Laboratorium 4

Łukasz Wala
AGH, Wydział Informatyki, Elektroniki i Telekomunikacji
Teoria Współbieżności 2022/23

Kraków, 12 listopada 2022

1 Treść zadania

Zaimplementować problem producentów i konsumentów z uwzględnieniem poniższych zasad:

1. Bufor o rozmiarze $2M$
2. Jest m producentów i n konsumentów
3. Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
4. Konsument pobiera losowa liczbę elementów (nie więcej niż M)
5. Zaimplementować przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities
6. Przeprowadzić porównanie wydajności (np. czas wykonywania) vs. różne parametry, zrobić wykresy i je skomentować

2 Implementacja - monitory

W języku Java każdy tworzony obiekt ma przypisany do siebie monitor. Użycie słowa kluczowego *synchronized* w definicji metody sprawia, że monitor obiektu, na którym metoda jest wywoływana, jest zablokowany na czas wykonywania, przez co pozostałe wątki chcące operować na tym obiekcie muszą czekać na jego zwolnienie.

```
cclass Producer extends Thread {
private Buffer buf;
    private int max;

    public Producer(Buffer buf, int max) {
        this.buf = buf;
        this.max = max;
```

```
    }

public void run() {
      int iters = ThreadLocalRandom.current().nextInt(0, max + 1);

  for (int i = 0; i < iters; ++i) {
buf.put(i);
  }
}
}

class Consumer extends Thread {
private Buffer buf;
    private int max;

    public Consumer(Buffer buf, int max) {
        this.buf = buf;
        this.max = max;
    }

public void run() {
      int iters = ThreadLocalRandom.current().nextInt(0, max + 1);

  for (int i = 0; i < iters; ++i) {
System.out.println(buf.get());
  }
}
}

class Buffer {
    private List<Integer> buf = new ArrayList<Integer>();
    private int size;

    public Buffer(int size) {
        this.size = size;
    }

    public synchronized void put(int i) {
        while (buf.size() >= size) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }
```

```
        buf.add(i);
        notifyAll();
    }

    public synchronized int get() {
        while (buf.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }

        int retVal = buf.get(0);
        buf.remove(0);
        notifyAll();
        return retVal;
    }
}

public class ProdCon {

    public static void main(String[] args) throws InterruptedException {
        int M = 100;
        Buffer buff = new Buffer(2*M);

        int m = 3;
        int n = 3;

        ExecutorService service = Executors.newFixedThreadPool(n + m);

        for (int i=0; i<m; ++i) {
            service.submit(new Producer(buff, M));
        }

        for (int i=0; i<m; ++i) {
            service.submit(new Consumer(buff, M));
        }

        service.shutdown();
        while (!service.awaitTermination(24L, TimeUnit.HOURS))
            ;
    }
}
```

3 Implementacja - Java Concurrency Utilities

Monitory można zastąpić mechanizmami z pakietu Java Concurrency Utilities. Zmiany dotyczą tylko klasy *Buffer*. Reszta kodu pozostaje taka sama.

3.1 Lock

Poniżej implementacja z użyciem klasy *Lock*:

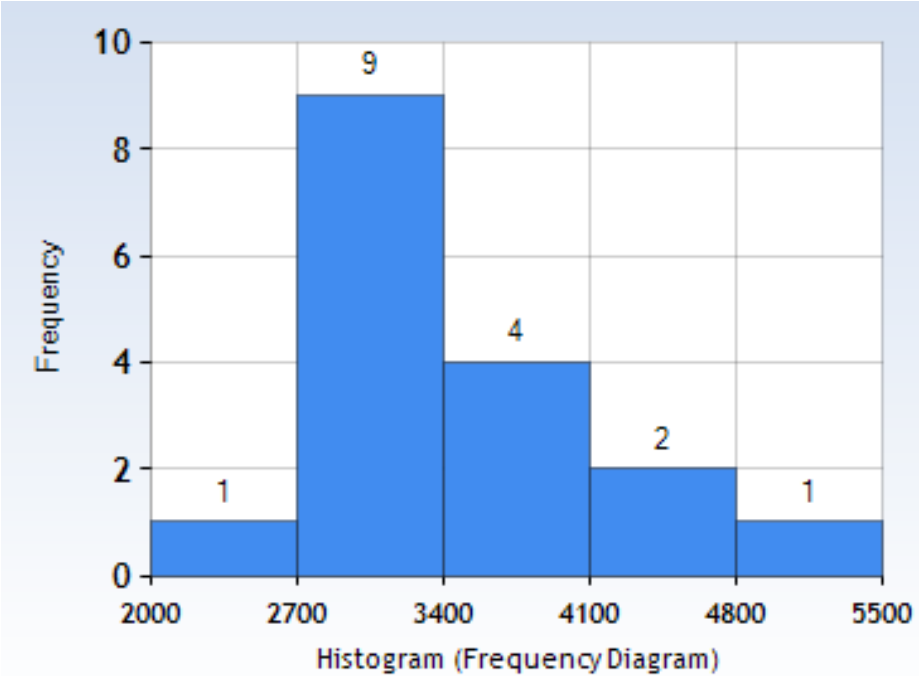
```
class Buffer {
    private List<Integer> buf = new ArrayList<Integer>();
    private int size;

    private ReentrantLock lock = new ReentrantLock();
    Condition stackEmptyCondition = lock.newCondition();
    Condition stackFullCondition = lock.newCondition();

    public Buffer(int size) {
        this.size = size;
    }

    public void put(int i) {
        try {
            lock.lock();
            while(buf.size() == size) {
                stackFullCondition.await();
            }
            buf.add(i);
            stackEmptyCondition.signalAll();
        } catch (InterruptedException e) {
            System.exit(0);
        } finally {
            lock.unlock();
        }
    }

    public int get() {
        try {
            lock.lock();
            while(buf.isEmpty()) {
                stackEmptyCondition.await();
            }
            int retVal = buf.get(0);
            buf.remove(0);
            return retVal;
        } catch (InterruptedException e) {
```

```
            System.exit(0);
            return 0;
        } finally {
            stackFullCondition.signalAll();
            lock.unlock();
        }
    }
}
```

4 Porównania

Poniżej znajduje się porównanie wydajności problemów zaimplementowanych za pomocą monitorów oraz *Java Concurrency Utilities*. Z racji na ograniczenia w treści polecenia wywołanie programu może skończyć się zablokowaniem, wywołania skutkujące zablokowaniem nie zostaną uwzględnione (przez co badane różnice pomiędzy n oraz m będą niewielkie, duże skutkują zablokowaniem programu).

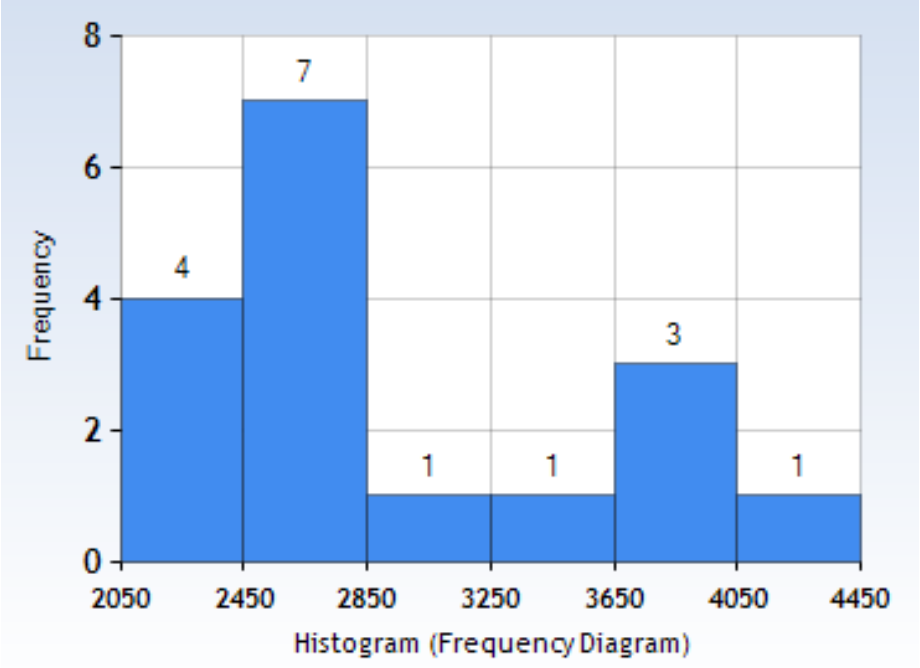
Do głownej funkcji programu dodana została funkcjonalność mierząca czas działania, a do metod klasy *Buffer* wywołania funkcji *sleep* spowalniające jej działanie. Poniżej wykres wywołań dla $M = 100$ oraz $m = n = 3$ przy użyciu monitorów:



Rysunek 1: Hisogram kolejnych wywołań (monitory), milisekundy

Średnia uzyskanych wartości to 3405 ms.

Poniżej eksperyment z takimi samymi parametrami, ale przy użyciu klasy *Lock* z *Java Concurrency Utilities*:



Rysunek 2: Hisogram kolejnych wywołań (*Java Concurrency Utilities*), milisekundy

Średnia uzyskanych wartości to 2918 ms, jest mniejsza niż w przypadku użycia monitorów. Zwiększanie wartości M skutkuje liniowym wzrostem czasu trwania wykonywania programu. Gdy użyte zostają wartości $n \neq m$, program zbyt często się blokuje.

5 Wnioski

Użycie narzędzi z pakietu *Java Concurrency Utilities* pozwala na osiągnięcie tego samego rezultatu,co przy używaniu monitorów i funkcji *wait* oraz *notifyAll* , jednak dużo łatwiej oraz często optymalniej. Pakiet udostępnia również narzędzia do przydatne w wielu konkretnych scenariuszach, np. *AtomicInteger*, odpowiednik typu *int* zapewniający atomowość operacj.

6 Bibliografia

1. Dokumentacja języka Java - docs.oracle.com