Teoria współbieżności

Laboratorium 2

Wyścig - wyjaśnienie

1. Licznik
2. Metoda inkrementacji w Bytecode

```
3. public void inc();
4.   Code:
5.    0:   aload_0
6.    1:   dup
7.    2:   getfield        #2; //Field i:I
8.    5:   iconst_1
9.    6:   iadd
10.   7:   putfield        #2; //Field i:I
11.   10:  return
12.}
```

   - 0: załadowanie na stos argumentu o indeksie 0 (niejawny argument `this`)
   - 2: wartość pola o referencji #2 jest pobierana i umieszczana na stosie
   - 5: umieszcza stałą całkowita '1' na stosie
   - 6: dodaje do siebie dwa górne elementy stosu (muszą być typu integer, są zdejmowane ze stosu) i umieszcza na nim wynik dodawania

Monitory w javie

1. Związane z każdym obiektem
2. Same nie są reprezentowane jako klasy / obiekty!
3. Synchronizują dostęp do metod i bloków *synchronized*
4. Dodatkowy monitor związany z klasą - dla metod statycznych synchronizowanych

Synchronizacja

1. Metody *synchronized*
2. Blok *synchronized*
   - użyteczny do synchronizacji dostępu do już istniejących obiektów, do których nie można dodać żadnych metod

```
3. synchronized ( obj ) {
4.     ...
5.   obj.method();
6.     ...
7. }
```

8. Zasady synchronizacji
   - Wejście do metody synchronizowanej (bloku synchronizowanego) powoduje *zajęcie monitora* (*którego?*), opuszczenie jej - *zwolnienie monitora*
   - Metoda synchronizowana (blok synchronizowany) może być wykonywana na rzecz danego obiektu tylko przez wątek będący w posiadaniu monitora tego obiektu
   - Tylko jeden wątek może być w posiadaniu monitora danego obiektu, a

po jego zajęciu tylko on sam może go zwolnić

Metody wait, notify / notifyAll

1. Zdefiniowane w klasie Object
2. Koordynacja działania wielu wątków
3. wait
    - może być wywołana tylko przez wątek będący w posiadaniu monitora
    - powoduje uśpienie wątku w kolejce związanej z monitorem (*którym?*)
    - zwalnia monitor
    - wątek może być obudzony tylko gdy inny wątek wywoła notify (*jaki to musi być wątek?*)
4. notify
    - budzi jeden wątek spośród oczekujących w kolejce 'wait' (*której kolejce?*)
    - obudzony wątek oczekuje aż wątek wywołujący `notify` zwolni monitor
    - `notifyAll` działa jak `notify`, ale budzi wszystkie wątki

Krytyka "Monitorów" Javy

1. Brinch Hansen - pionier programowania współbieżnego (koncepcja Monitora, Concurrent Pascal): Java ożywia koncepcje sprzed ćwierć wieku i robi to źle.
2. Hoare - pojęcie bezpieczeństwa (security) języka (z punktu widzenia współbieżności / równoległości): *język powinien umożliwić kompilatorowi i środowisku uruchomieniowemu wykrywanie jak największej ilości przypadków, w których koncepcje języka się załamują i produkują bezsensowne wyniki. Dla języka programowania równoległego najważniejszą miarą bezpieczeństwa jest sprawdzenie, czy procesy operują na rozłącznych zbiorach zmiennych i że nie kolidują ze sobą w sensie zależności czasowych* (np. rezultat zależy od relatywnej kolejności wykonania instrukcji przez dwa procesy).
3. Np. kompilator Concurrent Pascala sprawdza, czy każdy proces i monitor odwołuje się tylko do swoich zmiennych, że procesy komunikują się wyłącznie przez procedury monitora, i że procesy nie zablokują się poprzez rekursywne wywołanie procedury monitora.
4. Z kolei interpreter Concurrent Pascala zapewnia wzajemne wykluczanie wszystkich operacji na zmiennych jakiegokolwiek procesu lub monitora. Uniemożliwia nawet jednoczesny dostęp do zmiennej przez proces i urządzenie peryferyjne.
5. Brinch Hansen twierdzi, że tak naprawdę Java nie wspiera koncepcji monitorów, gdyż:
    - Metody javy są domyślnie niesynchronizowane, chyba że są zadeklarowane jako `synchronized`.
    - Zmienne klasy są publiczne (w obrębie pakietu), jeśli nie są zadeklarowane jako `private`.
6. To nie uniemożliwia całkowicie pisania poprawnych programów równoległych, ale uniemożliwia wykrywanie wielu błędów zależnych od czasu (*time-dependant*) na poziomie kompilacji i uruchomienia.
7. Błąd Javy: decyzja, żeby używać sekwencyjnej części języka do implementacji wsparcia dla równoległości. Java nie jest językiem "bezpiecznym".

Definicja operacji na semaforze

- Opuszczenie, s.P(): Jeśli S==1 to S=0, w przeciwnym wypadku wstrzymaj działanie procesu wykonującego tę operację.
- Podniesienie, s.V(): Jeśli są procesy wstrzymane w wyniku opuszczania semafora S, to wznów jeden z nich, w przeciwnym razie S=1.

Zadania

1. zaimplementowac semafor binarny za pomoca metod wait i notify, uzyc go do synchronizacji programu *Wyscig*
2. pokazac, ze do implementacji semafora za pomoca metod wait i notify nie wystarczy instrukcja *if* tylko potrzeba uzyc *while* . Wyjasnic teoretycznie dlaczego i potwierdzic eksperymentem w praktyce. (wskazowka: rozwazyc dwie kolejki: czekajaca na wejscie do monitora obiektu oraz kolejke zwiazana z instrukcja *wait* , rozwazyc kto kiedy jest budzony i kiedy nastepuje wyscig).
3. Zaimplementowac semafor licznikowy (ogolny) za pomoca semaforow binarnych. Czy semafor binarny jest szczegolnym przypadkiem semafora ogolnego ?

Bibliografia

1. Per Brinch Hansen: Java Insecure Parallelism. ACM SIGPLAN Notices, April 1999, s. 38-45.
2. Thomas W. Christopher, George K. Thiruvathukal: High-Performance Java Platform Computing, Prentice Hall, Luty 2001. Wybrane rozdziały.