *CS*)}
	  [(await (= (+ in1 in2) 0))
	  (set! in2 1)]
	  {(and (= in2 1) *CS*)}
	  kritická sekce
	  {(and (= in2 1) *CS*)}
	  (set! in2 0)
	  {(and (= in2 0) *CS*)}

Získali jsme hrubé řešení, které splňuje vzájemné vyloučení. Dostáváme se k samotnému převodu hrubého řešení na jemné.

Tedy máme hrubé řešení s invariantem `(<= (+ in1 in2) 1)`:

let ((in1 0) (in2 0))	

loop
  nekritická sekce
  [(await (= (+ in1 in2) 0))
  (set! in1 1)]
  kritická sekce
  (set! in1 0)
	loop
	  nekritická sekce
	  [(await (= (+ in1 in2) 0))
	  (set! in2 1)]
	  kritická sekce
	  (set! in2 0)

Zavedeme proměnné vyjadřující zdroje programu. V našem případě stačí jediná proměnné `mutex` dána podmínkou `(= mutex (- 1 (+ in1 in2)))`. Konkrétně do programu doplníme proměnnou `mutex` a zařídíme, aby podmínka, která ji zavádí, byla invariantem:

let ((in1 0) (in2 0) (mutex 1))	

loop
  nekritická sekce
  [(await (= (+ in1 in2) 0))
  (set! in1 1)
  (set! mutex (- mutex 1))]
  kritická sekce
  [(set! in1 0)
  (set! mutex (+ mutex 1))]
	loop
	  nekritická sekce
	  [(await (= (+ in1 in2) 0))
	  (set! in2 1)
	  (set! mutex (- mutex 1))]
	  kritická sekce
	  [(set! in2 0)
	  (set! mutex (+ mutex 1))]

Všimněme si, že z invariantů `(= mutex (- 1 (+ in1 in2)))` a `(<= (+ in1 in2) 1)` plyne, že i podmínka `(>= mutex 0)` je invariantem. Což je přesně invariant semaforu. V předchozím programu jsou proměnné `in1` a `in2` **synchronizační**. To znamená, že se používají k synchronizaci procesů. Vyskytují se v podmínkách aktivního čekání. Proměnná `mutex` je **pomocná**. Její přítomnost nemá vliv na chování programu. Roli proměnných můžeme zaměnit tak, že pomocné proměnné se stanou synchronizační a naopak synchronizační proměnné degradují na pomocné:

let ((in1 0) (in2 0) (mutex 1))	

loop
  nekritická sekce
  [(await (> mutex 0)))
  (set! in1 1)
  (set! mutex (- mutex 1))]
  kritická sekce
  [(set! in1 0)
  (set! mutex (+ mutex 1))]
	loop
	  nekritická sekce
	  [(await (> mutex 0)))
	  (set! in2 1)
	  (set! mutex (- mutex 1))]
	  kritická sekce
	  [(set! in2 0)
	  (set! mutex (+ mutex 1))]

Proměnné `in1` a `in2` jsou pomocné a můžeme je bez obav z programu odstranit:

let ((mutex 1))	

loop
  nekritická sekce
  [(await (> mutex 0)))
  (set! mutex (- mutex 1))]
  kritická sekce
  [(set! mutex (+ mutex 1))]
	loop
	  nekritická sekce
	  [(await (> mutex 0)))
	  (set! mutex (- mutex 1))]
	  kritická sekce
	  [(set! mutex (+ mutex 1))]

Invariant semaforu `(>= mutex 0)` zůstává platný. Nyní již v programu zůstávají příkazy realizující operace semaforu. Stačí je nahradit za operace a výchozí hodnota proměnné `mutex` se stane výchozí hodnotou semaforu:

let ((mutex (sem 1)))	

loop
  nekritická sekce
  (p mutex)
  kritická sekce
  (v mutex)
	loop
	  nekritická sekce
	  (p mutex)
	  kritická sekce
	  (v mutex)

Obdrželi jsme jemné řešení, které stále splňuje podmínku vzájemného vyloučení. Podmínky absence uváznutí a absence zbytečného čekání jsou také splněny. Uměli byste to dokázat? Jak se dozvíme za chvíli, splnění zaručení vstupu je závislé na implementaci semaforu. Výhodou řešení je, že lze použít i pro řešení problému kritické sekce pro libovolný počet procesů:

```
let ((mutex (sem 1)))
      proces i
  loop
    nekritická sekce
    (p mutex)
    kritická sekce
    (v mutex)
```

3 Blokování

Připomeňme si, že proces může být ve dvou stavech:

- **Běžící**
  Procesor vykonává instrukce procesu.
  Proces určený zásobníky: *ret*, *seek*, *rslt*, *exec* a *bnd*.
- **Připravený**
  Proces může být vykonáván.
  Proces ve frontě *rprs*.

Zavedeme nový stav, kdy proces je **blokovaný**. Kód blokovaného procesu nemůže být vykonáván. Pokud se běžící proces dostane do situace, kde nemůže dál pokračovat, zablokuje se. Jiný běžící proces může umožnit pokračování vykonávání zablokovaného procesu tím, že jej odblokuje. Kde budou uloženy blokované procesy, se dozvíme za chvíli.

Upravíme operace semaforu tak, aby mohl využít zablokování procesu. Obě operace se musí vykonat atomicky.

- **p** (počkat)
  Pokud je hodnota semaforu kladná, dekrementuj ji, jinak zablokuj běžící proces.
- **v** (vyhlásit)
  Pokud existuje aspoň jeden proces zablokovaný semaforem, jeden z nich odblokuj, jinak inkrementuj hodnotu semaforu.

4 Implementace

Semafor je pár `(value . blocked)`, kde

- *value* je nezáporné celé číslo (hodnota semaforu) a
- *blocked* je seznam procesů (procesy zablokované semaforem).

Procesy zablokované semaforem budeme ukládat do fronty. Dostaneme tak férový semafor. To znamená, že při odblokování nemůže být stále nějaký proces přehlížen.

Semafor vytvoříme slovem `:sem (val -- semaphore)`

```
(defprim :sem
  (let ((val (pop *rslt*)))
    (unless (<= 0 val)
      (error "Value can not be negative"))
    (push (cons val nil) *rslt*)))
```

Operace `p` bude realizována slovem `:p (semaphore --)`

```
(defprim :p
  (let ((sem (pop *rslt*)))
    (cond
     ((plusp (car sem))
      (decf (car sem)))
     (t
      (setf (cdr sem)
            (append (cdr sem)
                    (list (running-process))))
      (dispatcher)))))
```

Operace `v` bude realizována slovem `:v (semaphore --)`

```
(defprim :v
  (let ((sem (pop *rslt*)))
    (if (cdr sem)
        (insert-ready-process (pop (cdr sem)))
      (incf (car sem)))))
```

Ukázky

1. Hlavní proces čekající na skončení vedlejšího procesu pomocí semaforu.

```
0 :sem
:dup
(1 :print :drop :v :quit) :prsub
:p
```

2. Zablokování hlavního procesu. Ve stroji zůstane jen zaháĺčivý proces.

```
0 :sem :p
```

3. To samé jako první příklad, jen s použitím proměnných.

```
0 :sem s :bind
nil (1 :print s :val :v :quit) :prsub
s :val :p
:unbind
```

Semafor ve Scheme zpřístupníme pomocí následujících tří procedur:

```
(execute '(:sem :ret) 'sem :bind)
(execute '(:p nil :ret) 'p :bind)
(execute '(:v nil :ret) 'v :bind)
```

Ukázka:

```
(let ((s (sem 0)))
  (co
   (begin
     (print 1)
     (v s))
   (begin
     (p s)
     (print 2))))
```

Použitím semaforů můžeme změnit expanzi makra `co` tak, aby nepoužívala aktivní čekání. Konkrétně výraz:

`(co` *`body1 body2 ... bodyn`*`)`

expandujeme na:

```
(let ((start (sem 0))
      (end (sem 0)))
  (process-run-procedure (lambda (arg)
                           (p start)
                           body1
                           (v end1))
                         nil)
  ⋮
  (dotimes (i n) (v start))
  (dotimes (i n) (p end)))
```

Všimněte si, že makro `co` se obejde bez pomocného rekurzivního makra `xco`.

Co se stane?

```
(let ((s (sem 0)))
  (co
   (begin
     (p s)
     (print 1)))
  2)
```

5 Zámek

Víme, že semafory řeší problém kritické sekce. Pokud máme férové semafory, tak řešení splňuje i podmínku zaručení vstupu. Semafory tedy můžeme použít jako **zámky**. Což jsou obecně hodnoty, které se používají k ošetření kritické sekce. Zámek by se použil následovně.

```
let ((l (lock)))
    proces i
loop
  nekritická sekce
  (with-lock (l)
    kritická sekce)
```

Operátor `lock` vytvářející zámek může být procedura.

```
(define lock (lambda () (sem 1)))
```

Operátor `with-lock` by mohlo být makro, které by expandovalo následovně.

```
(with-lock (l)                (let ((lock-sym l))
  expr1                         (p lock-sym)
  ⋮                      ⇒      (let ((result (begin expr1 ... exprn)))
  exprn)                          (v lock-sym)
                                  result))
```

Pokud by pro nás zpoždění způsobené expanzí makra `with-lock` byl problém, můžeme makro přepsat na speciální operátor:

```
;; Vstup ((lock) body1 ... exprn)
(:split :swap :split :drop
 ; lock (body1 ... exprn)
 eval :val :clsub
 ; lock-val (body1 ... exprn)
 :dup :p
 ; lock-val (body1 ... exprn)
 :swap begin :swap :cons
 ; (begin body1 ... exprn) lock-val
 eval :val :clsub
 ; body-val lock-val
 :swap :v
 ; body-val
 :ret) 'with-lock :bind)
```

Ukázka použití při ošetření souběžné inkrementace sdílené proměnné:

```
(let ((n 0)
      (l (lock)))
  (co
   (dotimes (i 50)
     (with-lock (l)
       (set! n (+ n 1))))
   (dotimes (i 50)
     (with-lock (l)
       (set! n (+ n 1)))))
  n)
```

Zdrojové kódy k přednášce obsahují i kompilátor jazyka Scheme do zásobníkového jazyka nacházející se v souboru `05_compiler.lisp`. Více informací o něm, se dočtete ve zmíněném souboru.

Otázky a úkoly na cvičení

1. Definujte operátor `->` implikace. Konkrétně výraz `(-> cond1 cond2)` zapisuje podmínku: „Jestliže *cond1*, pak *cond2*.“ Použijte zkrácené vyhodnocování. Tedy například výraz `(-> nil (/ 1 0))` se vyhodnotí na *Pravdu*.

2. Synchronizujte procesy v následujícím programu tak, aby se nejprve vytiskla jednička a až poté dvojka. Synchronizaci proveďte podle postupu představeného v druhé kapitole. Tedy nejprve vypracujte hrubé řešení, které pak převedete na jemné pomocí semaforů.

```
(print 1)  |  (print 2)
```

3. Podle stejného postupu synchronizujte následující procesy tak, aby se nejprve vytiskla lichá čísla a poté až sudá čísla.

```
(print 1)  |  (print 3)
(print 2)  |  (print 4)
```

4. Rozšiřte operátor `->` tak, aby přijímal libovolný počet argumentů.

5. Definujte operátor `co-dotimes`, který funguje obdobně jako `dotimes` až na to, že těla iterací vyhodnocuje v oddělených procesech. Například:

```
(co-dotimes (i 10)
  (dotimes (j 10)
    (print i)))
```

   způsobí souběžný tisk čísel. Čísla se budou tisknout v deseti procesech a každý proces desetkrát vytiskne jedno z čísel od nuly po devítku.