Paradigmata programování 4

Přednáška 8. Bariéry

verze z 1. dubna 2025

Jan Laštovička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Obsah

Součet prefixů

Součet prefixů

Je dán vektor `v1` velikosti `n`.

Například:

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Součet prefixů

Je dán vektor `v1` velikosti `n`.

Například:

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Zavedeme:

- (`vsum` v m n) je součet prvků vektoru v od indexu m po n (včetně).

Součet prefixů

Je dán vektor `v1` velikosti `n`.

Například:

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Zavedeme:

- `(vsum` v m n`)` je součet prvků vektoru v od indexu m po n (včetně).

Chceme vyrobit vektor `sum` délky `n`,

- kde `(= (vref sum i) (vsum v1 0 i))` pro každé `i`.

Pro příklad `v1` je `sum`:

0	1	2	3	4	5	6	7
1	3	6	10	15	21	28	36

Součet prefixů

Je dán vektor `v1` velikosti `n`.

Například:

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Zavedeme:

- `(vsum` v m n`)` je součet prvků vektoru v od indexu m po n (včetně).

Chceme vyrobit vektor `sum` délky `n`,

- kde `(= (vref sum i) (vsum v1 0 i))` pro každé `i`.

Pro příklad `v1` je `sum`:

0	1	2	3	4	5	6	7
1	3	6	10	15	21	28	36

Sekvenční průchod má lineární časovou složitost.

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)))
```

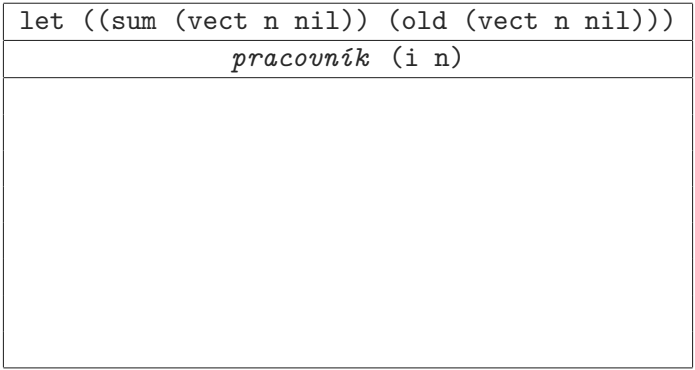
Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
```

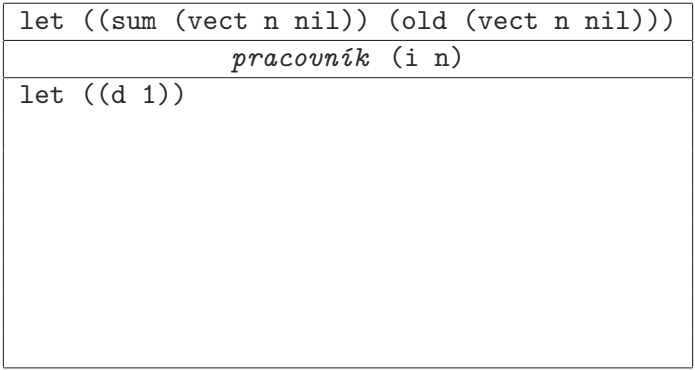
Paralelní součet prefixů

- `v1` ... vektor velikosti `n`



Paralelní součet prefixů

- `v1` ... vektor velikosti `n`



Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
```

pracovník `(i n)`

```
let ((d 1))
  (vset! sum i (vref v1 i))
```

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (while (< d n)
```

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (while (< d n)
    (vset! old i (vref sum i))
```

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (while (< d n)
    (vset! old i (vref sum i))
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
```

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (while (< d n)
    (vset! old i (vref sum i))
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (set! d (* d 2)))
```

Paralelní součet prefixů

- v1 ... vektor velikosti n

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (while (< d n)
    (vset! old i (vref sum i))
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (set! d (* d 2)))
```

... špatně

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (barrier)
  (while (< d n)
    (vset! old i (vref sum i))
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (set! d (* d 2)))
```

... špatně

Paralelní součet prefixů

- `v1` ... vektor velikosti `n`

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (barrier)
  (while (< d n)
    (vset! old i (vref sum i))
    (barrier)
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (set! d (* d 2)))
```

... špatně

Paralelní součet prefixů

- v1 ... vektor velikosti n

```
let ((sum (vect n nil)) (old (vect n nil)))
                pracovník (i n)
let ((d 1))
  (vset! sum i (vref v1 i))
  (barrier)
  (while (< d n)
    (vset! old i (vref sum i))
    (barrier)
    (when (<= 0 (- i d))
      (vset! sum i (+ (vref old (- i d))
                      (vref sum i))))
    (barrier)
    (set! d (* d 2)))
```

Paralelní součet prefixů

Paralelní součet prefixů

Počet kroků:

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Invariant cyklu: `(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))`

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Invariant cyklu: `(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))`

Simulace:

d	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Invariant cyklu: `(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))`

Simulace:

d	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8
1	1	3	5	7	9	11	13	15

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Invariant cyklu: `(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))`

Simulace:

d	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8
1	1	3	5	7	9	11	13	15
2	1	3	6	10	14	18	22	26

Paralelní součet prefixů

Počet kroků: `(ceiling (log n 2))`

Invariant cyklu: `(= (vref sum i) (vsum v1 (max (+ (- i d) 1) 0) i))`

Simulace:

d	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8
1	1	3	5	7	9	11	13	15
2	1	3	6	10	14	18	22	26
4	1	3	6	10	15	21	28	36

Bariéry

Bariéry

```
loop
  (co-dotimes (i n)
    úkol pro i)
```

Bariéry

```
loop
  (co-dotimes (i n)
    úkol pro i)
```

... neefektivní

Bariéry

```
loop
  (co-dotimes (i n)
    úkol pro i)
```

... neefektivní

```
        pracovník (i n)
loop
  úkol pro i
  čekej, až všech n dokončí úkol
```

Bariéry

```
loop
  (co-dotimes (i n)
    úkol pro i)
```

... neefektivní

```
        pracovník (i n)
loop
  úkol pro i
  čekej, až všech n dokončí úkol
```

... synchronizace pomocí bariéry

Obsah

Sdílené počítadlo

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

```
let ((count 0) (passed (vect n nil)))
        pracovník (i n)
úkol pro i
```

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

```
let ((count 0) (passed (vect n nil)))
                pracovník (i n)
úkol pro i
[(inc! count)]
```

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

```
let ((count 0) (passed (vect n nil)))
```

pracovník (i n)

úkol pro i
[(inc! count)]
[(await (= count n))
 (vset! passed i t)]

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

```
let ((count 0) (passed (vect n nil)))
```

pracovník (i n)

úkol pro i
[(inc! count)]
[(await (= count n))
 (vset! passed i t)]

Je znovupoužitelná?

Sdílené počítadlo

- `n` ... počet pracovníků
- `count` ... počet pracovníků, kteří dorazili k bariéře
- `(vref passed i)` ... pracovník `i` prošel bariérou

Invarianty: `(-> (vref passed i) (= count n))`

```
let ((count 0) (passed (vect n nil)))
```

pracovník (i n)

úkol pro i
[(inc! count)]
[(await (= count n))
 (vset! passed i t)]

Je znovupoužitelná? Není.

Fetch-and-add

Fetch-and-add

- *var* ... proměnná
- *num* ... číslo

Fetch-and-add

- *var* ... proměnná
- *num* ... číslo

Složená atomická operace: `(fa` *var num*`)`

Fetch-and-add

- *var* ... proměnná
- *num* ... číslo

Složená atomická operace: `(fa` *var num*`)`

Atomicky:

1. Zjistí hodnotu *val* proměnné *var*,
2. zvýší hodnotu proměnné *var* o *num*
3. a vrátí hodnotu *val*.

Fetch-and-add

- *var* ... proměnná
- *num* ... číslo

Složená atomická operace: `(fa` *var num*`)`

Atomicky:

1. Zjistí hodnotu *val* proměnné *var*,
2. zvýší hodnotu proměnné *var* o *num*
3. a vrátí hodnotu *val*.

Implementace bariéry:

Fetch-and-add

- *var* ... proměnná
- *num* ... číslo

Složená atomická operace: `(fa` *var num*`)`

Atomicky:

1. Zjistí hodnotu *val* proměnné *var*,
2. zvýší hodnotu proměnné *var* o *num*
3. a vrátí hodnotu *val*.

Implementace bariéry:

```
(fa count 1)
(await (= count n))
```

Znovupoužitelná bariéra

```
let ((count 0))
    pracovník (i n)
  loop
    úkol pro i
    [(inc! count)]
    (await (= count n))
```

Znovupoužitelná bariéra

```
let ((count 0))
    pracovník (i n)
  loop
    úkol pro i
    [(inc! count)]
    (await (= count n)
    (set! count 0)
```

Znovupoužitelná bariéra

```
let ((count 0))
    pracovník (i n)
  loop
    úkol pro i
    [(inc! count)]
    (await (= count n)
    (set! count 0)
```

... špatně

Znovupoužitelná bariéra

```
let ((count 0))
    pracovník (i n)
 loop
   úkol pro i
   [(inc! count)]
   (await (= count n))
   [(dec! count)]
   (await (= count 0))
```

Znovupoužitelná bariéra

```
let ((count 0))
    pracovník (i n)
  loop
    úkol pro i
    [(inc! count)]
    (await (= count n))
    [(dec! count)]
    (await (= count 0))
```

... špatně

Znovupoužitelná bariéra

```
let ((count1 0) (count2 n))
      pracovník (i n)
  loop
    úkol pro i
    [(inc! count1)]
    (await (= count1 n))
    [(dec! count2)]
    (await (= count2 0))
```

Znovupoužitelná bariéra

```
let ((count1 0) (count2 n))
      pracovník (i n)
  loop
    úkol pro i
    [(inc! count1)]
    (await (= count1 n))
    [(dec! count2)]
    (await (= count2 0))
```

... Dobře, ale roli počítadel je potřeba po každém průchodu prohodit.

Znovupoužitelná bariéra

Nebo dát obě role za sebe:

```
let ((count1 0) (count2 n))
      pracovník (i n)
 loop
   úkol pro i
   [(inc! count1)]
   (await (= count1 n))
   [(dec! count2)]
   (await (= count2 0))
   [(dec! count1)]
   (await (= count1 0))
   [(inc! count2)]
   (await (= count2 n))
```

Obsah

Vlajky

Vlajky

flags ... vektor logických hodnot (vlajek)

Vlajky

flags ... vektor logických hodnot (vlajek)

- (vref *flags* *i*) je *Pravda* ... vlajka *i* je vyvěšena
- (vref *flags* *i*) je *Nepravda* ... vlajka *i* je sejmuta

Vlajky

flags ... vektor logických hodnot (vlajek)

- `(vref` *flags* *i*`)` je *Pravda* ... vlajka *i* je vyvěšena
- `(vref` *flags* *i*`)` je *Nepravda* ... vlajka *i* je sejmuta

Pravidla:

1. Ten, kdo čeká na vyvěšení vlajky, ji snímá.
2. Vlajka nesmí být vyvěšena, dokud není jisté, že je sejmuta.

Vlajky

flags ... vektor logických hodnot (vlajek)

- (vref *flags i*) je *Pravda* ... vlajka *i* je vyvěšena
- (vref *flags i*) je *Nepravda* ... vlajka *i* je sejmuta

Pravidla:

1. Ten, kdo čeká na vyvěšení vlajky, ji snímá.
2. Vlajka nesmí být vyvěšena, dokud není jisté, že je sejmuta.

Procedury:

- (flag-set! *flags i*) ... vyvěsí vlajku
- (flag-clear! *flags i*) ... sejme vlajku
- (flag-set? *flags i*) ... rozhodne, zda je vlajka vyvěšena

Vlajky

flags ... vektor logických hodnot (vlajek)

- (vref *flags i*) je *Pravda* ... vlajka *i* je vyvěšena
- (vref *flags i*) je *Nepravda* ... vlajka *i* je sejmuta

Pravidla:

1. Ten, kdo čeká na vyvěšení vlajky, ji snímá.
2. Vlajka nesmí být vyvěšena, dokud není jisté, že je sejmuta.

Procedury:

- (flag-set! *flags i*) ... vyvěsí vlajku
- (flag-clear! *flags i*) ... sejme vlajku
- (flag-set? *flags i*) ... rozhodne, zda je vlajka vyvěšena
- (wait-and-clear! *flags i*) ... čeká na vyvěšení vlajky a pak ji sejme:

```
(await (flag-set? flags i))
(flag-clear! flags i)
```

Vlajky a koordinátoři

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
`pracovník` `(i n)`	*`koordinátor`*
`loop` *`úkol pro`* `i`	`loop`

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
pracovník `(i n)`	*koordinátor*

`loop`
`  úkol pro i`
`  (flag-set! arrive i)`
	`loop`

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
pracovník `(i n)`	*koordinátor*

`loop`
`  úkol pro i`
`  (flag-set! arrive i)`
`  (wait-and-clear! continue i)`
	`loop`

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
`pracovník` `(i n)`	*`koordinátor`*

`loop`
`  úkol pro i`
`  (flag-set! arrive i)`
`  (wait-and-clear! continue i)`
	`loop`
	`  (dotimes (i n)`
	`  (wait-and-clear! arrive i))`

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
pracovník `(i n)`	*koordinátor*

`loop`
`  úkol pro i`
`  (flag-set! arrive i)`
`  (wait-and-clear! continue i)`
	`loop`
	`  (dotimes (i n)`
	`  (wait-and-clear! arrive i))`
	`  (dotimes (i n)`
	`  (flag-set! continue i))`

Vlajky a koordinátoři

Vlajky:

- `(flag-set? arrive i)` ... proces `i` dorazil k bariéře
- `(flag-set? continue i)` ... proces `i` může opustit bariéru

`let ((arrive (vect n nil)) (continue (vect n nil)))`	
pracovník `(i n)`	*koordinátor*

`loop`
`  úkol pro i`
`  (flag-set! arrive i)`
`  (wait-and-clear! continue i)`
	`loop`
	`  (dotimes (i n)`
	`  (wait-and-clear! arrive i))`
	`  (dotimes (i n)`
	`  (flag-set! continue i))`

... nesymetrické

Bariéra se stromovou strukturou

Bariéra se stromovou strukturou

Pracovníci jsou uspořádání do struktury binárního stromu.

Bariéra se stromovou strukturou

Pracovníci jsou uspořádání do struktury binárního stromu.

- `(left i)` ... index levého následníka `(+ (* 2 i) 1)`
- `(right i)` ... index pravého následníka `(+ (* 2 i) 2)`

Bariéra se stromovou strukturou

Pracovníci jsou uspořádání do struktury binárního stromu.

- `(left i)` ... index levého následníka `(+ (* 2 i) 1)`
- `(right i)` ... index pravého následníka `(+ (* 2 i) 2)`

list `i`

```
(flag-set! arrive i)
(wait-and-clear! continue i)
```

Bariéra se stromovou strukturou

Pracovníci jsou uspořádání do struktury binárního stromu.

- `(left i)` ... index levého následníka `(+ (* 2 i) 1)`
- `(right i)` ... index pravého následníka `(+ (* 2 i) 2)`

list `i`

```
(flag-set! arrive i)
(wait-and-clear! continue i)
```

kořen `i`

```
(wait-and-clear! arrive (left i))
(wait-and-clear! arrive (right i))
(flag-set! continue (left i))
(flag-set! continue (right i))
```

Bariéra se stromovou strukturou

Pracovníci jsou uspořádání do struktury binárního stromu.

- `(left i)` ... index levého následníka `(+ (* 2 i) 1)`
- `(right i)` ... index pravého následníka `(+ (* 2 i) 2)`

list i

```
(flag-set! arrive i)
(wait-and-clear! continue i)
```

kořen i

```
(wait-and-clear! arrive (left i))
(wait-and-clear! arrive (right i))
(flag-set! continue (left i))
(flag-set! continue (right i))
```

vnitřní uzel i

```
(wait-and-clear! arrive (left i))
(wait-and-clear! arrive (right i))
(flag-set! arrive i)
(wait-and-clear! continue i)
(flag-set! continue (left i))
(flag-set! continue (right i))
```

Obsah

Symetrické bariéry

Symetrické bariéry

- `(wait-and-set!` *`flags i`*`)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Symetrické bariéry

- `(wait-and-set!` *flags* *i*`)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Synchronizace procesu `i` s procesem `j`:

proces `i`
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

Symetrické bariéry

- `(wait-and-set! flags i)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Synchronizace procesu `i` s procesem `j`:

proces i
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

- Bariéra pro `n` procesů je rozdělena do `k` fází.

Symetrické bariéry

- `(wait-and-set!` *`flags i`*`)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Synchronizace procesu `i` s procesem `j`:

proces `i`
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

- Bariéra pro `n` procesů je rozdělena do `k` fází.
- V každé fázi se synchronizuje proces s nějakým procesem.

Symetrické bariéry

- `(wait-and-set!` *`flags i`*`)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Synchronizace procesu `i` s procesem `j`:

proces `i`
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

- Bariéra pro `n` procesů je rozdělena do `k` fází.
- V každé fázi se synchronizuje proces s nějakým procesem.
- Každá fáze má vlastní vlajky: `(vect k (vect n nil))`

Symetrické bariéry

- `(wait-and-set!` *`flags i`*`)` ... čeká na sejmutí vlajky a pak ji vyvěsí

Synchronizace procesu `i` s procesem `j`:

proces `i`
`(wait-and-set! arrive i)`
`(wait-and-clear! arrive j)`

- Bariéra pro `n` procesů je rozdělena do `k` fází.
- V každé fázi se synchronizuje proces s nějakým procesem.
- Každá fáze má vlastní vlajky: `(vect k (vect n nil))`
- Fáze číslujeme od nuly.

Motýlí bariéra

Motýlí bariéra

- počet fází synchronizace: `k`

Motýlí bariéra

- počet fází synchronizace: `k`
- počet procesů: `(= n (expt 2 k))`

Motýlí bariéra

- počet fází synchronizace: `k`
- počet procesů: `(= n (expt 2 k))`

Ve fázi `f` se proces `i` synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Motýlí bariéra

- počet fází synchronizace: `k`
- počet procesů: `(= n (expt 2 k))`

Ve fázi `f` se proces `i` synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Přesněji se ve fázi `f` proces `i` synchronizuje s procesem:

- `(+ i d)`, pokud `(< (mod i (expt 2 (+ f 1))) d)`,
- `(- i d)`, jinak.

Motýlí bariéra

- počet fází synchronizace: `k`
- počet procesů: `(= n (expt 2 k))`

Ve fázi `f` se proces `i` synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Přesněji se ve fázi `f` proces `i` synchronizuje s procesem:

- `(+ i d)`, pokud `(< (mod i (expt 2 (+ f 1))) d)`,
- `(- i d)`, jinak.

Například:

proces	0	1	2	3	4	5	6	7
fáze 0	1	0	3	2	5	4	7	6
fáze 1	2	3	0	1	6	7	4	5
fáze 2	4	5	6	7	0	1	2	3

Rozšiřovací bariéra

Rozšiřovací bariéra

- počet procesů: `n`

Rozšiřovací bariéra

- počet procesů: `n`
- počet fází synchronizace: `(= k (ceiling (log n 2)))`

Rozšiřovací bariéra

- počet procesů: `n`
- počet fází synchronizace: `(= k (ceiling (log n 2)))`

Opět se ve fázi `f` proces synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Rozšiřovací bariéra

- počet procesů: `n`
- počet fází synchronizace: `(= k (ceiling (log n 2)))`

Opět se ve fázi `f` proces synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Proces `i` ve fázi `f` se synchronizuje s procesem `(mod (- i d) n))`.

Rozšiřovací bariéra

- počet procesů: `n`
- počet fází synchronizace: `(= k (ceiling (log n 2)))`

Opět se ve fázi `f` proces synchronizuje s procesem vzdáleným `(= d (expt 2 f))`.

Proces `i` ve fázi `f` se synchronizuje s procesem `(mod (- i d) n))`.

Například:

proces	0	1	2	3	4	5
fáze 0	5	0	1	2	3	4
fáze 1	4	5	0	1	2	3
fáze 2	2	3	4	5	0	1

Obsah

Náčrtek

- `arrive1`, `arrive2` ... kolikrát proces dorazil k bariéře
- `depart1`, `depart2` ... kolikrát proces opustil bariéru

`let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0))`	
1. pracovník	*2. pracovník*

`loop`
  úkol pro 1. pracovníka
  `{(= arrive1 depart1)}`
  `[(inc! arrive1)]`
  `{(= arrive1 (+ depart1 1))}`
  `[(inc! depart1)]`
  `{(= arrive1 depart1)}`
	`loop`
	  úkol pro 2. pracovníka
	  `{(= arrive2 depart2)}`
	  `[(inc! arrive2)]`
	  `{(= arrive2 (+ depart2 1))}`
	  `[(inc! depart2)]`
	  `{(= arrive2 depart2)}`

Podmínka: `(and (<= depart1 arrive2) (<= depart2 arrive1))`

Hrubé řešení

let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0))	
1. pracovník	*2. pracovník*

loop
  úkol pro 1. pracovníka
  {(= arrive1 depart1)}
  [(inc! arrive1)]
  {(= arrive1 (+ depart1 1))}
  [(await (< depart1 arrive2))
  (inc! depart1)]
  {(= arrive1 depart1)}
	loop
	  úkol pro 2. pracovníka
	  {(= arrive2 depart2)}
	  [(inc! arrive2)]
	  {(= arrive2 (+ depart2 1))}
	  [(await (< depart2 arrive1))
	  (inc! depart2)]
	  {(= arrive2 depart2)}

Invariant: `(and (<= depart1 arrive2) (<= depart2 arrive1))`

Zavedení zdrojů

- `(= barrier1 (- arrive1 depart2))`
- `(= barrier2 (- arrive2 depart1))`

```
let ((arrive1 0) (depart1 0) (arrive2 0) (depart2 0)
     (barrier1 0) (barrier2 0))
```

1. pracovník	*2. pracovník*
`loop`	`loop`
`úkol pro 1. pracovníka`	*`úkol pro 2. pracovníka`*
`[(inc! arrive1)`	`[(inc! arrive2)`
`(inc! barrier1)]`	`(inc! barrier2)]`
`[(await (< depart1 arrive2))`	`[(await (< depart2 arrive1))`
`(inc! depart1)`	`(inc! depart2)`
`(dec! barrier2)]`	`(dec! barrier1)]`

Invarianty:

- `(and (<= depart1 arrive2) (<= depart2 arrive1))`,
- `(<= 0 barrier1)`, `(<= 0 barrier2)`

Záměna proměnných

Invarianty:

- `(<= 0 barrier1)`
- `(<= 0 barrier2)`

`let ((barrier1 0) (barrier2 0))`	
1. pracovník	*2. pracovník*

`loop`
  úkol pro 1. pracovníka
  `[(inc! barrier1)]`
  `[(await (< 0 barrier2))`
  `(dec! barrier2)]`
	`loop`
	  úkol pro 2. pracovníka
	  `[(inc! barrier2)]`
	  `[(await (< 0 barrier1))`
	  `(dec! barrier1)]`

Řešení pomocí semaforů

let ((barrier1 (sem 0)) (barrier2 (sem 0)))	
1. pracovník	*2. pracovník*

loop
  úkol pro 1. pracovníka
  (v barrier1)
  (p barrier2)
	loop
	  úkol pro 2. pracovníka
	  (v barrier2)
	  (p barrier1)