Programowanie współbieżne

Lista zadań nr 3

Na ćwiczenia 23 i 24 października 2024

wersja finalna

Zadanie 1. (J. Burns, L. Lamport). Istnieje niezakleszczający algorytm implementujący zamek, działający dla n wątków i wykorzystujący dokładnie n bitów współdzielonej pamięci. Pokaż, że poniższa implementacja spełnia te warunki, tj. warunek wzajemnego wykluczania i niezakleszczenia. Czy spełnia również warunek niezagłodzenia?

```
class OneBit implements Lock {
    private boolean[] flag;

    public OneBit (int n) {
        flag = new boolean[n]; // all initially false
    }

    public void lock() {
        int i = ThreadID.get(); // ThreadID.get() returns 0,1,..,n-1
        do {
            flag[i] = true;
            for (int j = 0; j < i; j++) {
                if (flag[j] == true) {
                    flag[i] = false;
                    while (flag[j] == true) {} // wait until flag[j] == false
                    break;
                }
            }
        } while (flag[i] == false);
        for (int j = i+1; j < n; j++) {
            while (flag[j] == true) {} // wait until flag[j] == false
        }
    }

    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

Zadanie 2 (2pkty). Poprzednie zadanie pokazało, że n współdzielonych bitów *wystarczy* do implementacji zamka dla n wątków. Okazuje się, że to ograniczenie jest *dokładne*, czyli że n współdzielonych bitów jest *koniecznych* do rozwiązania tego problemu. Udowodnij to twierdzenie.

Wskazówka: Rozdział 2.9 w "The Art of Multiprocessor Programming" 2e.

To twierdzenie ma ważną implikację: zamki oparte wyłącznie na zapisie/odczycie współdzielonej pamięci są nieefektywne, dlatego potrzebne jest wsparcie ze strony sprzętu i systemu operacyjnego.

Definicja 1. Formalną definicję **linearyzacji** można streścić w dwóch punktach:

1. Ciąg wywołania metod w programie współbieżnym powinien mieć taki efekt, jak gdyby te metody zostały wykonane w pewnym sekwencyjnym porządku, jedna po drugiej.
2. Efekt każdej metody w programie współbieżnym powinien wystąpić w pewnym punkcie czasu pomiędzy jej wywołaniem a powrotem (punkt linearyzacji).

Zadanie 3. Sprawdź, czy poniższe diagramy reprezentują linearyzowalne historie. Użyj nieformalnej definicji linearyzacji z Definicji 1.

Zadanie 4. Powtórz zadanie 3, tym razem używając formalnej definicji linearyzacji (slajd 132). Dla każdego diagramu zdefiniuj odpowiadającą mu historię H. Jeśli to możliwe, zdefiniuj historię G oraz legalną sekwencyjną historię S spełniające warunki z definicji.

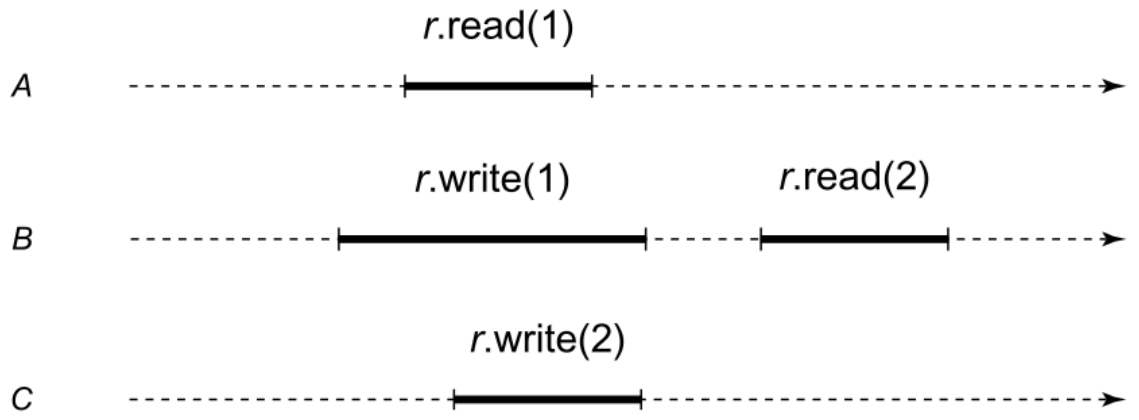

Diagram 1 (zapis/odczyt współdzielonej komórki pamięci r)

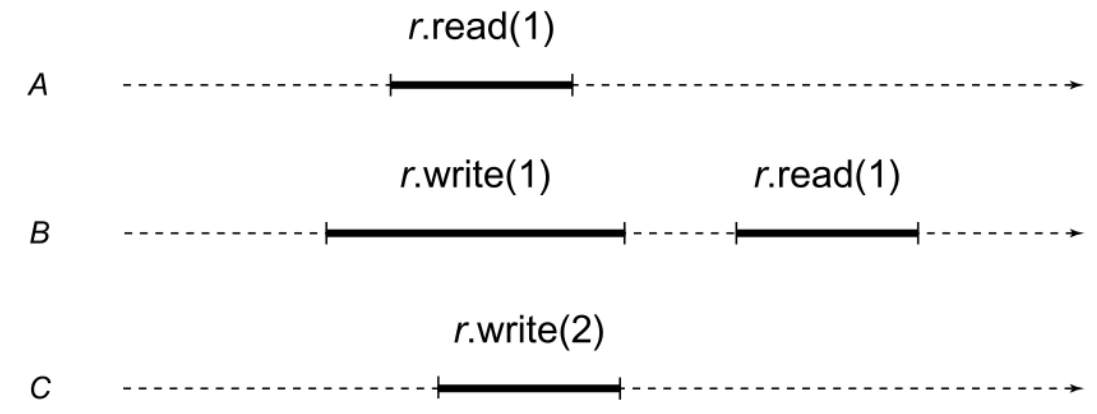

Diagram 2 (zapis/odczyt współdzielonej komórki pamięci r)

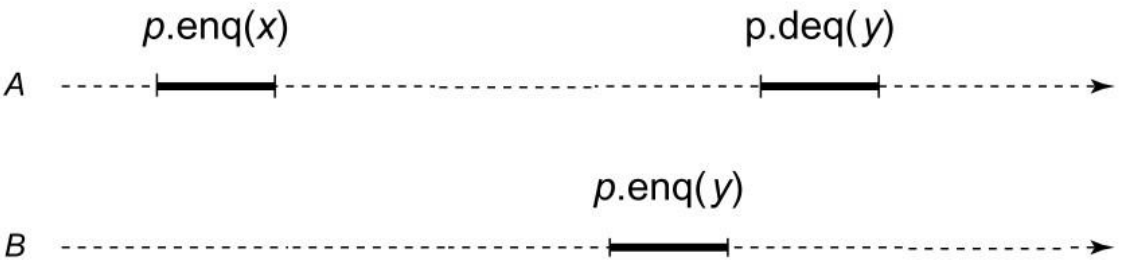

Diagram 3 (p jest kolejką FIFO)

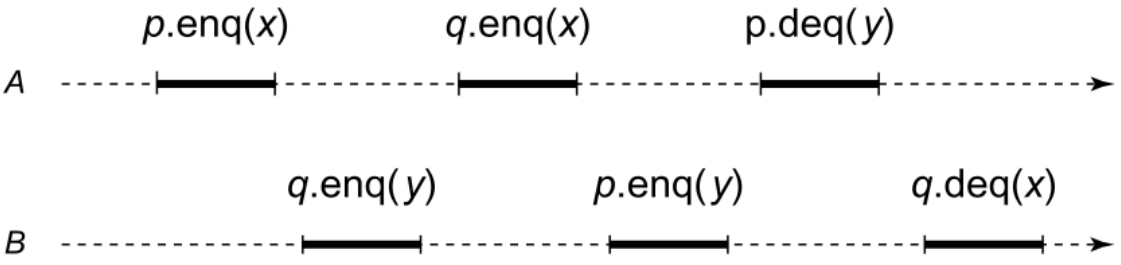

Diagram 4 (p i q są kolejkami FIFO)

Zadanie 5 (fork/join). Plik RookieMergeSort.java zawiera wielowątkową implementację algorytmu sortowania przez scalanie. Taki wzorzec wykorzystania wątków nosi nazwę fork/join.

1. Zrefaktoryzuj ten kod (przenieś fragmenty kodu do nowych metod, zmień nazwy zmiennych tak by zwiększyć jego czytelność) przy okazji weryfikując poprawność implementacji.

2. Czy synchronizacja za pomocą wewnętrznych zamków jest w nim niezbędna? Jeśli nie, to wprowadź odpowiednie poprawki.
3. Program wykonuje wiele alokacji pomocniczych tablic, co jest zbędne. Wprowadź poprawki tak, by alokować tylko jedną pomocniczą tablicę, współdzieloną między wątkami. Zadbaj, by modyfikacje nie wiązały się z koniecznością wprowadzenia dodatkowej synchronizacji.

Zadanie 6 (fork/join). Kontynuacja zadania 5.

1. Sortowanie małych tablic w osobnych wątkach jest nieefektywne. Zmodyfikuj program z poprzedniego zadania tak, by tablice nieprzekraczające pewnego zadanego rozmiaru były sortowane w jednym wątku.
2. Czy utworzenie dwóch nowych wątków w celu posortowania podtablic jest konieczne? Uzasadnij to lub wprowadź modyfikacje w której do rekurencyjnego sortowania dwóch połówek tablicy używa się tylko jednego nowego wątku.

Zadanie 7 (fork/join ze stałą liczbą wątków). Zmodyfikuj program (z zadania 5 lub z zadania 6, jak wolisz) tak, by używał nie więcej niż M wątków, gdzie M jest stałą. Każdy wątek po stwierdzeniu, że osiągnięto maksymalną liczbę utworzonych wątków powinien sortować zadaną tablicę szeregowo. Pamiętaj o właściwej synchronizacji współdzielonych zasobów.

Zadanie 8 (fork/join). Napisz wielowątkowy program, który w zadanej tablicy liczb całkowitych wyszuka najdłuższy spójny podciąg wystąpień tej samej liczby. Jeśli takich podciągów jest wiele, to wystarczy znaleźć jeden. Np. dla ciągu [1,2,1,2,1,2,1,2,3,3,3] wynik to [3,3,3], a dla [1,2,3,3,4,1] wynik to [3,3]. Wynik należy wypisać na konsoli.

Zadanie 9. Poniżej znajduje się "zoptymalizowany" wariant algorytmu piekarni, w którym etykietę wątku pobiera się ze współdzielonej zmiennej `counter`. Czy ten algorytm spełnia warunki a) wzajemnego wykluczania, b) niezakleszczenia, c) niezagłodzenia?

```
class BakeryOpt implements Lock {
   boolean flag[];
```

```
    int label[];
    int counter;

    public Bakery(int n) {
        flag = new boolean[n];
        label = new Label[n];
        counter = 0;

        for (int i = 0; i < n; i++) {
            flag[i] = false;
            label[i] = 0;
        }

        public void lock () {
            flag[i] = true; // i – numer bieżącego wątku
            label[i] = counter++;

            while (∃k flag[k] && (label[i], i) > (label[k], k));

        }

        public void unlock () {
            flag[i] = false;
        }
    }
}
```

Wskazówka: Zauważ, że podczas działania algorytmu wartość zmiennej `counter` może zmniejszyć się. Instrukcja `counter++` nie jest atomowa!