Programowanie współbieżne

Lista zadań nr 5

Na ćwiczenia 6 i 7 listopada 2024

Zadanie 1. W kodzie metody **lock()** z zadania 9 z listy 3, instrukcję **label[i] = counter++** zastępujemy instrukcją **label[i] = max(label[i], counter++)**. Czy tak zmodyfikowany algorytm spełnia warunki a) wzajemnego wykluczania, b) niezakleszczenia, c) niezagłodzenia?

Zadanie 2. 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;
    }
```

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

```
}
```

Zadanie 3. 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?

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

---

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

[3] Nie wymagam formalnego dowodu.

Zadanie 5. Zdefiniuj następujące własności postępu: **nieczekanie** (ang. *wait-freeness*), **niewstrzymywanie** (ang. *lock-freeness*). Dlaczego te własności nazywane są **nieblokującymi** (ang. *non-blocking*) i **niezależnymi** (ang. *independent*). Dlaczego **niezakleszczenie** i **niezagłodzenie** są **blokujące** i **zależne**? Czy współbieżna metoda w nietrywialny sposób wykorzystująca zamki może być nieczekająca? Dana jest metoda **weird()**, której każde i-te wywołanie powraca po wykonaniu 2^i instrukcji. Czy ta metoda jest nieczekająca? A **nieczekająca z limitem kroków** (ang. *bounded wait-free*)?

Wskazówka: The Art of Multiprocessor Programming 2e, rozdział 3.8.

Zadanie 6. W 32-bitowym systemie komputerowym wszystkie komórki pamięci są atomowymi 32-bitowymi rejestrami MRMW. Chcesz w tym systemie zasymulować programowo 64-bitowy rejestr za pomocą dwóch rejestrów 32-bitowych. W tym celu implementujesz metody **read()** i **write()**, które odczytują i zapisują te dwa rejestry po kolei. Czy otrzymany 64-bitowy rejestr jest bezpieczny, regularny, atomowy czy też nie spełnia żadnego z tych warunków?

Zadanie 7. Algorytm Petersona spełnia warunki wzajemnego wykluczania i niezagłodzenia, jeśli zmienne **flag[0]**, **flag[1]** oraz **victim** oznaczają rejestry atomowe. Czy te warunki zostaną zachowane, jeśli dla **flag[0]** i **flag[1]** użyjemy rejestrów regularnych?

Zadanie 8. Rejestr typu SRSW nazywamy **1-regularnym**, jeśli spełnia następujące warunki:

- wywołanie **read()** która nie jest współbieżne z żadnym wywołaniem **write()** zawsze zwraca wartość umieszczoną tam przez ostatnie wcześniejsze wywołanie **write()**,
- wywołanie **read()** współbieżne z dokładnie jednym wywołaniem **write()** zwraca wartość zapisaną przez to wywołanie, lub przez ostatnie wcześniejsze wywołanie **write()**,
- w przeciwnym przypadku (wywołanie **read()** jest współbieżne z wieloma wywołaniami **write()**) wartość zwracana przez **read()** jest dowolna.

Skonstruuj M-wartościowy 1-regularny rejestr SRSW używając $O(\log M)$ Boolowskich regularnych rejestrów SRSW. Uzasadnij poprawność swojej konstrukcji.

Def. Rejestr nazwiemy **dobrym**, jeśli dla każdego ciągu współbieżnych dostępów do tego rejestru (zapisów i odczytów) każda wartość odczytana występuje wśród wartości zapisanych (tzn. wartości odczytane nie biorą się "z powietrza").

Zadanie 9. Mamy dobry rejestr typu MRSW. Dla dowolnej historii współbieżnych dostępów do tego rejestru, porządek zapisów jest jednoznacznie wyznaczony (bo jest tylko jeden wątek zapisujący). Możemy więc wszystkie zapisy ponumerować, oznaczając i-ty zapis przez W^i. Przez R^i oznaczmy odczyt rejestru, który zwrócił wartość zapisaną tam przez zapis W^i. Zauważmy, w każdym ciągu dostępów do rejestru może być co najwyżej jeden zapis W^i, oraz że istnieją takie ciągi dostępów, w których występuje wiele odczytów R^i. Pokaż, że dobry rejestr MRSW jest **regularny** wtedy i tylko wtedy, gdy dla każdego ciągu współbieżnych dostępów:

dla każdego i nie jest prawdą, że $R^i \rightarrow W^i$ (*), oraz

dla każdych i oraz j nie jest prawdą, że $W^i \rightarrow W^j \rightarrow R^i$ (**).

Strzałka $\rightarrow$ oznacza relację *happens before* na metodach odczytu i zapisu rejestru.