Systemy Rozproszone - Ćwiczenie 4

1 Synchronizacja wątków: myjnia samochodowa

Poniżej znajduje się przykład ilustrujący klasyczny problem synchronizacji wątków. Obiekt klasy `Car` jest współdzielony przez 2 wątki. Samochód może znajdować się w jednym w 2 stanów: nawoskowany (waxed==true) i wypolerowany (waxed==false). Na samochodzie działają 2 wątki: `Waxer`, którego zadaniem jest nawoskowanie samochodu, oraz `Polisher`, którego zadaniem jest jest wypolerowanie samochodu. Nawoskowanie samochodu jest możliwe tylko wtedy jeżeli samochód jest wypolerowany (czyli ma stan waxed==false) i skutkuje zmianą jego stanu na waxed==true. Wypolerowanie samochodu jest możliwe tylko wtedy jeżeli samochód jest nawoskowany (czyli ma stan waxed==true) i skutkuje jego zmianą stanu na waxed==false. Obydwa wątki wykonują swoje działania w nieskończonej pętli, a oczekiwanie wątku na osiągnięcie właściwego stanu przez samochód odbywa się w metodach `waitUntilWaxed()` i `waitUntilPolished()`. Wykonaniu działana na samochodzie (metody `polish()` i `wax()`) skutkuje zmianą jego stanu, i wznowieniem pracy przez oczekujący wątek. Zawieszenie pracy wątku uzyskuje się wywołując metodę `Thread.wait()`, a wznowienie pracy wszystkich oczekujących wątków uzyskuje się poprzez wywołanie metody `Thread.notifyAll()`. Wątek główny, po uruchomieniu wątków `Waxer` i `Polisher`, odczekuje 100ms `Thread.sleep()`, a następnie przerywa pracę 2 wątków `Thread.interrupt()`.

```
/* plik: WaxOMatic.java */
class Car {
  boolean waxed = false;

  public synchronized void wax() throws InterruptedException {
    waitUntilPolished();
    waxed = true;
    notifyAll();
  }

  public synchronized void polish() throws InterruptedException {
    waitUntilWaxed();
    waxed = false;
    notifyAll();
  }
```

```
  public synchronized boolean isWaxed() {
    return waxed;
  }

  private synchronized void waitUntilWaxed() throws InterruptedException {
    while (!isWaxed()) {
      System.out.println(Thread.currentThread().getName()
        + " is waiting until waxed");
      wait();
    }
  }

  private synchronized void waitUntilPolished() throws InterruptedException {
    while (isWaxed()) {
      System.out.println(Thread.currentThread().getName()
        + " is waiting until polished");
      wait();
    }
  }
}

class Waxer extends Thread {
  Car car;
  public Waxer(Car c) {
    car = c;
  }
  public void run() {
    try {
      while(true) {
        System.out.println(Thread.currentThread().getName()
          + " is waxing...");
        car.wax();
        System.out.println(Thread.currentThread().getName()
          + " finished waxing");
      }
    }
    catch (InterruptedException e) {
      System.out.println("Waxer interrupted");
    }
  }
}

class Polisher extends Thread {
  Car car;
  public Polisher(Car c) {
```

```
    car = c;
  }
  public void run() {
    try {
      while(true) {
        System.out.println(Thread.currentThread().getName()
          + " is polishing...");
        car.polish();
        System.out.println(Thread.currentThread().getName()
          + " finished polishing");
      }
    }
    catch (InterruptedException e) {
      System.out.println("Polisher interrupted");
    }
  }
}


public class WaxOMatic {
  public static void main(String[] args) throws Exception {
    Car car = new Car();
    Waxer waxer = new Waxer(car);
    Polisher polisher = new Polisher(car);
    waxer.start();
    polisher.start();
    Thread.sleep(100);
    waxer.interrupt();
    polisher.interrupt();
  }
}
```

Wprowadź następujące modyfikacje do programu:

- Dodaj opóznienie do metod `wax()` i polish `polish()` symulujące czas wykonania działania na samochodzie. Jak wpłynie to na działanie programu.
- Wprowadz dodatkowy wątek do programu `Washer`, którego zadaniem jest mycie samochodu, w taki sposób, żeby samochód zmieniał stany w następujący sposób: waxed →polished→washed→waxed→polished→washed→ ....

2 Problem producenta i konsumenta

Kolejny przykład ilustruje klasyczny problem producenta i konsumenta w odniesieniu do dwóch wątków działających na współdzielonym stosie `WaitingStack`.

Pierwszy wątek, `StackPusher`, pełni rolę producenta i jego zadaniem jest wkładanie elementów na stos. Drugi wątek, `StackPopper`, pełni rolę konsumenta i jego zadaniem jest zdejmowania elementów ze stosu. Zwróć uwagę na metody `waitForNotEmpty()`, `waitForFullEmpty()`, `push()`, `pop()`.

```
/* plik: WaitingStack.java */

class WaitingStack {
  int[] data;
  int capacity;
  int current = -1;

  public WaitingStack(int count) {
    capacity = count;
    data = new int[count];
  }

  public synchronized void push(int number) throws InterruptedException
  {
    String name = Thread.currentThread().getName();
    System.out.format("%s: entered push()\n", name);
    waitForNotFull();
    System.out.format("%s: can push()\n", name);
    current++;
    data[current] = number;
    System.out.format("%s pushed %d to [%d]\n", name, number, current);
    notifyAll();
    System.out.format("%s: ended push()\n", name);
  }

  public synchronized int pop() throws InterruptedException
  {
    String name = Thread.currentThread().getName();
    System.out.format("%s: entered pop()\n", name);
    waitForNotEmpty();
    System.out.format("%s: can pop\n", name);
    int value = data[current];
    System.out.format("%s: popped %d from [%d]\n", name, value, current);
    current--;
    notifyAll();
    System.out.format("%s: ended pop()\n", name);
    return value;
  }

  private synchronized void waitForNotEmpty() throws InterruptedException
  {
```

```
    while(current==-1) {
      System.out.println(Thread.currentThread().getName()
        + " is waiting to pop");
      wait();
    }
  }

  private synchronized void waitForNotFull() throws InterruptedException
  {
    while(current==capacity-1) {
      System.out.println(Thread.currentThread().getName()
        + " is waiting to push");
      wait();
    }
  }
}

class StackPusher extends Thread {
  WaitingStack stack;
  int value = 0;
  public StackPusher(WaitingStack ws) {
    stack = ws;
  }
  public void run() {
    try {
      while(true) {
        sleep(100);
        stack.push(++value);
        yield();
      }
    }
    catch (InterruptedException e) {
      System.out.println(Thread.currentThread().getName()
        + " interrupted");
    }
  }
}

class StackPopper extends Thread {
  WaitingStack stack;
  public StackPopper(WaitingStack ws) {
    stack = ws;
  }

  public void run() {
    try {
```

```
      while(true) {
        stack.pop();
        yield();
      }
    }
    catch (InterruptedException e) {
      System.out.println(Thread.currentThread().getName()
        + " interrupted");
    }
  }

}

public class WaitingStackDemo {

  public static void main(String[] args) throws Exception {
    WaitingStack ws = new WaitingStack(10);
    StackPusher pu = new StackPusher(ws);
    StackPopper po1 = new StackPopper(ws);
    StackPopper po2 = new StackPopper(ws);
    StackPopper po3 = new StackPopper(ws);

    pu.start();

    po1.start();
    po2.start();
    po3.start();


    Thread.sleep(1000);
    pu.interrupt();
    po1.interrupt();
    po2.interrupt();
    po3.interrupt();
  }
}
```

3 Bar mleczny "Prawo dżungli"

Rozważ producenta i konsumenta na przykładzie baru mlecznego wydającego posiłki wiecznie głodnym klientom. Producentem jest kucharz umieszczający dania na ladzie, konsumentami są klienci przebywający w barze. Wiecznie głodni klienci, których liczba n jest stała, konsumują wydawane potrawy na zasadzie "kto pierwszy ten lepszy". Zaimplementuj powyższy przykład przy użyciu wątków. Po zakończeniu programu wypisz ile potraw zjadł każdy z klientów.