Programowanie współbieżne

Lista zadań nr 5

Na ćwiczenia 23 listopada 2022

Zadanie 1. Podaj przykład dwóch nietrywialnych diagramów sekwencyjnie spójnych historii oraz dwóch, które nie są sekwencyjnie spójne. Historie mogą dotyczyć dowolnych (czyli wybranych przez Ciebie) współbieżnych struktur danych.

Wskazówka: Możesz wzorować się na diagramach z poprzedniej listy.

Zadanie 2. Pokaż, że sekwencyjna spójność nie ma własności kompozycji.

Wskazówka: slajdy 165-175, PRW-3.pdf. The Art of Multiprocessor Programming 2e, rozdział 3.3.3.

Zadanie 3.

1. Uzasadnij, że klasa **WaitFreeQueue** poprawnie implementuje dwuwątkową kolejkę współbieżną na maszynie z atomowym dostępem do pamięci (dostępy są linearyzowalne). W szczególności pokaż, co dzieje się gdy obydwa wątki współbieżnie wykonują operacje na (prawie) pustej i (prawie) pełnej kolejce.
2. Zauważ, że kolejka ta działa również poprawnie na maszynie z sekwencyjnie spójną pamięcią.

Wskazówka: Kod klasy WaitFreeQueue znajduje się na slajdzie 152. Obserwacja w p. 2. będzie nieomal trywialna.

Zadanie 4. Przypomnij dowód własności wzajemnego wykluczania dla algorytmu Petersona. Pokaż dlaczego ten dowód może się załamać dla procesora o modelu pamięci *słabszym*[1] niż sekwencyjna spójność.

Wskazówka: slajdy 181-183, PRW-3.pdf.

Zadanie 5. Pokaż, że klasa **WaitFreeQueue** nie jest poprawną implementacją dwuwątkowej kolejki, na maszynie o modelu pamięci słabszym niż sekwencyjna spójność[2].

---

[1] np. dopuszczającym zmianę kolejności wykonywania instrukcji w wątku (ang. *out of order execution*)

[2] By uzyskać implementację poprawną, do przechowywania zmiennych współdzielonych należy użyć wątkowo bezpiecznych obiektów, jak w kolejnych zadaniach.

Wskazówka: Ta implementacja podana jest w języku wysokiego poziomu i musi być przetłumaczona, przed uruchomieniem, do języka bliższego maszynie. W języku tym, chcąc operować na współdzielonych zmiennych, trzeba je najpierw sprowadzić do pamięci lokalnej. Np. instrukcja head++ może być przetłumaczona do lok1 = head, lok1 = lok1 + 1, head = lok1, gdzie lok1 jest zmienną lokalną. W zmiennych lokalnych pamięta się wartości wyrażeń arytmetycznych przed ich ponownym zapisem do pamięci współdzielonej. Procesor może wykonać taki program zmieniając kolejność instrukcji w ramach pojedynczego wątku.

Zadanie 6. Klasa **AtomicInteger**[3] opakowuje wartość typu całkowitego udostępniając metody niepodzielnego dostępu, np. **boolean compareAndSet(int expect, int update).** Metoda ta porównuje wartość zapisaną w obiekcie z argumentem **expect** i jeśli są równe, to zmienia zapisaną wartość na **update.** W przeciwnym przypadku nic się nie dzieje. Porównanie i ewentualna zmiana zachodzą w sposób niepodzielny (atomowy). Klasa ta udostępnia też metodę **int get()** zwracającą wartość zapisaną w obiekcie. Modyfikacje obiektów tej klasy są natychmiast widoczne dla wszystkich wątków w programie.

Z użyciem klasy **AtomicInteger** zaprogramowano poniższą implementację kolejki FIFO, dopuszczającej wiele wątków wkładających i wyciągających elementy. Pokaż, że jest ona niepoprawna. W tym celu pokaż, że nie jest linearyzowalna.

```
class IQueue<T> {
    AtomicInteger head = new AtomicInteger(0);
    AtomicInteger tail = new AtomicInteger(0);
    T[] items = (T[]) new Object[Integer.MAX_VALUE];
    public void enq(T x) {
        int slot;
        do {
            slot = tail.get();
        } while (!tail.compareAndSet(slot, slot+1));
        items[slot] = x;
    }
    public T deq() throws EmptyException {
        T value;
        int slot;
        do {
            slot = head.get();
            value = items[slot];
            if (value == null)
                throw new EmptyException();
        } while (!head.compareAndSet(slot, slot+1));
        return value;
    }
}
```

Zadanie 7. Poniższa implementacja kolejki FIFO dopuszczającej wiele wątków wkładających i wyciągających elementy, używa klas **AtomicInteger** oraz **AtomicReference<T>**[4]**.** Pokaż, że w treści metody **enq()** nie ma pojedyńczego punktu linearyzacji, a

[3] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html
[4] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicReference.html

dokładniej: a) pierwsza instrukcja **enq()** nie jest punktem linearyzacji oraz b) druga instrukcja **enq()** nie jest punktem linearyzacji. Czy z powyższych punktów wynika, że **enq()** nie jest linearyzowalna?

Wskazówka: dla każdego z punktów a) i b) podaj diagram wykonania z dwoma wykonaniami enq() i jednym deq(), w których metody enq() nie są zlinearyzowane w porządku wykonania pierwszej (odpowiednio drugiej) instrukcji. Oprócz samych wykonań metod, na diagramie wygodnie będzie zaznaczyć te instrukcje.

```
public class HWQueue<T> {
    AtomicReference<T>[] items;
    AtomicInteger tail;
    static final int CAPACITY = Integer.MAX_VALUE;

    public HWQueue() {
        items = (AtomicReference<T>[]) Array.newInstance(AtomicReference.class,
                CAPACITY);
        for (int i = 0; i < items.length; i++) {
            items[i] = new AtomicReference<T>(null);
        }
        tail = new AtomicInteger(0);
    }

    public void enq(T x) {
        int i = tail.getAndIncrement();
        items[i].set(x);
    }

    public T deq() {
        while (true) {
            int range = tail.get();
            for (int i = 0; i < range; i++) {
                T value = items[i].getAndSet(null);
                if (value != null) {
                    return value;
                }
            }
        }
    }
}
```

Zadanie 8. Uzasadnij[5], że **HWQueue<T>** jest poprawną implementacją kolejki FIFO dla wielu wątków.

[5] Nie wymagam formalnego dowodu.