Teoria Współbieżności
Ćwiczenie 1

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie studentów z wątkami oraz synchronizacją w Java.

2 Wątki w JVM

1. Architektura JVM: Chapter 5 of Inside the Java Virtual Machine by Bill Venners.
2. Cykl życia wątku, stany wątku
   - New
   - Runnable
   - Running
   - Not Running
   - Dead
3. Szeregowanie wątkow w Javie
   (a) Ogólne pojęcia szeregowania: wywłaszczania (*preemption*), podziału czasu (*time-slicing*, *time-sharing*), priorytety.
   (b) Dokładne zachowanie się wątków jest zależne od platformy.
   (c) Wątki mogą być zaimplementowane całkowicie w przestrzeni użytkownika lub korzystać z natywnych interfejsów platformy.

(d) Wątkom można przypisać priorytety (1-10). Jedyną gwarancją jest to, że wątkom o najwyższym priorytecie zostanie przydzielony CPU. Wątki o niższym priorytecie mają gwarancję przydziału CPU tylko wtedy, gdy wątki z wyższym priorytetem są zablokowane, w przeciwnym wypadku nie ma tej gwarancji.

(e) Specyfikacja nie zakłada podziału czasu.

(f) Operacje odczytu i zapisu danych typów prostych (primitives) pomiędzy pamięcią główną a roboczą pamięcią wątku są atomowe. Jedynym wyjątkiem mogą być operacje na 64-bitowych typach long i double, które mogą być zrealizowane jako dwie operacje 32-bitowe.

(g) Reguły szeregowania

- W każdym momencie, spośród kilku wątkow w stanie RUNNABLE wybierany jest ten o najwyższym priorytecie
- Wykonywany wątek może być wywłaszczony, jeśli pojawi się wątek o wyższym priorytecie, gotowy do wykonania
- Jeśli wiele wątków ma ten sam priorytet, wybierany jest jeden z nich, wg kolejności (round-robin)
- Na niektórych systemach może być zaimplementowany podział czasu (wątki sa wywłaszczane po upływie kwantu czasu)

3 Java niezbędnik

1. Klasa Thread

```
class MojWatek extends Thread {
  public void run() {
    //KOD WATKU
  }
  //...
}
...
MojWatek mw = new MojWatek(); //
mw.start(); //

// mw.join(); czekmy na zakonczenie
```

2. Interfejs *Runnable*

```
class MojWatek implements Runnable {
  public void run() {
    //KOD WATKU
  }
  //...
}
...
MojWatek mw = new MojWatek();
Thread t = new Thread(mw);
t.setName("Jacek"); // nazwanie watku
// a w watku Thread.currentThread().getName()
// odczytanie nazwy watku
t.setPriority(1); // ustaw priorytet
// Thread.currentThread().getPriority()
t.start();
```

3. API JavaDoc

4. Anonimowa klasa wewnętrzna:

```
new Thread() {
  public void run() {
    for(;;) System.out.println("Thread 1");
  }
}.start();
new Thread() {
  public void run() {
    for(;;) System.out.println("Thread 2");
  }
}.start();
```

5. Każdy obiekt w Javie może być użyty do ryglowania (jako monitor) - *synchronized* .

```
// dla funkcji - rownowazne z synchronized(this)
synchronized ...  foo(){
        // sekcja krytyczna
}
//albo blok instrukcji
synchronized(obj){
        // sekcja krytyczna
}
```

4 Termin „wyścig”

Więcej niż jeden wątek korzysta jednocześnie z zasobu dzielonego, przy czym co najmniej jeden próbuje go zmienić. Przyczyna niedeterministycznego zachowania się programu. Może prowadzić do trudnych do wykrycia błędów.

5 Ćwiczenia

Zadanie teoretyczne

W systemie działa N wątkow, które dzielą obiekt licznika (początkowy stan licznika $S = 0$). Każdy wątek wykonuje w pętli 5 razy inkrementację licznika S.
Zakładamy, że inkrementacja składa się z sekwencji trzech instrukcji i każda jest atomowa:

- **read** - odczyt z pamięci,
- **inc** - zwiększenie o 1
- **write** - zapis do pamięci

Zmienne globalne
S ← 0
Kod każdego wątku
W0: for i ← 1..5 :
W1: tmp ← S
W2: tmp++
W3: S ← tmp

Wątki nie są synchronizowane.

5.1 Zadania implementacyjne

1. „Rewolwerowiec”. Sekcja lokalna i wątki.
   (a) Napisz wątek odliczajacy do wystrzału: “5” , “4” , ... , “Pif! Paf!”. Uruchom kilka rewolwerowców na raz.
   (b) Rozszerz klasę wątków aby odliczały od pewnej zadanej liczby.

(c) Prosto zmodyfikuj kod zadania, zrób tak aby reszta wątków automatycznie kończyła odliczanie gdy conajmniej jeden z nich strzeli.

2. „Dodawanie i sekcja krytyczna”: w poniższym zadaniu wypisz jaka jest wartość inta po zakończeniu działania pierwszego i drugiego wątku (zaraz po skończeniu swojej inkrementacji).

   (a) Napisz klasę posiadającą prywatnego inta i metodę inkrementyującą o 1. Przekaż tą klasę do dwóch wątków które będą ją wiele razy inkrementować - przetestuj dla 100, 1k, 10k, 100k, 1kk.

   (b) Używając *synchronized* popraw poprzedni kod. Uzasadnij, że rzeczywiscie działa poprawnie po każdym wątku.

   (c) Zmodyfikuj kod z poprzedniego zadania: ustaw zmienną int na static i w pętli zrób dwie metody inkrementujące wykonywane zaraz po sobie. Jedna taka jak wcześniej (z podpunkut (b)) a druga *synchronized* i *static* ! Wytłumacz wynik.

3. „Operacje atomowa?”. Używając wątków:

   (a) pokaż ekperymentalnie, że operacja zapisu do *float*ów (albo *double*i) w Javie nie jest atomowa. Podpowiedź, można użyć *Double.MAX_VALUE* , *Double.MIN_NORMAL* oraz 0.

   (b) sprawdz czy operacja zapisu stringów jest atomowa.

4. Semafory*

   (a) * Napisz klase ktora dziala jak semafor binarny, pokaż że działa (np w zadaniu 3). Trzeba użyć *wait* i *notify*.

```
1  class Semafor {
2      private boolean stan = true;
3      private int czeka = 0;
4
5      public Semafor() {
6      }
7
8      public synchronized void P() {
9
10     }
11
12     public synchronized void V() {
13
```

```
14      }
15  }
```

(b) * Napisz klasę która działa jak semafor licznikowy, pokaż, że działa (np w zadaniu 1). Trzeba użyć *wait* i *notify*.

5. Zasymuluj:

   (a) komitety liczące głosy parlamentalne. Każdy komitet (wątek) liczy swoją zadaną liczbę głosów i przy okazji updateuje główny licznik głosów. Pokaż działanie komitetów, na przyład: po podliczeniu wszystkich swoich każdy komitet podaje ile wyliczył głosów i jaki jest główny stan głosów.

   (b) uproszczone losowanie Lotto/Bingo. Jeden wątek (gracz) losuje na bieżąco liczby a reszta uczetników sprawdza czy wygrała. Gdy chociaż jeden gracz wygra zabawa (program) się kończy. Użyj kodu:

```
1  class Bingo implements Runnable {
2     private static List<Bingo> gracze = new
          ArrayList<Bingo>() ;
3     ...
4  }
```

6. Napisz kod w którym dojdzie do zakleszczenia, pokaż to eksperymentalnie. Sekcja krytyczna nie musi wykonywać jakielkwoliek operacji - wystarczy *System.out.print("Sekcja krytyczna")*. Możesz użyć (pseudo)kodu z poprzednich zajęć.

Literatura

[1] Bill Venners, Inside the Java Virtual Machine (rozdział 5, rozdział 20), McGraw-Hill Companies; 2nd Bk&Cdr edition, 2000.

[2] Bruce Eckel, "Thinking in Java" - rozdział o wątkach

[3] Bartosz Baliś, Teoria Współbieżności, materiały własne

[4] Maciej Woźniak, Teoria Współbieżności, materiały własne

Uwagi: sluzalec@agh.edu.pl