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

6. Výrobci a spotřebitelé

verze z 19. března 2025

1 Odvozené příkazy

Pro zjednodušení vyjadřování si můžeme zavést příkazy odvozené ze základních příkazů pomocí maker. Například inkrementaci a dekrementaci hodnoty proměnné zavedeme makry:

- `(inc!` *`var`*`) => (set!` *`var`* `(+` *`var`* `1))`
- `(dec!` *`var`*`) => (set!` *`var`* `(-` *`var`* `1))`

Jistě oba právě zavedené příkazy nebudou atomické na sdílených proměnných. Pravidla pro příkazy získáme z pravidla přiřazení:

{*condition2*}	{*condition3*}
(inc! *var*)	(dec! *var*)
{*condition*}	{*condition*}

kde *condition2* (resp. *condition3*) vznikne z *condition* nahrazením všech výskytů proměnné *var* za `(+` *`var`* `1)` (resp. `(-` *`var`* `1)`).

2 Zadání

V této přednášce se budeme zabývat komunikací mezi procesy, kde procesy posílají data jiným procesům. Obecně rozdělíme procesy do dvou skupin: na **výrobce** a **spotřebitele**. Výrobci produkují data, nazývané **zprávy**, a posílají je spotřebitelům, kteří obdržené zprávy zpracovávají. Začneme zjednodušenou variantou s jedním výrobcem a jedním spotřebitelem. Způsob komunikace vyjádříme následujícím náčrtkem.

`let ((buffer nil))`	
výrobce	*spotřebitel*

`let ((message nil))`
`loop`
`  `*`vytvoř zprávu`* `message`
`  (set! buffer message)`
	`let ((message nil))`
	`loop`
	`  (set! message buffer)`
	`  `*`zpracuj zprávu`* `message`

Na řešení klademe požadavky:

- Žádná zpráva nemůže být zpracována vícekrát.
- Každá zpráva musí být zpracována.

3 Hrubé řešení

Nejprve vypracujeme hrubé řešení používající složené atomické příkazy. Pro formalizaci prvního požadavku doplníme do náčrtku proměnné `da` (*deposit after*) a `fi` (*fetch in*), kde význam první proměnné je počet dokončení ukládání zprávy výrobcem a význam druhé proměnné je počet zahájení vyzvedávání zprávy spotřebitelem:

`let ((buffer nil)) (da 0) (fi 0))`	
výrobce	*spotřebitel*

`let ((message nil))`
`loop`
`vytvoř zprávu message`
`(set! buffer message)`
`(inc! da)`
	`let ((message nil))`
	`loop`
	`(inc! fi)`
	`(set! message buffer)`
	`zpracuj zprávu message`

Nyní můžeme požadovat, aby podmínka `(<= fi da)` byla invariantem programu. Podmínka je splněna ve výchozím stavu programu: `(<= 0 0)`. Ověříme, že příkazy měnící proměnné podmínky ji zachovávají. Zachování podmínky příkazem `(inc! da)` vede na pravdivost programu:

```
{(<= fi da)}
(inc! da)
{(<= fi da)}
```

Program je pravdivý. Stačí použít pravidlo přiřazení:

```
{(<= fi (+ da 1))}
(inc! da)
{(<= fi da)}
```

a zesílit předpoklad:

```
{(<= fi da)}
(inc! da)
{(<= fi da)}
```

Příkaz `(inc! fi)` podmínku nezachovává, protože program:

```
{(<= fi da)}
(inc! fi)
{(<= fi da)}
```

pravdivý není. Stačí jej spustit ve stavu, kde `fi` a `da` jsou rovny nule. Vykonání příkazu musíme hlídat. Proto se aktivnímu čekání někdy říká **hlídka**. Podmínkou hlídky bude předpoklad programu, který vznikne použitím pravidla přiřazení:

```
{(<= (+ fi 1) da)}
(inc! fi)
{(<= fi da)}
```

Podmínku hlídky můžeme zjednodušit na `(< fi da)`. Hlídku doplníme do náčrtku:

`let ((buffer nil) (da 0) (fi 0))`	
výrobce	*spotřebitel*

`let ((message nil))`
` loop`
`  `*vytvoř zprávu*` message`
`  (set! buffer message)`
`  (inc! da)`
	`let ((message nil))`
	` loop`
	`  (await (< fi da))`
	`  (inc! fi)`
	`  (set! message buffer)`
	`  `*zpracuj zprávu*` message`

Program nyní vyhovuje prvnímu požadavku. Podobně pro vyjádření druhého požadavku nejprve do programu doplníme proměnné `di` (*deposit in*) a `fa` (*fetch after*) tak, aby hodnota první byla počtem zahájení ukládání zprávy výrobcem a hodnota druhé byla počtem dokončení vyzvedávání zprávy spotřebitelem:

`let ((buffer nil) (da 0) (fi 0) (di 0) (fa 0))`	
výrobce	*spotřebitel*

`let ((message nil))`
` loop`
`  `*vytvoř zprávu*` message`
`  (inc! di)`
`  (set! buffer message)`
`  (inc! da)`
	`let ((message nil))`
	` loop`
	`  (await (< fi da))`
	`  (inc! fi)`
	`  (set! message buffer)`
	`  (inc! fa)`
	`  `*zpracuj zprávu*` message`

Druhý požadavek nyní vyjádříme podmínkou `(<= di (+ fa 1))`, po které chceme, aby byla invariantem programu. Podmínka je jistě splněna ve výchozím stavu. Příkaz `(inc! di)` ji nezachovává. Odvodíme podmínku hlídky:

```
{(<= di fa)}
(inc! di)
{(<= di (+ fa 1))}
```

Příkaz `(inc! fa)` podmínku zachovává. Doplníme do programu hlídky:

let ((buffer nil) (da 0) (fi 0) (di 0) (fa 0))	
výrobce	*spotřebitel*
let ((message nil)) loop *vytvoř zprávu* message (await (<= di fa)) (inc! di) (set! buffer message) (inc! da)	let ((message nil)) loop (await (< fi da)) (inc! fi) (set! message buffer) (inc! fa) *zpracuj zprávu* message

Dostáváme hrubé řešení pro jednoho výrobce a jednoho spotřebitele. Zajištěním atomičnosti kritických příkazů, získáme řešení pro libovolný počet výrobců a spotřebitelů:

let ((buffer nil) (da 0) (fi 0) (di 0) (fa 0))	
výrobci	*spotřebitelé*
let ((message nil)) loop *vytvoř zprávu* message [(await (<= di fa)) (inc! di)] (set! buffer message) [(inc! da)]	let ((message nil)) loop [(await (< fi da)) (inc! fi)] (set! message buffer) [(inc! fa)] *zpracuj zprávu* message

Splnění požadavků zajišťuje invariant `(and (<= fi da) (<= di (+ fa 1)))`.

4 Přes počítání zdrojů k jemnému řešení

Pro přechod na jemné řešení pomocí semaforů, zavedeme pomocné proměnné počítající zdroje. Z pohledu výrobce je zdrojem prázdné úložiště, což vyjádříme proměnnou `empty` danou rovností `(= empty (+ (- fa di) 1))`. Spotřebitel má za zdroj plné úložiště vyjádřené proměnnou `full` danou vztahem `(= full (- da fi))`. Doplníme proměnné `empty` a `full` do programu tak, aby se rovnosti, které je určují, staly invarianty programu:

```
let ((buffer nil) (da 0) (fi 0) (di 0) (fa 0))
     (empty 1) (full 0))
```

výrobci	*spotřebitelé*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(await (< fi da))`
`[(await (<= di fa))`	`(inc! fi)]`
`(inc! di)]`	`(dec! full)]`
`(dec! empty)]`	`(set! message buffer)`
`(set! buffer message)`	`[(inc! fa)]`
`[(inc! da)]`	`(inc! empty)]`
`(inc! full)]`	*zpracuj zprávu* `message`

Z dokázaných invariantů programu můžeme snadno odvodit, že i podmínky `(<= 0 empty)` a `(<= 0 full)` jsou invarianty. To je dobré znamení, protože se jedná o invarianty semaforu. Převedeme řízení na proměnné `empty` a `full`:

```
let ((buffer nil) (da 0) (fi 0) (di 0) (fa 0)
     (empty 1) (full 0))
```

výrobci	*spotřebitelé*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(await (< 0 full))`
`[(await (< 0 empty))`	`(inc! fi)`
`(inc! di)`	`(dec! full)]`
`(dec! empty)]`	`(set! message buffer)`
`(set! buffer message)`	`[(inc! fa)`
`[(inc! da)`	`(inc! empty)]`
`(inc! full)]`	*zpracuj zprávu* `message`

Proměnné `da`, `fi`, `di` a `fa` jsou nyní pomocné a můžeme je bezpečně odstranit:

```
let ((buffer nil)
     (empty 1) (full 0))
```

výrobci	*spotřebitelé*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(await (< 0 full))`
`[(await (< 0 empty))`	`(dec! full)]`
`(dec! empty)]`	`(set! message buffer)`
`(set! buffer message)`	`[(inc! empty)]`
`[(inc! full)]`	*zpracuj zprávu* `message`

K dosažení jemného řešení stačí nahradit hodnoty proměnných `empty` a `full` za semafory a nahradit složené atomické příkazy za operace `p` a `v` semaforu:

let ((buffer nil) (empty (sem 1)) (full (sem 0)))	
výrobci	*spotřebitelé*
let ((message nil)) loop *vytvoř zprávu* message (p empty) (set! buffer message) (v full)	let ((message nil)) loop (p full) (set! message buffer) (v empty) *zpracuj zprávu* message

5 Fronta

Pokud rychlost vytváření zpráv výrobci a rychlost jejich zpracovávání spotřebiteli je přibližně stejná, může k plynulosti komunikace přispět zavedení fronty zpráv, do které se budou ukládat vytvořené dosud nezpracované zprávy.

Fontu implementujeme pomocí **vektoru**, což je v Lispu a tedy i v našem Scheme jednorozměrné pole. Vektor vytvoříme makrem `vect`. Konkrétně výraz `(vect` *`length expr`*`)` se vyhodnotí na nový vektor délky *`length`*, kde každý prvek vznikne samostatným vyhodnocením výrazu *`expr`*. Základní procedury pro práci s vektory jsou:

- `(vref` *`vector index`*`) =>` *`element`*
- `(vset!` *`vector index value`*`) =>` *`value`*

Procedura `vref` vrací hodnotu vektoru na indexu a procedura `vset!` nastavuje prvek vektoru na indexu. Obě procedury se vykonají atomicky. Pro další detaily se podívejte do zdrojového kódu k přednášce.

Fronta pak bude datová struktura s položkami:

- `buffer` ... vektor prvků velikosti `n`
- `front` ... index prvního prvku
- `rear` ... index za posledním prvkem

Nastavení a čtení položek struktury je atomické.

Frontu vytvoříme konstruktorem `make-queue`, kterému zadáme délku vektoru. Výchozí hodnota položek `front` a `rear` je nula. Na frontě zavedeme dvě operace, které nebudou atomické:

1. `(enqueue queue message)` Nastaví prvek vektoru `buffer` na indexu `rear` na `message` a posune `rear` na další index případně na nulu.
2. `(dequeue queue)` Posune `front` na další index případně na nulu a vrátí prvek vektoru `buffer` na původní hodnotě indexu `front`.

6 Vyrovnávací paměť

Proměnná `n` udává velikost vyrovnávací paměti. Do náčrtku řešení přidáme pomocné proměnné:

```
let ((queue (make-queue n)))
     (di 0) (da 0) (fi 0) (fa 0))
```

výrobce	*spotřebitel*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(inc! fi)]`
`[(inc! di)]`	`(set! message (dequeue queue))`
`(enqueue queue message)`	`[(inc! fa)]`
`[(inc! da)]`	*zpracuj zprávu* `message`

Aby program splňoval výše uvedené požadavky, které klademe na řešení problému výrobce a spotřebitele, musí být podmínka `(and (<= di (+ fa n)) (<= fi da))` invariantem. Přidáme k příkazům hlídky, které to zajistí:

```
let ((queue (make-queue n))
     (di 0) (da 0) (fi 0) (fa 0))
```

výrobce	*spotřebitel*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(await (< fi da))`
`[(await (< di (+ fa n)))`	`(inc! fi)]`
`(inc! di)]`	`(set! message (dequeue queue))`
`(enqueue queue message)`	`[(inc! fa)]`
`[(inc! da)]`	*zpracuj zprávu* `message`

Podobně jako v předchozím případě záměnou proměnných podle `(= empty (+ (- fa di) n))` a `(= full (- da fi))` dostaneme řešení pomocí zdrojů:

```
let ((queue (make-queue n))
     (empty n) (full 0))
```

výrobce	*spotřebitel*
`let ((message nil))`	`let ((message nil))`
`loop`	`loop`
vytvoř zprávu `message`	`[(await (< 0 full))`
`[(await (< 0 empty))`	`(dec! full)]`
`(dec! empty)]`	`(set! message (dequeue queue))`
`(enqueue queue message)`	`[(inc! empty)]`
`[(inc! full)]`	*zpracuj zprávu* `message`

Obdržené řešení už přímočaře předěláme na řešení pomocí semaforů:

```
let ((queue (make-queue n))
     (empty (sem n)) (full (sem 0)))
```

výrobce	*spotřebitel*

`let ((message nil))`
`loop`
`  vytvoř zprávu message`
`  (p empty)`
`  (enqueue queue message)`
`  (v full)`
	`let ((message nil))`
	`loop`
	`  (p full)`
	`  (set! message (dequeue queue))`
	`  (v empty)`
	`  zpracuj zprávu message`

Pro rozšíření řešení na libovolný počet výrobců a spotřebitelů je nutné zavést dva zámky pro operace `enqueue` a `dequeue`, protože operace nejsou atomické.

```
let ((queue (make-queue n))
     (empty (sem n)) (full (sem 0))
     (deposit-lock (lock)) (fetch-lock (lock)))
```

výrobci	*spotřebitelé*

`let ((message nil))`
`loop`
`  vytvoř zprávu message`
`  (p empty)`
`  (with-lock (deposit-lock)`
`  (enqueue queue message))`
`  (v full)`
	`let ((message nil))`
	`loop`
	`  (p full)`
	`  (with-lock (fetch-lock)`
	`  (set! message (dequeue queue))`
	`  (v empty)`
	`  zpracuj zprávu message`

Otázky a úkoly na cvičení

1. Vytvořte dva procesy: výrobce a spotřebitele, kde výrovce bude zasílat spotřebiteli postupně čísla od nuly po devadesát devět včetně. Spotřebitel obdržená čísla tiskne. V řešení nepoužívejte vyrovnávací paměť.

2. Upravte předchozí řešení tak, aby spotřebitel nevěděl, kolik čísel od výrobce obdrží.

3. Rozšiřte řešení na větší počet výrobců a spotřebitelů.

4. Implementujte frontu pomocí pole, jak je uvedeno výše.

5. Vyřešte znovu první úkol, ale tentokrát s použitím vyrovnávací paměti.

6. Opět rozšiřte řešení na více výrobců a spotřebitelů.

7. Zapouzdřete řešení výrobců a spotřebitelů s použitím vyrovnávací paměti do datové struktury `sync-queue`, která by se používala následovně.

`let ((queue (make-sync-queue n)))`	
výrobci	*spotřebitelé*

`let ((message nil))`
`  loop`
`  `*vytvoř zprávu*` message`
`  (sync-enqueue queue message)`
	`let ((message nil))`
	`  loop`
	`  (set! message (sync-dequeue queue))`
	`  `*zpracuj zprávu*` message`

8. Upravte řešení z předchozího úkolu tak, aby fronta byla reprezentována seznamem. Vyrovnávací paměť tak bude mít potencionálně nekonečnou velikost. Proto přidání do fronty nebude nikdy blokující.

9. Upravte odebírání z fronty tak, aby nebylo blokující. V případě prázdné fronty vraťte `nil`.

10. Vytvořte datovou strukturu `channel` (kanál), která bude zapouzdřovat řešení výrobců a spotřebitelů tak, že výrobce bude blokován, dokud si spotřebitel zprávu nevyzvedne.

11. Přidejte do Scheme operátor `<--` pro snadnější komunikaci pomocí kanálů. Použití:

`let ((channel (make-channel)))`	
výrobci	*spotřebitelé*

`let ((message nil))`
`  loop`
`  `*vytvoř zprávu*` message`
`  (<-- channel message)`
	`let ((message nil))`
	`  loop`
	`  (set! message (<-- channel))`
	`  `*zpracuj zprávu*` message`