1 Monitor

Monitor je objekt, který zabaluje data a procedury (tak, jak to známe z OOP), ale navíc také synchronizaci. Metody monitoru jsou všechny synchronizované, tedy pouze jedno vlákno může zaráz vykonávat libovolnou metodu monitoru. Navíc monitor může obsahovat podmínky, ke kterým se dostaneme.

Příklad 1. Jednoduchým příkladem je synchronizovaný čítač. V naší knihovně bychom podědili třídu `MONITOR`.

```
(defclass counter (monitor)
  ((count :initform 0)))
```

Inkrementaci pak definujeme makrem `DEFINE-MONITOR-METHOD`.

```
(define-monitor-method increment ((counter counter))
  (incf (slot-value counter 'count)))
```

Je důležité, aby třída, která dědí z monitoru byla uvedena jako první argument.

Monitor si můžeme představit tak, že obsahuje mutex, na který se čeká na začátku každé metody a na konci každé metody se signalizuje. Nicméně pokud je potřeba v rámci metody počkat než bude splněna nějaká podmínka, můžeme monitoru přiřadit takzvanou podmíněnou proměnnou, na kterou lze čekat, což uspí vlákno a signalizuje mutex, aby ostatní vlákna měly možnost podmínku splnit. Splnění podmínky je pak potřeba signalizovat.

Poznámka. Je dobré podotknout, že mutex, bežně nazývaný také zámek, musí být takzvaně reentrantní, nebo-li aby jedno vlákno mohlo vícekrát signalizovat tento mutex. Pokud by reentrantní nebyl, nastal by deadlock, a to v případě, že v rámci jedné metody monitoru je zavolána jiná metoda monitoru.

Příklad 2. Pokud bychom chtěli, aby čítač šlo také dekrementovat, ale pouze pokud je hodnota kladná, přidáme podmíněnou proměnnou.

```
(defclass counter (monitor)
  ((count :initform 0)
   (positivep :initform (condition-variable))))
```

Samotná dekrementace pak podmíněnou proměnnou využívá.

```
(define-monitor-method decrement ((counter counter))
  (with-slots (count positivep) counter
    (loop :until (plusp count)
       :do (wait-on-condition positivep))
    (decf count)))
```

Inkrementace pak musí signalizovat, že hodnota už je znovu kladná.

```
(define-monitor-method increment ((counter counter))
  (incf (slot-value counter 'count))
  (signal-condition (slot-value counter 'positivep)))
```

Podmíněné proměnné, které používáme jsou neblokující, což znamená, že vlákno, které signalizuje proměnnou není blokováno a dokončí svou práci. To má ale za následek, že je potřeba po každém čekání zkontrolovat, jestli už je podmínka splněna.

Poznámka. Existují ještě další typy podmíněných proměnných. Jednou z nich je blokující, která blokuje vlákno, které signalizuje proměnnou. Tento typ byl prvním návrhem [1], [2]. Druhou pak je implicitní, která nespecifikuje podmíněné proměnné, ale je obsažena jedna implicitní podmínka pro jeden monitor. Pak tedy není potřeba žádnou vytvářet, pouze se čeká na a signalizuje samotný monitor. Tento typ je například použit v Javě, kde metody monitoru jsou označeny klíčovým slovem `synchronized` a pro podmíněnou proměnnou používáme `wait` a `notify/notifyAll`.

Neblokující podmíněné proměnné se používají např. v Pthread nebo také ve většině implementací CL. Lze je také najít v knihovně `java.util.concurrent.locks`.

Schematické obrázky vysvětlující princip fungování jednotlivých typů lze najít na wikipedii.

1.1 Producent-konzument

Řešením tohoto problému je v podstatě vytvoření sdílené blokující fronty, která bude mít metody `enqueue` a `dequeue`. Použijeme podobnou terminologii a řešení jako při použití semaforů.

```
(defclass queue (monitor)
  ((itemsp :initform (make-instance 'condition-variable))
   (spacep :initform (make-instance 'condition-variable))
   (buffer :initform nil)
   (capacity :initform 1 :initarg :capacity)))
(defun queue (capacity)
  (make-instance 'queue :capacity capacity))
```

Podmíněná proměnná `itemsp` značí konzumentům, že existuje nějaký prvek v bufferu a naproti tomu `spacep` značí producentům, že je místo v bufferu.

```
(define-monitor-method enqueue ((queue queue) value)
  (with-slots (buffer capacity spacep itemp) queue
    (loop :while (= (length buffer) capacity)
          :do (wait-on-condition spacep))
    (setf buffer `(,@buffer ,value))
    (signal-condition itemp))
  t)
```

```
(define-monitor-method dequeue ((queue queue))
  (with-slots (buffer capacity spacep itemp) queue
    (loop :while (null buffer)
          :do (wait-on-condition itemp))
    (let ((value (pop buffer)))
      (signal-condition spacep)
      value)))
```

1.2 Čtenáři-písaři

Abychom tento problém vyřešili dostatečně obecně, vytvoříme třídu, která bude sloužit jako zámek do místnosti, kde budou dokumenty, které se budou číst a do kterých se bude zapisovat. Bude mít dvoje dveře, jedny pro čtenáře a jedny pro písaře, jako tomu bylo u řešení pomocí semaforů. Budeme mít tedy čtyři metody monitoru `reader-enter`, `reader-leave`, `writer-enter` a `writer-enter`.

```
(defclass read-write-lock (monitor)
  ((emptyp :initform (make-instance 'condition-variable))
   (writerp :initform nil)
   (readers :initform 0)))
(defun read-write-lock ()
  (make-instance 'read-write-lock))
```

Použijeme pouze jednu podmíněnou proměnnou `emptyp`, která značí, že místnost je prázdná. Dále proměnná `writerp`, značí, že v místnosti je čtenář a proměnná `readers` bude uvádět počet čtenářů v místnosti.

```
(define-monitor-method reader-enter ((rw read-write-lock))
  (with-slots (writerp readers emptyp) rw
    (loop :while writerp
          :do (wait-on-condition emptyp))
    (incf readers))
  t)

(define-monitor-method reader-leave ((rw read-write-lock))
  (with-slots (writerp readers emptyp) rw
    (decf readers)
    (when (zerop readers)
      (broadcast-condition emptyp)))
  t)

(define-monitor-method writer-enter ((rw read-write-lock))
  (with-slots (writerp readers emptyp) rw
    (loop :while (or writerp (positivep readers))
          :do (wait-on-condition emptyp))
    (setf writerp t))
  t)
```

```
(define-monitor-method writer-leave ((rw read-write-lock))
  (with-slots (writerp emptyp) rw
    (setf writerp nil)
    (broadcast-condition emptyp)
  t)
```

Čtenáři čekají dokud je v místnosti písař a písaři čekají dokud jsou v místnosti čtenáři nebo písař. Při výstupu z místnosti, poslední čtenář signalizuje, že nikdo není v místnosti a v případě odchodu písaře jsme použili k signalizaci metodu `broadcast-condition`, která probudí všechna čekající vlákna. Tady je to potřeba, jelikož v případě, že čeká více čtenářů, chceme probudit všechny a nemůžeme vědět, kolik jich na podmínce čeká.

V tomto řešení mají stejně jako u prvního řešení se semafory přednost čtenáři. Abychom dali přednost písařům, je potřeba uspat všechny příchozí čtenáře, jakmile je před vchodem první písař a ten musí počkat než budou pryč všichni čtenáři. Jednoduchou úpravou metody `writer-enter` tohoto docílíme.

```
(define-monitor-method writer-enter ((rw read-write-lock))
  (with-slots (writerp readers emptyp) rw
    (loop :while writerp
          :do (wait-on-condition emptyp))
    (setf writerp t)
    (loop :while (positivep readers)
          :do (wait-on-condition emptyp)))
  t)
```

Jakmile přijde první písař, nastaví slot `writerp` na `t` a tím zajistí, že už žádní noví čtenáři neprojdou. Pak už stačí akorát počkat, než všichni místnost opustí.

Zde by bylo ještě idiomatické přejmenovat podmíněnou proměnnou na `nowriterp` a přidat ještě navíc `noreaderp`. Nicméně řešení je funkční i takto.

Odkazy

[1] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Commun. ACM*, roč. 17, č. 10, 549–557, říj. 1974, ISSN: 0001-0782. DOI: 10.1145/355620.361161. WWW: https://doi.org/10.1145/355620.361161.

[2] P. B. Hansen, "Structured Multiprogramming", *Commun. ACM*, roč. 15, č. 7, 574–578, čvc 1972, ISSN: 0001-0782. DOI: 10.1145/361454.361473. WWW: https://doi.org/10.1145/361454.361473.