Programowanie współbieżne

Lista zadań nr 5

Na ćwiczenia 18. listopada 2021

Zadanie 1 (przeformułowane zad. 5. z listy 4). Algorytm implementujący zamek nazywamy First-Come-First-Served (FCFS) gdy dla każdego systemu złożonego z przynajmniej dwóch wątków A i B, jeśli $D_A^j \rightarrow D_B^k$ to $CS_A^j \rightarrow CS_B^k$, gdzie D_C^i oraz CS_C^i oznaczają odpowiednio i-te wykonanie sekcji wejściowej algorytmu oraz i-te zajęcie zamka przez wątek C. Intuicyjnie oznacza to, że jeśli A zakończy wykonanie sekcji wejściowej zanim zrobi to B, to A weźmie zamek zanim zrobi to B. Pokazaliśmy, że algorytm Petersona jest FCFS gdy za sekcję wejściową przyjąć kilka pierwszych instrukcji metody **lock()**, a dokładniej instrukcje odczytu numeru wątku oraz ustawiania flag i ofiary. Zmieńmy teraz definicję sekcji wejściowej tak, by oznaczała po prostu pierwszą instrukcję metody **lock()**.

1. Pokaż, że żaden z poniższych wariantów algorytmu Petersona nie jest FCFS przy tak zmienionej definicji sekcji wejściowej:
   a. pierwszą instrukcją algorytmu jest odczyt numeru wątku: i = Thread.getID(), gdzie i jest zmienną lokalną wątku,
   b. pierwszą instrukcją jest ustawienie flagi: flag[Thread.getID()] = true,
   c. pierwszą instrukcją jest ustawienie ofiary: victim = Thread.getID(), czyli instrukcje ustawiania flag i ofiary są w odwróconej kolejności w stosunku do oryginalnego algorytmu.
2. Wywnioskuj stąd, że **żaden** algorytm złożony z więcej niż jednej instrukcji nie jest FCFS, jeśli sekcja wejściowa to pierwsza instrukcja metody **lock()**.

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

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

Zadanie 3. Przypomnij dowód własności wzajemnego wykluczania dla algorytmu Petersona. Pokaż dlaczego ten dowód może się załamać dla procesora o współczesnej architekturze.

Wskazówka: slajdy 181-183.

Zadanie 4. Klasa **AtomicInteger**[1] 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 5. Poniższa implementacja kolejki FIFO dopuszczającej wiele wątków wkładających i wyciągających elementy, używa klas **AtomicInteger** oraz **AtomicReference<T>**[2]. Pokaż, że w treści metody **enq()** nie ma pojedyńczego punktu linearyzacji, a 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?

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

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 6. Uzasadnij[3], że **HWQueue<T>** jest poprawną implementacją kolejki FIFO dla wielu wątków.

Zadanie 7. Dana jest metoda **weird(),** której każde i-te wywołanie powraca po wykonaniu 2^i instrukcji. Czy ta metoda ma własność *wait-free*? A *bounded wait-free*?

[3] Nie wymagam formalnego dowodu.