Paradigmata programování 4

Přednáška 5. Semafory

verze z 11. března 2025

Jan Laštovička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Obsah

Problém s aktivním čekáním

Petersonův algoritmus:

```
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)

Problém s aktivním čekáním

Petersonův algoritmus:

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

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

- Čekající proces stále vyhodnocuje podmínku v `await`.

Problém s aktivním čekáním

Petersonův algoritmus:

```
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)

- Čekající proces stále vyhodnocuje podmínku v `await`.
- Nevadí, pokud je procesů méně nebo rovno počtu procesorů.

Problém s aktivním čekáním

Petersonův algoritmus:

```
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)

- Čekající proces stále vyhodnocuje podmínku v `await`.
- Nevadí, pokud je procesů méně nebo rovno počtu procesorů.
- Lze vyřešit lépe?

Semafor

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Dvě atomické operace:

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Dvě atomické operace:
p (passering, počkat)

- Vem zdroj. Případně čekej, než je zdroj k dispozici.

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Dvě atomické operace:
p (passering, počkat)

- Vem zdroj. Případně čekej, než je zdroj k dispozici.

v (vrijgave, vyhlásit)

- Přidej zdroj.

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Dvě atomické operace:
p (passering, počkat)

- Vem zdroj. Případně čekej, než je zdroj k dispozici.

v (vrijgave, vyhlásit)

- Přidej zdroj.

Zavedeme pomocné proměnné:

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

Semafor

- nizozemský informatik Edsger Wybes Dijkstra (1962)
- analogie semaforu na železnici
- abstraktní datová struktura
- Při vytvoření zadáme číslo `k` udávající počet zdrojů: `(sem k)`

Dvě atomické operace:
p (passering, počkat)

- Vem zdroj. Případně čekej, než je zdroj k dispozici.

v (vrijgave, vyhlásit)

- Přidej zdroj.

Zavedeme pomocné proměnné:

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

Invariant semaforu: `(<= cp (+ cv k))`

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Definujeme:

- Aktuální počet zdrojů `s`:

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Definujeme:

- Aktuální počet zdrojů `s`: `(= s (+ k (- cv cp)))`

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Definujeme:

- Aktuální počet zdrojů `s`: `(= s (+ k (- cv cp)))`

- Semafor ztotožníme s aktuálním počtem zdrojů (hodnotou semaforu).

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Definujeme:

- Aktuální počet zdrojů `s`: `(= s (+ k (- cv cp)))`

- Semafor ztotožníme s aktuálním počtem zdrojů (hodnotou semaforu).
- Hodnotu semaforu nelze přímo získat ani nastavit.

Počet zdrojů

Máme:

- `k` ... výchozí počet zdrojů
- `cv` ... počet vykonání operací `v`
- `cp` ... počet vykonání operací `p`

Definujeme:

- Aktuální počet zdrojů `s`: `(= s (+ k (- cv cp)))`

- Semafor ztotožníme s aktuálním počtem zdrojů (hodnotou semaforu).
- Hodnotu semaforu nelze přímo získat ani nastavit.

Alternativní invariant semaforu SEM = `(>= s 0)`

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

invariant není zachován $\Rightarrow$ použijeme sladění

Operace p

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

invariant není zachován ⇒ použijeme sladění

Operace p

- s ... aktuální počet zdrojů
- invariant: (>= s 0)

Z pravidla přiřazení dostáváme:

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

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

invariant není zachován ⇒ použijeme sladění

Tedy operace (p s) je:

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

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

- Předpoklad `(>= s -1)` zesílíme na `(>= s 0)`.

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

- Předpoklad `(>= s -1)` zesílíme na `(>= s 0)`.

Operace v

- `s` ... aktuální počet zdrojů
- invariant: `(>= s 0)`

Z pravidla přiřazení dostáváme:

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

- Předpoklad `(>= s -1)` zesílíme na `(>= s 0)`.

Tedy operace `(v s)` je pouze:

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

Obsah

Náčrtek pro semafory

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

Náčrtek pro semafory

let ((in1 0) (in2 0))	
loop	loop
nekritická sekce	*nekritická sekce*
vstupní protokol	*vstupní protokol*
kritická sekce	*kritická sekce*
výstupní protokol	*výstupní protokol*

Náčrtek pro semafory

`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)`

Náčrtek pro semafory

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

Náčrtek pro semafory

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

Vzájemné vyloučení ... $CS =$

Náčrtek pro semafory

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

Vzájemné vyloučení ... CS = (<= (+ in1 in2) 1)

Náčrtek pro semafory

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

Vzájemné vyloučení ... CS = `(<= (+ in1 in2) 1)`

Z pravidla přiřazení: `{(set! in1 1) (<= (+ in1 in2) 1)}` a `{(set! in2 1) (<= (+ in1 in2) 1)}`

Náčrtek pro semafory

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

Vzájemné vyloučení ... CS = (<= (+ in1 in2) 1)

Z pravidla přiřazení: {(<= in2 0)} (set! in1 1) {(<= (+ in1 in2) 1)} a {(<= in1 0)} (set! in2 1) {(<= (+ in1 in2) 1)}

Řešení nahrubo

CS = `(<= (+ in1 in2) 1)`

`let ((in1 0) (in2 0))`	

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

Řešení nahrubo

CS = `(<= (+ in1 in2) 1)`

```
let ((in1 0) (in2 0))
```

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

Řešení nahrubo

CS = `(<= (+ in1 in2) 1)`

`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
  `(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
	  `(set! in2 0)`
	  `{(and (= in2 0)` *CS*`)}`

Použití semaforů

Invariant: `(<= (+ in1 in2) 1)`

```
let ((in1 0) (in2 0))
```

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

Použití semaforů

Invariant: `(<= (+ 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)`

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`

Použití semaforů

Invariant: `(<= (+ in1 in2) 1)`

`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)]`

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`

Použití semaforů

Invariant: `(<= (+ in1 in2) 1)`

`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))]`

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`
2. převedeme řízení na `mutex`

Použití semaforů

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)]

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`
2. převedeme řízení na `mutex`
3. odstraníme `in1` a `in2`

Použití semaforů

let ((mutex (sem 1)))	
loop *nekritická sekce* (p mutex) *kritická sekce* (v mutex)	loop *nekritická sekce* (p mutex) *kritická sekce* (v mutex)

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`
2. převedeme řízení na `mutex`
3. odstraníme `in1` a `in2`
4. nahradíme `mutex` semaforem

Použití semaforů

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

1. zavedeme: `(= mutex (- 1 (+ in1 in2)))`
2. převedeme řízení na `mutex`
3. odstraníme `in1` a `in2`
4. nahradíme `mutex` semaforem
5. lze jednoduše zobecnit na libovolný počet procesů

Obsah

Stav procesu

Stav procesu

Současné stavy:

Stav procesu

Současné stavy:

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

Stav procesu

Současné stavy:

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

Nový stav: Blokovaný

Stav procesu

Současné stavy:

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

Nový stav: Blokovaný

- Proces nemůže být vykonáván.

Stav procesu

Současné stavy:

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

Nový stav: Blokovaný

- Proces nemůže být vykonáván.
- Běžící proces se může zablokovat.

Stav procesu

Současné stavy:

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

Nový stav: Blokovaný

- Proces nemůže být vykonáván.
- Běžící proces se může zablokovat.
- Odblokovat jej může pouze běžící proces.

Stav procesu

Současné stavy:

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

Nový stav: Blokovaný

- Proces nemůže být vykonáván.
- Běžící proces se může zablokovat.
- Odblokovat jej může pouze běžící proces.
- Kam ukládat blokované procesy?

Blokující semafor

Upravíme operace semaforu.

Blokující semafor

Upravíme operace semaforu.

Atomické operace:

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
p (počkat)

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
p (počkat)

- Pokud je hodnota semaforu kladná, dekrementuj ji,

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
p (počkat)

- Pokud je hodnota semaforu kladná, dekrementuj ji,
- jinak zablokuj běžící proces.

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
p (počkat)

- Pokud je hodnota semaforu kladná, dekrementuj ji,
- jinak zablokuj běžící proces.

v (vyhlásit)

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
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,

Blokující semafor

Upravíme operace semaforu.

Atomické operace:
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.

Obsah

Implementace semaforu

Implementace semaforu

Semafor je pár (*value* . *blocked*)

Implementace semaforu

Semafor je pár (*value* . *blocked*)

- *value* je nezáporné celé číslo (hodnota semaforu)

Implementace semaforu

Semafor je pár (*value* . *blocked*)

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

Implementace semaforu

Semafor je pár (*value* . *blocked*)

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

Blokované procesy budeme ukládat do fronty

Implementace semaforu

Semafor je pár (*value* . *blocked*)

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

Blokované procesy budeme ukládat do fronty $\Rightarrow$ dostaneme férový semafor

Konstruktor semaforu

Konstruktor semaforu

```
(val -- semaphore)
```

Konstruktor semaforu

(*val -- semaphore*)

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

Operace p

Operace p

(*semaphore* --)

Operace 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

Operace v

(*semaphore* --)

Operace v

(*semaphore* --)

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

Ukázky

Ukázky

1

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

Ukázky

1

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

2

```
0 :sem :p
```

Ukázky

1

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

2

```
0 :sem :p
```

3

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

Semafor ve Scheme

Semafor ve Scheme

Procedury pro práci se semafory:

Semafor ve Scheme

Procedury pro práci se semafory:

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

Semafor ve Scheme

Procedury pro práci se semafory:

```
(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))))
```

Nová expanze makra co

Nová expanze makra co

- Makro co nově používá semafory.

`(co` *`body1 body2 ... bodyn`*`)` $\Rightarrow$

Nová expanze makra co

- Makro co nově používá semafory.

`(co` *body1 body2 ... bodyn*`)` $\Rightarrow$

```
(let ((start (sem 0))
      (end (sem 0)))
```

Nová expanze makra co

- Makro co nově používá semafory.

(co *body1 body2 ... bodyn*) $\Rightarrow$

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

Nová expanze makra co

- Makro co nově používá semafory.

(co *body1 body2 ... bodyn*) $\Rightarrow$

```
(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)))
```

Nová expanze makra co

- Makro co nově používá semafory.

(co *body1 body2 ... bodyn*) ⇒

```
(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)))
```

Makro není rekurzivní.

Co se stane?

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

Obsah

Řešení kritické sekce zámkem

```
let ((l (lock)))
```

proces i

```
loop
  nekritická sekce
  (with-lock (l)
    kritická sekce)
```

Jak naprogramovat?

Zámek

Zámek

Vytvoření zámku:

Zámek

Vytvoření zámku:

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

Zámek

Vytvoření zámku:

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

Makro `with-lock`:

```
(with-lock (l)
  expr1
  ⋮                ⇒
  exprn)
```

Zámek

Vytvoření zámku:

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

Makro `with-lock`:

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

Zámek

Vytvoření zámku:

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

Makro `with-lock`:

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

výkonnostní problémy ⇒ přepsat na speciální operátor

with-lock jako 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

```
(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)
```