MP18 @ II UWr 11 czerwca 2018 r.

# Lista zagadnień nr 15

## Przed zajęciami

Tematem przewodnim jest programowanie współbieżne. Przed zajęciami należy zapoznać się z **Rozdziałem 3.4** podręcznika. Należy rozumieć pojęcia **wątku**, **współbieżności**, **synchronizacji**, **blokady** (ang. *lock*), **zakleszczenia** (ang. *deadlock*).

## W trakcie zajęć

### Ćwiczenie 1.

Przyjmijmy, że Maciej, Filip i Marek mają wspólne konto, na którym początkowo znajduje się 100 jednostek. Współbieżnie Maciej deponuje 10 jednostek, Filip pobiera 20, natomiast Marek pobiera połowę jednostek z konta. Operacje te można zapisać następująco (kolejność arbitralna):

```
(set! balance (+ balance 10))           ; Maciej
(set! balance (- balance 20))           ; Filip
(set! balance (- balance (/ balance 2))) ; Marek
```

- Załóżmy, że system bankowy wymusza, aby operacje na koncie zostały wykonane w jakiejś konkretnej kolejności (nie wiemy, jakiej). Jakie są możliwe końcowe stany konta?
- Załóżmy, że dozwolone jest dowolne przeplatanie operacji (tzn. pomiędzy odczytaniem i zapisaniem wartości w jednym z wątków inny wątek może wykonać zapis). Jakie teraz są możliwe stany konta?

### Ćwiczenie 2.

Jakie są (teoretycznie) możliwe wartości x po wykonaniu następującego kodu:

```
(define x 10)
(run-concurrent
```

```
  (lambda () (set! x (* x x)))
  (lambda () (set! x (* x x x))))
```

Które z tych wartości są dalej możliwe po zastosowaniu synchronizacji:

```
(define x 10)
(define s (make-serializer))
(run-concurrent
  (s (lambda () (set! x (* x x))))
  (s (lambda () (set! x (* x x x)))))
```

### Ćwiczenie 3.

Zmodyfikujmy implementację kont bankowych (z synchronizacją) w taki sposób, aby zwracanie stanu konta odbywało się również przez synchronizowaną metodę:

```
  (define (dispatch m)
    (cond [(eq? m 'withdraw) (protected withdraw)]
          [(eq? m 'deposit) (protected deposit)]
          [(eq? m 'balance) (protected (lambda () balance))]
          [else (error "Unknown request -- MAKE-ACCOUNT"
                       m)]))
```

Czy taka definicja chroni nas przed jakimiś niechcianymi zachowaniami? Jeśli tak, to zademonstruj przykładowy scenariusz; jeśli nie, uzasadnij.

### Ćwiczenie 4.

Rozważ problem transferu pomiędzy dwoma kontami. Załóżmy, że został zaimplementowany w taki sposób:

```
(define (transfer from-account to-account amount)
  ((from-account 'withdraw) amount)
  ((to-account 'deposit) amount))
```

Czy takie podejście jest prawidłowe? Czy może musimy z jakiegoś powodu zastosować bardziej złożoną synchronizację, tak jak dla zamiany wartości kont przedstawionej na wykładzie?

### Ćwiczenie 5.

Rozważmy implementację kont bankowych, w których metody `deposit` i `withdraw` są synchronizowane, a oprócz tego udostępniany jest synchronizator:

```
(define (dispatch m)
  (cond [(eq? m 'withdraw) (account-serializer withdraw)]
        [(eq? m 'deposit) (account-serializer deposit)]
        [(eq? m 'balance) balance]
        [(eq? m 'serializer) account-serializer]
        [else (error "Unknown request -- MAKE-ACCOUNT"
                     m)]))
```

Jaki jest problem z takim rozwiązaniem? *Podpowiedź*: co stanie się, gdy będziemy chcieli zamienić wartości dwóch kont?

**Ćwiczenie 6.**

Omów scenariusz, w którym ochrona przed zakleszczeniami zaprezentowana na wykładzie nie wystarcza. *Podpowiedź*: rozwiązanie z wykładu wymaga, aby znać z wyprzedzeniem wszystkie zasoby, z których chcemy korzystać (np. zanim zaczniemy zamieniać wartości kont, wiemy już, na których kontach chcemy wykonać operacje).