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

7. Technika štafetového kolíku

verze z 25. března 2025

1 Čtenáři a písaři

Techniku štafetového kolíku si představíme na problému *čtenářů a písařů*. Jedná se o zobecnění problému kritické sekce. Procesy jsou rozděleny do dvou skupin: čtenáři a písaři. V chráněné části může být pouze jeden písař a žádný čtenář, nebo žádný písař a libovolný počet čtenářů.

čtenář	*písař*

loop
  nekritická sekce
  vstupní protokol čtenáře
  čte
  výstupní protokol čtenáře
	loop
	  nekritická sekce
	  vstupní protokol písaře
	  píše
	  výstupní protokol písaře

Hrubé řešení dostaneme, když budeme počítat, kolik čtenářů a písařů je v kritické sekci:

let ((nr 0) (nw 0))	
čtenář	*písař*

loop
  [(await (= nw 0))
  (inc! nr)]
  čte
  [(dec! nr)]
	loop
	  [(await (and (= nr 0) (= nw 0)))
	  (inc! nw)]
	  píše
	  [(dec! nw)]

Správnost řešení zaručí invariant `(and (or (= nr 0) (= nw 0)) (<= nw 1))`.

2 Štafetový kolík

Štafetový kolík je semafor, který má na začátku hodnotu jedna.

```
(define make-baton
        (lambda ()
          (sem 1)))
```

O štafetový kolík se uchází **skupiny**. Skupina je datová struktura `baton-group` (`bg`) s položkami *`baton`*, *`waiting-count`* a *`semaphore`*, kde *`baton`* je štafetový kolík a *`waiting-count`* je počet procesů ve skupině, kteří čekají na semaforu *`semaphore`*. Skupinu vytvoříme konstruktorem `make-baton-group`, kterému zadáme štafetový kolík. Položka *`waiting-count`* má výchozí hodnotu nula a položka `semaphore` je semafor s výchozí hodnotou nula. Pro položky struktury máme selektory `get-bg-baton`, `get-bg-count` a `get-bg-semaphore` a pomocné operace `bg-inc!` a `bg-dec!` na inkrementaci a dekrementaci položky *`waiting-count`*.

Pokud proces nemůže postoupit dál, vzdá se štafetového kolíku a začne čekat.

```
(define baton-wait
        (lambda (bg)
          (bg-inc! bg)
          (v (get-bg-baton bg))
          (p (get-bg-semaphore bg))))
```

Můžeme zjistit, zda někdo ze skupiny čeká na kolík:

```
(define baton-waiting-p
        (lambda (bg)
          (< 0 (get-bg-count bg))))
```

Pokud víme, že někdo ze skupiny čeká na kolík, můžeme mu kolík předat.

```
(define baton-signal
        (lambda (bg)
          (bg-dec! bg)
          (v (get-bg-semaphore bg))))
```

Následujícím postupem můžeme libovolné hrubé řešení převést na jemné. Nejprve si vytvoříme štafetový kolík *`baton`*.

Poté vezmeme všechny podmíněné atomické operace sdílející proměnné:

```
[(await condition1) . body1]
[(await condition2) . body2]
⋮
[(await conditionn) . bodyn]
```

pro každou z nich vytvoříme skupinu *`group1`*, *`group2`*, ..., *`groupn`*.

V případě čtenářů a písařů máme dvě podmíněné atomické operace.

```
[(await (= nw 0))
 (inc! nr)]
[(await (and (= nr 0) (= nw 0)))  (inc! nw)]
```

Vytvoříme pro ně skupiny *readers* a *writers*.

Každou podmíněnou atomickou operaci [(await *condition*) *expr1* ... *exprn*] pro skupinu *group* nahradíme za

```
(begin
  (p baton)
  (unless condition (baton-wait group)
  expr1
  ⋮
  exprn
  signal)
```

kde *signal* je

```
(cond
  ((and condition1 (baton-waiting-p group1))
   (baton-signal group1))
  ((and condition2 (baton-waiting-p group2))
   (baton-signal group2))
  ⋮
  ((and conditionn (baton-waiting-p groupn))
   (baton-signal groupn))
  (t
   (v baton)))
```

V nahrazeném kódu nejprve proces čeká na štafetový kolík:

```
(p baton)
```

Pak kontroluje podmínku atomické operace a v případě, že není splněna, kolíku se vzdává:

```
(unless condition (baton-wait group)
```

Poté atomicky vyhodnotí výrazy v atomické operaci:

```
expr1
⋮
exprn
```

Na závěr se vzdá kolíku:

```
signal
```

V této části se zjišťuje, zda existuje proces, který by čekal na kolík a jehož podmínka by byla splněna. V kladném případě se jednomu z nich kolík předá. Jinak proces kolík vrací:

```
(v baton)
```

Podle uvedeného postupu podmíněný atomický složený příkaz:

```
[(await (= nw 0))
 (inc! nr)]
```

nahradíme za:

```
(begin
  (p baton)
  (unless (= 0 nw)
    (baton-wait readers))
  (inc! nr)
  reader-writer-signal)
```

kde *reader-writer-signal* je:

```
(cond
  ((and (= nw 0) (baton-waiting-p readers))
   (baton-signal readers))
  ((and (= nr 0) (= nw 0) (baton-waiting-p writers))
   (baton-signal writers))
  (t
   (v baton)))
```

Nepodmíněný atomický složený příkaz:

```
[expr1 ... exprn]
```

nahradíme za:

```
(begin
  (p baton)
  expr1
```

```
  ⋮
  exprn
  signal)
```

Například:

```
[(dec! nr)]
```

přejde na:

```
(begin
  (p baton)
  (dec! nr)
  reader-writer-signal)
```

Nyní přejdeme na schéma řešení čtenářů a písařů.

```
let ((nr 0) (nw 0) (baton (make-baton)))
  let ((readers (make-baton-group baton))
       (writers (make-baton-group baton)))
```

čtenář	*písař*
loop	loop
reader-enter	*writer-enter*
čte	*píše*
reader-leave	*writer-leave*

Zbývá doplnit vstupní a výstupní protokoly.

reader-writer-signal:

```
(cond
  ((and (= nw 0) (baton-waiting-p readers))
   (baton-signal readers))
  ((and (= nr 0) (= nw 0) (baton-waiting-p writers))
   (baton-signal writers))
  (t
   (v baton))))
```

reader-enter:

```
(p baton)
(unless (= 0 nw)
  (baton-wait readers))
(inc! nr)
reader-writer-signal
```

reader-leave:

```
(p baton)
(dec! nr)
reader-writer-signal
```

writer-enter:

```
(p baton)
(unless (and (= nr 0)
             (= nw 0))
(baton-wait writers))
(inc! nw)
reader-writer-signal
```

writer-leave:

```
(p baton)
(dec! nw)
reader-writer-signal
```

Všimněte si, že některé části vstupních a výstupních protokolů jsou zbytečné. Kód můžeme zjednodušit následovně.
reader-enter:

```
(p baton)
(when (< 0 nw)
  (baton-wait readers))
(inc! nr)
(if (baton-waiting-p readers)
    (baton-signal readers)
  (v baton))
```

reader-leave:

```
(p baton)
(dec! nr)
(if (and (= nr 0)
         (baton-waiting-p writers))
    (baton-signal writers)
  (v baton))
```

writer-enter:

```
(p baton)
(when (or (> nr 0)
          (> nw 0))
  (baton-wait writers))
(inc! nw)
(v baton)
```

writer-leave:

```
(p baton)
(dec! nw)
(cond
 ((baton-waiting-p readers)
  (baton-signal readers))
 ((baton-waiting-p writers)
  (baton-signal writers))
 (t
  (v baton)))
```

Výhodou techniky štafetového kolíku je, že nemusíme přecházet na řízení pomocí zdrojů a můžeme k synchronizaci využít proměnné z hrubého řešení. Také máme pod kontrolou probouzení procesů z jednotlivých skupin. Můžeme tedy snadno řešení změnit tak, aby upřednostňovalo písaře.

3 Klasické řešení čtenářů a písařů

Vraťme se k náčrtku problému čtenářů a písařů:

čtenář	*písař*
loop	loop
nekritická sekce	*nekritická sekce*
vstupní protokol čtenáře	*vstupní protokol písaře*
čte	*píše*
výstupní protokol čtenáře	*výstupní protokol písaře*

Zavedeme si tři proměnné. Proměnná `reading` udává, zda nějaký čtenář čte, proměnná `writing` uchovává vektor, kde hodnota na indexu `i` určuje, zda písař `i` zapisuje, a proměnná `nr` udržuje počet čtenářů, kteří čtou.

```
let ((reading 0) (nr 0) (writing (vect n 0)))
```

čtenář	*písař* i

loop
  [(inc! nr)
  (when (= nr 1)
  (inc! reading))]
  čte
  [(dec! nr)
  (when (= nr 0)
  (dec! reading))]
	loop
	  [(vinc! writing i)]
	  píše
	  [(vdec! writing i)]

Funkce `vinc!` inkrementuje a `vdec!` dekrementuje prvek vektoru na zadaném indexu. Správnost řešení lze nyní vyjádřit podmínkou `(<= (+ reading (vector-sum writing)) 1)`, kde procedura `vector-sum` počítá součet prvků vektoru.

Výraz `(+ reading (vector-sum writing))` si označíme *SUM*. Dostáváme řešení nahrubo.

```
let ((reading 0) (nr 0) (writing (vect n 0)))
```

čtenář	*písař* i

loop
  [(await (or (> nr 0) (<= *SUM* 0)))
  (inc! nr)
  (when (= nr 1)
  (inc! reading))]
  čte
  [(dec! nr)
  (when (= nr 0)
  (dec! reading))]
	loop
	  [await (<= *SUM* 0))
	  (vinc! writing i)]
	  píše
	  [(vdec! writing i)]

Složené atomické operace čtenáře můžeme nahradit zámkem.

```
let ((reading 0) (nr 0) (writing (vect n 0)) (mutex-r (lock)))
```

čtenář	*písař*

loop
  (with-lock (mutex-r)
  (inc! nr)
  (when (= nr 1)
  [(await (<= *SUM* 0))
  (inc! reading)]))
  čte
  (with-lock (mutex-r)
  (dec! nr)
  (when (= nr 0)
  [(dec! reading)]))
	loop
	  [await (<= *SUM* 0))
	  (vinc! writing i)]
	  píše
	  [(vdec! writing i)]

Uvažujeme proměnnou `rw` danou podmínkou `(= rw (- 1 (+ reading (vector-sum writing))))`. Proměnná udává zdroj, o který se uchází čtenáři jako skupina nebo libovolný písař.

Záměnou proměnných dostáváme řešení řízené pomocí jednoho zdroje.

`let ((nr 0) (rw 1) (mutex-r (lock)))`	
čtenář	*písař*

`loop`
`  (with-lock (mutex-r)`
`  (inc! nr)`
`  (when (= nr 1)`
`  [(await (< 0 rw))`
`  (dec! rw)]))`
čte
`  (with-lock (mutex-r)`
`  (dec! nr)`
`  (when (= nr 0)`
`  [(inc! rw)]))`
	`loop`
	`  [(await (< 0 rw))`
	`  (dec! rw)]`
	píše
	`  [(inc! rw)]`

Zdroj stačí nahradit za semafor.

`let ((nr 0) (rw (sem 1)) (mutex-r (lock)))`	
čtenář	*písař*

`loop`
`  (with-lock (mutex-r)`
`  (inc! nr)`
`  (when (= nr 1)`
`  (p rw)))`
čte
`  (with-lock (mutex-r)`
`  (dec! nr)`
`  (when (= nr 0)`
`  (v rw)))`
	`loop`
	`  (p rw)`
	píše
	`  (v rw)`

Otázky a úkoly na cvičení

1. Vyjděte z prvního řešení problému čtenářů a písařů obdrženého pomocí techniky štafetového kolíku a zapouzdřete jej do datové struktury `rw`. Na struktuře budou dány čtyři atomické operace: `reader-enter1`, `reader-leave1`, `writer-enter1` a `writer-leave1`. Způsob použití:

let ((rw (make-rw)))	
čtenář	*písař*

loop
  nekritická sekce
  (reader-enter1 rw)
  čte
  (reader-leave1 rw)
	loop
	  nekritická sekce
	  (writer-enter1 rw)
	  píše
	  (writer-leave1 rw)

2. Implementujte pro strukturu z předchozího úkolu operace `reader-enter`, `reader-leave`, `writer-enter` a `writer-leave` zjednodušující vstupní a výstupní protokoly popsané v textu.

3. Upravte řešení tak, aby písaři měli zaručený vstup.

4. Upravte řešení tak, aby byl zaručen vstup jak pro písaře, tak pro čtenáře.

5. Přidejte makra `with-reader` a `with-writer` zjednodušující použití řešení:

let ((rw (make-rw)))	
čtenář	*písař*

loop
  nekritická sekce
  (with-reader (rw)
  čte)
	loop
	  nekritická sekce
	  (with-writer (rw)
	  píše)

6. Napište paralelní program, který se bude skládat z procesů rozdělených na čtenáře a písaře. Program má na vstupu vektor, kde všechny prvky mají stejnou hodnotu, například `#(1 1 1 1)`. Čtenář v cyklu tiskne prvky vektoru. Pro každý prvek vytiskne seznam `(`*i j v*`)`, kde *i* je číslo čtenáře, *j* je tisknutý index a *v* je hodnota na indexu *j*. Například pro počáteční vektor tiskař *2* vytiskne:

   ```
   (2 0 1)
   (2 1 1)
   (2 2 1)
   (2 3 1)
   ```

   Písař cyklicky změní všechny prvky vektoru na své číslo. Například písař 2 změní vektor na `#(2 2 2 2)`. Invarinat programu je, že prvky vektoru se musí vždy rovnat.