Problem wzajemnego wykluczania

1. W programie dane są dwa wątki: *Alicja* i *Robert*, które jednocześnie (współbieżnie) modyfikują n razy współdzielony licznik – pierwszy z nich zwiększa go o 1, a drugi zmniejsza o 1. Twoim zadaniem jest zaimplementować asymetryczny protokół wzajemnego wykluczania dla tych wątków. Wzajemne wykluczanie dla wątku *Alicja* działa wg protokołu:

   a) podnieś flagę,

   b) czekaj, aż Robert opuści swoją,

   c) modyfikuj licznik,

   d) opuść flagę,

   gdzie podnoszenie flagi polega na ustawieniu zmiennej ulotnej typu logicznego na wartość `true`.

   Robert działa wg (asymetrycznego) protokołu:

   a) podnieś flagę,

   b) dopóki flaga Alicji jest podniesiona:

      - opuść flagę
      - czekaj, aż Alicja opuści flagę
      - podnieś flagę;

   c) modyfikuj licznik,

   d) opuść flagę.

   Protokół powinien zostać zaimplementowany w klasie `ARLock`:

```
class ARLock {
    volatile boolean flagaAlicji;
    volatile boolean flagaRoberta;

    public void lockAlicja() {
        ...
    }

    public void unlockAlicja() {
        ...
    }

    public void lockRobert() {
        ...
    }
```

```
    public void unlockRobert() {
        ...
    }
```

gdzie metody `lockAlicja()` i `lockRobert()` zawierają protokół wstępny przedstawionego protokołu wzajemnego wykluczania, natomiast metody `unlockAlicja()` i `unlockRobert()` zawierają protokoły końcowe.

Klasa wątku *Alicja* powinna wyglądać następująco:

```
class Alicja extends Thread {
    ARLock lock;
    Licznik licznik;
    int n;

    ...

    @Override
    public void run() {
        for (int i = 0; i < n_; ++i) {
            lock.lockAlicja();
            licznik.dodaj(+1);
            Thread.yield();
            lock.unlockAlicja();
        }
    }
}
```

Klasa wątku *Robert* powinna wyglądać analogicznie.

Przykładowe zastosowanie zaimplementowanych powyżej klas wątków oraz protokołu to:

```
public class AlicjaRobert {

    public static void main(String[] args) throws InterruptedException {
        Licznik licznik = new Licznik();
        ARLock lock = new ARLock();
        final int n = 1000000;
        Thread alicja = new Alicja(licznik, lock, n);
        Thread robert = new Robert(licznik, lock, n);

        alicja.start();
        robert.start();

        alicja.join();
        robert.join();

        System.out.println("Końcowa wartość licznika: " + licznik.getValue());
    }
}
```

Program powinien wypisać na ekranie wartość 0.

2. Zaimplementuj algorytm Petersona dla wzajemnego wykluczania dwóch wątków i przetestuj go w problemie zwiększania współdzielonego licznika. Wątki pierwszy (id=0) i drugi (id=1) w metodzie `run()` wykonują n razy pętlę polegającą na uzyskaniu dostępu do sekcji krytycznej, zwiększeniu licznika o 1 i wyjściu z sekcji krytycznej.

Algorytm Petersona powinien być implementowany za pomocą klasy `PetersonLock`, której szkic wygląda następująco:

```
class PetersonLock {
    public static final int PIERWSZY = 0;
    public static final int DRUGI = 1;

    public static volatile int czyjaKolej = PIERWSZY;

    /*
    W Javie nie ma ulotnych tablic, stąd konieczność użycia AtomicIntegerArray

    Ustawienie wartości komórki to: chceWejsc.set(indeks, wartosc)
    Odczyt wartości komórki to: chceWejsc.get(indeks)
    */
    public static AtomicIntegerArray chceWejsc = new AtomicIntegerArray(2);

        /* Zakłada blokadę - wątek wywołujący podaje swój identyfikator */
    public void lock(int id) {
        ...
        // teraz wątek jest w sekcji krytycznej
    }

        /* Zwalnia blokadę - wątek wywołujący podaje swój identyfikator */
    public void unlock(int id) {
        ...
        // teraz wątek wyszedł z sekcji krytycznej
        }
}
```

Tablica `chceWejsc` służy do przechowywania informacji o żądaniach wejścia do sekcji krytycznej poszczególnych procesów, tj. chceWejsc.get(0) == 1, jeżeli proces (wątek) od id=0 chce wejść do sekcji krytycznej.

Działanie wątków roboczych opisuje poniższy kod:

```
public void run() {
    for (int i = 0; i < n_; ++i) {
        blokada_.lock(id_);
        licznik_.dodaj(1);
        blokada_.unlock(id_);
    }
}
```

Przykładowy program:

```
public static void main(String[] args) throws InterruptedException {
    final Licznik licznik = new Licznik();
    final PetersonLock blokada = new PetersonLock();
    final int n = 10_000_000;
    final int id1 = 0;
    final int id2 = 1;

    Wykonawca w1 = new Wykonawca(id1, licznik, n, blokada);
    Wykonawca w2 = new Wykonawca(id2, licznik, n, blokada);

    final long startTime = System.currentTimeMillis();
    w1.start();
    w2.start();

    w1.join();
    w2.join();

    final long elapsed = System.currentTimeMillis() - startTime;
```

```
    System.out.println("Końcowa wartość licznika: " + licznik.dajWartosc());
    System.out.printf("Czas działania: %.3f sek.\n", (elapsed / 1000.0));
}
```

Porównaj szybkość działania algorytmu do wersji, w której zastosowano `synchronized`

3. Napisz klasę blokady (zamka) o nazwie TSLock implementującą algorytm wzajmnego wykluczania z wykorzystaniem atomowej instrukcji `getAndSet` z klasy `java.util.concurrent.atomic.AtomicBoolean`. Klasa ta powinna implementować poniższy interfejs:

```
interface MyLock {
    public void lock();
    public void unlock();
}
```

Następnie przetestuj utworzoną klasę dla problemu zwiększania współdzielonego licznika (jak w poprzednim zadaniu). Porównaj czasy działania z wbudowaną konstrukcją `synchronized` oraz poprzednim zadaniem. Czy algorytm działa poprawnie również dla większej liczby wątków ($\geqslant 3$)?

4. Zaimplementuj klasę implementującą blokadę piekarnianą (ang. bakery lock). Bilety zaimplementuj za pomocą `AtomicLongArray`, natomiast flagi za pomocą `AtomicIntegerArray`. Szkielet klasy wygląda następująco:

```
class BakeryLock implements MyLock {
    AtomicIntegerArray flags;
    AtomicLongArray tickets;
    final int threadsCount;
    final static int TRUE = 1;
    final static int FALSE = 0;
    public BakeryLock(int threadsCount) {
        threadsCount = threadsCount;
        flags = new AtomicIntegerArray(threadsCount);
        tickets = new AtomicLongArray(threadsCount);
    }
    public void lock() {
        final int id = ThreadID.get();
        ...
    }
    public void unlock() {
        final int id = ThreadID.get();
        flags.set(id, FALSE);
    }
}
```

Klasa `ThreadID` zwraca unikalny identyfikator wątku. Wątki numerowane są od 0. Przykładowa implementacja klasy to:

```
class ThreadID {
  private static volatile int nextID = 0;
  private static ThreadLocalID threadID = new ThreadLocalID();
  public static int get() {
    return threadID.get();
  }
  public static void reset() {
    nextID = 0;
  }
```

```
  private static class ThreadLocalID extends ThreadLocal<Integer> {
    protected synchronized Integer initialValue() {
      return nextID++;
    }
  }
}
```

Sprawdź efektywność blokady na problemie współbieżnej modyfikacji licznika. Jak zmienia się wydajność wraz ze wzrostem liczby wątków? Sprawdź dla 2, 4 i 8 wątków.

5. Zaimplementuj blokad `TTASLock` opartą na blokadzie `TSLock` z zad. 7., ale zoptymalizowaną pod kątem wykorzystania pamięci podręcznej procesora. Następnie dodaj do blokady wykładnicze wycofywanie, które powoduje uśpienie wątku na losowy okres czasu, w przypadku gdy nie uda mu się założyć blokady. Okres czasu powinien być wydłużany po każdej nieudanej próbie. Porównaj wydajność zaimplementowanych blokad na zadaniu inkrementacji współdzielonego licznika. Jak zmienia się czas działania programu dla różnej liczby wątków – 2, 4, 8?

6. W problemie Producenta-Konsumenta modelowany jest układ dwóch obiektów:

   - obiekt klasy Producent „produkuje" liczby, które umieszcza w buforze o ograniczonej pojemności;
   - obiekt klasy Konsument „konsumuje" liczby, które w buforze umieścił producent;
   - obiekt bufora jest, oczywiście, współdzielony między producentem a konsumentem, natomiast producent nie ma bezpośredniego dostępu do konsumenta i na odwrót.

   Zaimplementuj niezbędne klasy (Producent, Konsument, Bufor) i wykonaj symulację, w której producent produkuje ciąg $N = 20$ losowych liczb całkowitych z zakresu [1, 10]. Liczby te są pobierane przez konsumenta. Pojemność bufora jest ograniczona do 5. W przypadku braku elementów w buforze wątek konsumenta usypiany jest na losowy okres czasu < 100ms. Analogicznie działa producent, gdy bufor jest pełny.

   Do synchronizacji dostępu do bufora zastosować wbudowany mechanizm synchronizacji. Zarówno producent, jak i konsument powinni wypisywać informacje o każdej wyprodukowanej, czy usuniętej liczbie.

   Przykład:

```
Producent: produkuję 6
Producent: produkuję 3
Producent: produkuję 5
Konsument: otrzymałem 6
Producent: produkuję 4
Konsument: otrzymałem 3
Producent: produkuję 4
Konsument: otrzymałem 5
```