Programowanie współbieżne

Lista zadań nr 6

Na ćwiczenia 13 i 14 listopada 2024

Zadanie 1. Mamy dobry[1] rejestr typu MRMW. Zapisom i odczytom z tego rejestru nadajemy unikalną numerację. Dla dowolnej historii współbieżnych dostępów do tego rejestru, przez W^i oznaczamy punkt w czasie, w którym nastąpił efekt i-tego zapisu, a przez R^i oznaczmy punkt w czasie, w którym nastąpił efekt odczytu, który zwrócił wartość zapisaną tam przez zapis W^i. Pokaż, że dobry rejestr typu MRMW jest **atomowy** wtedy i tylko wtedy gdy każda historia dostępów do tego rejestru spełnia warunki:

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

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

dla dla każdych i oraz j jeśli $R^i \to R^j$ to $i \leq j$ (***).

Strzałka $\to$ oznacza relację *happens before*.

Zadanie 2. Pokaż poprawność konstrukcji regularnego M-wartościowego rejestru MRSW używającej regularnych rejestrów Boolowskich MRSW.

Wskazówka: The Art of Multiprocessor Programming 2e, rozdział 4.2.3.

Zadanie 3. Pokaż poprawność konstrukcji atomowego rejestru MRSW używającej atomowych rejestrów SRSW.

Wskazówka: TAoMP 2e, rozdział 4.2.5.

Zadanie 4. Pokaż poprawność konstrukcji atomowego rejestru MRMW używającej atomowych rejestrów MRSW.

Wskazówka: TAoMP 2e, rozdział 4.2.6.

[1] Definicja dobrego rejestru: patrz zadanie 8 z listy 5.

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<>**[2] 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<>**[3] 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!

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