Programowanie współbieżne

Lista zadań nr 3

Na ćwiczenia 9 listopada 2022

Zadanie 1 (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 2 (fork/join). Kontynuacja zadania 1.

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 3 (fork/join ze stałą liczbą wątków). Zmodyfikuj program (z zadania 1 lub z zadania 2, 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 4 (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 5 (pula wątków). Napisz wielowątkowy program sortujący przez scalanie tablicę liczb typu **int** w następujący sposób. Na początku stwórz pewną stałą liczbę M wątków roboczych oraz początkowo pustą kolejkę FIFO, która będzie przechowywać zadania dla wątków. Zadaniem są informacje np. "posortuj podtablicę [l,r] wejścia", "scal podtablice [l,m] i [m+1, r]". Każdy wątek powinien próbować skonsumować zadanie z kolejki, wykonać je oraz ewentualnie wyprodukować następne zadania i dodać je do kolejki. Po czym kroki te powtórzyć. Jako kolejki użyj **java.util.concurrent.ConcurrentLinkedQueue<>**[1] implementującej interfejs **java.util.Queue<>**. Najważniejsze operacje tego interfejsu to **offer(e)** i **pool()**. Ta kolejka jest **wątkowo bezpieczna** (użycie jej w programie wielowątkowym nie wymaga dodatkowej synchronizacji), **nieograniczona** (rozmiar nie jest zadany z góry) oraz **nieblokująca** (operacja **poll()** wywołana na pustej kolejce zwraca **null** zamiast zablokować się w oczekiwaniu na element). Zastanów się, w jaki sposób wątki wykryją, że nie ma więcej zadań do wykonania i należy zakończyć pracę. Samo stwierdzenie, że kolejka jest pusta nie wystarczy, gdyż na początku działania programu może być mniej zadań, niż wątków chcących je wykonać.

Zadanie 6. Wariant zadania poprzedniego. Tym razem jako kolejki użyj **java.util.concurrent.LinkedBlockingQueue<>**[2] implementującą interfejs **java.util.concurrent.BlockingQueue<>**. Najważniejsze operacje tego interfejsu to **put(e)** i **take()**. Ta kolejka jest **wątkowo bezpieczna**, **nieograniczona** oraz **blokująca** (operacja **take()** wywołana na pustej kolejce usypia wątek w oczekiwaniu na element). Które rozwiązanie, to czy z zadania poprzedniego, jest bardziej wydajne?

Uwaga: sama podmiana kolejki nieblokującej na blokującą może nie dać poprawnego rozwiązania. Zastanów się, czy warunek zakończenia pracy wątków nadal jest poprawny!

Zadanie 7. (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.

---

[1] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html

```
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 8 (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.