Programowanie współbieżne

Lista zadań nr 2

Na ćwiczenia 26 października 2022

Uwaga: W poniższych zadaniach tam, gdzie to możliwe przeprowadź formalne rozumowania z użyciem relacji →.

Zadanie 1. Poniższy algorytm ma w zamierzeniu implementować interfejs **Lock** dla dowolnej liczby n wątków. Czy ten algorytm spełnia warunek a) wzajemnego wykluczania (ang. *mutual exclusion*), b) niezagłodzenia (ang. *starvation-freedom*), c) niezakleszczenia (ang. *deadlock-freedom*)? Zmiennymi współdzielonymi przez wątki są `turn` i `busy`.

```
class Foo implements Lock {
    private int turn;
    private boolean busy = false;
    public void lock() {
        int me = ThreadID.get(); /*get id of my thread*/
        do {
            do {
                turn = me;
            } while (busy);
            busy = true;
        } while (turn != me);
    }
    public void unlock() {
        busy = false;
    }
}
```

Zadanie 2. Rozważmy wariant algorytmu Petersena, w którym metodę **unlock()** zmieniliśmy na poniższą. Czy ten algorytm spełnia warunek a) niezakleszczenia, b) niezagłodzenia?

```
public void unlock() {
    int i = ThreadID.get(); /*returns 0 or 1*/
    flag[i] = false;
    int j = 1 - i;
    while (flag[j] == true) {}
}
```

Zadanie 3. Przypomnij, co to znaczy że algorytm ma własność r-ograniczonego czekania (ang. *r-Bounded Waiting*). Czym są sekcja wejściowa (ang. *doorway section*) i sekcja oczekiwania (ang. *waiting section*) w algorytmie Petersena? Pokaż, że ten algorytm ma własność 0-ograniczonego czekania, tzn, że jest FCFS (*First Come First Served*).

Zadanie 4. Rozważmy algorytm tree-lock będący generalizacją algorytmu Petersena dla dowolnej liczby n wątków, będącej potęgą 2. Tworzymy pełne drzewo binarne o n/2 liściach, w każdym węźle drzewa umieszczamy zamek obsługiwany zwykłym algorytmem Petersena. Wątki przydzielamy po dwa do każdego liścia drzewa. W metodzie **lock()** algorytmu tree-lock wątek musi zająć każdy zamek na drodze od swojego liścia do korzenia drzewa. W metodzie **unlock()** algorytm zwalnia wszystkie zajęte wcześniej zamki, w kolejności od korzenia do liścia. Czy ten algorytm spełnia warunek a) wzajemnego wykluczania, b) niezagłodzenia, c) niezakleszczenia? Każdy z zamków traktuje jeden z rywalizujących o niego wątków jako wątek o numerze 0 a drugi jako wątek 1.

Zadanie 5.

1. Czy istnieje taka liczba r, być może zależna od n, że algorytm tree-lock spełnia własność r-ograniczonego czekania (ang. *r-Bounded Waiting*)? Jako sekcję wejściową (ang. *doorway section*) algorytmu przyjmij fragment kodu przed pętlą while zamka w odpowiednim liściu.
2. Pokaż, być może modyfikując nieco oryginalny algorytm, że założenie o numerach wątków w poprzednim zadaniu może być łatwo usunięte.

Zadanie 6. W dobrze zaprojektowanym programie wielowątkowym rywalizacja o zamki powinna być niewielka. Najbardziej praktyczną miarą wydajności algorytmu implementującego zamek jest więc liczba kroków potrzebna do zajęcia wolnego zamku, jeśli tylko jeden wątek się o to ubiega. Poniższy kod jest propozycją uniwersalnego wrappera, który ma przekształcić dowolny zamek na zamek wykonujący tylko stałą liczbę kroków w opisanym przypadku. Czy ten algorytm spełnia warunek a) wzajemnego wykluczania, b) niezagłodzenia, c) niezakleszczenia? Załóż, że oryginalny zamek spełnia te warunki.

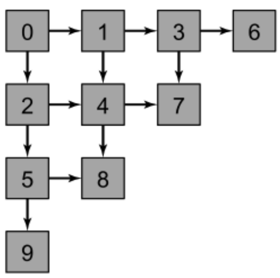
```
class FastPath implements Lock {
    private Lock lock;
    private int x, y = -1;
    public void lock() {
        int i = ThreadID.get();
        x = i;                      // I'm here
        while (y != -1) {}          // is the lock free?
        y = i;                      // me again?
        if (x != i)                 // Am I still here?
            lock.lock();            // slow path
    }
```

```
    public void unlock() {
        y = -1;
        lock.unlock();
    }
}
```

Zadanie 7. Pewna liczba n wątków wywołuje współbieżnie metodę **visit()** klasy Bouncer podanej poniżej. Pokaż, że a) co najwyżej jeden wątek otrzyma jako wartość zwracaną STOP, b) co najwyżej n-1 wątków otrzyma wartość DOWN, c) co najwyżej n-1 wątków otrzyma wartość RIGHT.

```
class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private boolean goRight = false;
    private int last = -1;
    int visit() {
        int i = ThreadID.get();
        last = i;
        if (goRight)
            return RIGHT;
        goRight = true;
        if (last == i)
            return STOP;
        else
            return DOWN;
    }
}
```

Zadanie 8. Czasami zachodzi potrzeba przypisania wątkom unikalnych identyfikatorów w postaci niedużych liczb całkowitych. W tym celu obiektom klasy Bouncer nadaje się numery i ustawia tworząc graf jak na rysunku poniżej.



Każdy z n wątków współbieżnie wykonuje następującą procedurę. Najpierw odwiedza obiekt 0 (wywołuje jego funkcję visit()). Jeśli otrzyma wynik STOP to na tym kończy, jeśli RIGHT to odwiedza obiekt 1, jeśli DOWN to odwiedza 2. W ogólności, będąc w dowolnym obiekcie wątek kończy, idzie na prawo lub na dół w zależności od wartości zwróconej przez visit(). Każdy wątek otrzymuje identyfikator tego obiektu, w którym zakończył działanie.

1. Pokaż, że to w istocie nastąpi, tzn. że każdy wątek otrzyma kiedyś wynik STOP. Załóż, że graf jest nieograniczony.
2. Pokaż, że liczbę wierzchołków grafu można ograniczyć. Znajdź ich dokładną liczbę.