Lista zagadnień nr 15

Programowanie współbieżne

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).

Zadania domowe

Zadanie 23

Problem obiadujących filozofów to klasyczny problem z dziedziny synchronizacji współbieżnych procesów. Wyobraźmy sobie pięciu filozofów siedzących przy okrągłym stole. Przy stole jest pięć talerzy (po jednym dla każdego filozofa) i pięć widelców (pomiędzy talerzami – każdy filozof ma z lewej i prawej strony po jednym widelcu). Kiedy filozof robi się głodny, musi podnieść dwa widelce (po swojej lewej i prawej stronie), aby zjeść posiłek. Po skończonym posiłku odkłada widelce, aby jego koledzy też mogli ich użyć.

Zaimplementuj procedurę `philosopher`, mającą realizować zjedzenie posiłku przez filozofa. (Procedura ta powinna być oczywiście udostępniona skryptowi sprawdzającemu przy pomocy formy `provide`). Procedura ta będzie otrzymywać dwa parametry. Pierwszym będzie reprezentacja stołu `dining-table`, a drugim – numer filozofa `k` (z zakresu od 0 do 4). Filozof może podnieść `i`-ty widelec, używając wywołania:

```
((dining-table 'pick-fork) i)
```

Odłożenie widelca natomiast jest realizowane wywołaniem:

```
((dining-table 'put-fork) i)
```

Filozof o numerze `k` może sięgnąć tylko do widelców o numerach `k` i $k+1$ (modulo 5). W ramach procedury należy podnieść dwa widelce (co oznacza skonsumowanie posiłku przez filozofa), a następnie je odłożyć.

Każdy filozof będzie posiadać własny, współbieżnie działający do pozostałych wątek. Każdy z tych pięciu wątków będzie wywoływać procedurę `philosopher` pewną liczbę razy (w zależności od tego, jak bardzo głodny jest dany filozof). Należy zadbać o to, aby nigdy nie nastąpiło zakleszczenie (i filozofowie nie umarli z głodu, czekając na swoje widelce).

Nie ma potrzeby synchronizować samodzielnie wywołań `pick-fork` i `put-fork` – operacje te są zaimplementowane w taki sposób, że współbieżne wywołania

tych procedur są bezpieczne. Próba podniesienia widelca, który już został podniesiony przez innego filozofa, spowoduje, że wątek będzie oczekiwał na odłożenie widelca.

Reprezentacja stołu używana do sprawdzenia rozwiązania nie jest udostępniona. Przetestowanie rozwiązania lokalnie wymaga zaimplementowania własnej reprezentacji. Taka implementacja nie jest jednak częścią zadania i nie trzeba jej załączać do rozwiązania.