Teoria Współbieżności

Laboratorium 5

Andrzej Ratajczak

1. Producenci i konsumenci z losową ilością pobieranych i wstawianych opcji

Założenia:

- Bufor o rozmiarze 2M
- Jest *m* producentów i *n* konsumentów
- Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
- Konsument pobiera losowa liczbę elementów (nie więcej niż M)

Zadanie wykonałem w dwóch wariantach, korzystając z mechanizmów systemowych oraz z mechanizmów z pakietu Java Concurrency Utilities.

```
class Producer implements Runnable {
    private Buffer _buf;
    private int upperBound;
    private int value;
    public Producer(Buffer buf, int upperBound, int value) {
        _buf = buf;
        this.upperBound = upperBound;
        this.value = value;
    }

    public void run() {
        for (int i = 0; i < value; ++i) {
            _buf.put(i);
        }
    }
}

class Consumer implements Runnable {
    private Buffer _buf;
    private int upperBound;
    private int value;
    public Consumer(Buffer buf, int upperBound, int value) {
        _buf = buf;
        this.upperBound = upperBound;
        this.value = value;
    }

    public void run() {
        for (int i = 0; i < value; ++i) {
            _buf.get();
        }
    }
}

class Buffer {
```

```
    private int _size;
    private int iterator = -1;
    private int[] buffer;
    private Lock lock = new ReentrantLock();

    public Buffer(int size) {
        _size = size;
        buffer = new int[size];
    }

    private boolean canPut() {
        return iterator + 1 < _size;
    }

    public void put(int i) {
        lock.lock();
        while (!canPut()) {
            lock.unlock();
            lock.lock();
        }
        iterator++;
        buffer[iterator] = i;
        lock.unlock();
    }

    private boolean canGet() {
        return iterator >= 0;
    }

    public int get() {
        lock.lock();
        while (!canGet()) {
            lock.unlock();
            lock.lock();
        }
        int k = buffer[iterator];
        iterator--;
        lock.unlock();
        return k;
    }
}

public class Main {

    public static void main(String[] args) throws InterruptedException {

        int M = 100;
        int m = 5;
        int n = 5;

        Random rn = new Random();

        int checker = 0;
        int newValue;
        for(int res = 0; res < 100;) {
            checker = 0;
            Buffer buffer = new Buffer(2 * M);
            long begin = System.nanoTime();
            ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(m + n);
            for (int i = 0; i < m; i++) {
                newValue = rn.nextInt(M) + 1;
                checker += newValue;
```

```
                executor.execute(new Producer(buffer, M, newValue));
            }
            for (int i = 0; i < n; i++) {
                newValue = rn.nextInt(M) + 1;
                checker -= newValue;
                executor.execute(new Consumer(buffer, M, newValue));
            }
            executor.shutdown();
            if (checker < 0 || checker > 2 * M) {
                executor.awaitTermination(0, TimeUnit.SECONDS);
                executor.shutdownNow();
                continue;
            }

            long end = System.nanoTime();
            System.out.println(end - begin);
            executor.shutdownNow();
            res++;
        }
    }
}
```

Kod źródłowy 1. Implementacja rozwiązania przy użyciu funkcji systemowych.

```
class Buffer {

    private int _size;
    private int iterator = -1;
    private int[] buffer;
    private Lock lock = new ReentrantLock();

    public Buffer(int size) {
        _size = size;
        buffer = new int[size];
    }

    private boolean canPut() {
        return iterator + 1 < _size;
    }

    public void put(int i) {
        lock.lock();
        while (!canPut()) {
            lock.unlock();
            lock.lock();
        }
        iterator++;
        buffer[iterator] = i;
        lock.unlock();
    }

    private boolean canGet() {
        return iterator >= 0;
    }

    public int get() {
        lock.lock();
        while (!canGet()) {
            lock.unlock();
            lock.lock();
        }
        int k = buffer[iterator];
        iterator--;
```

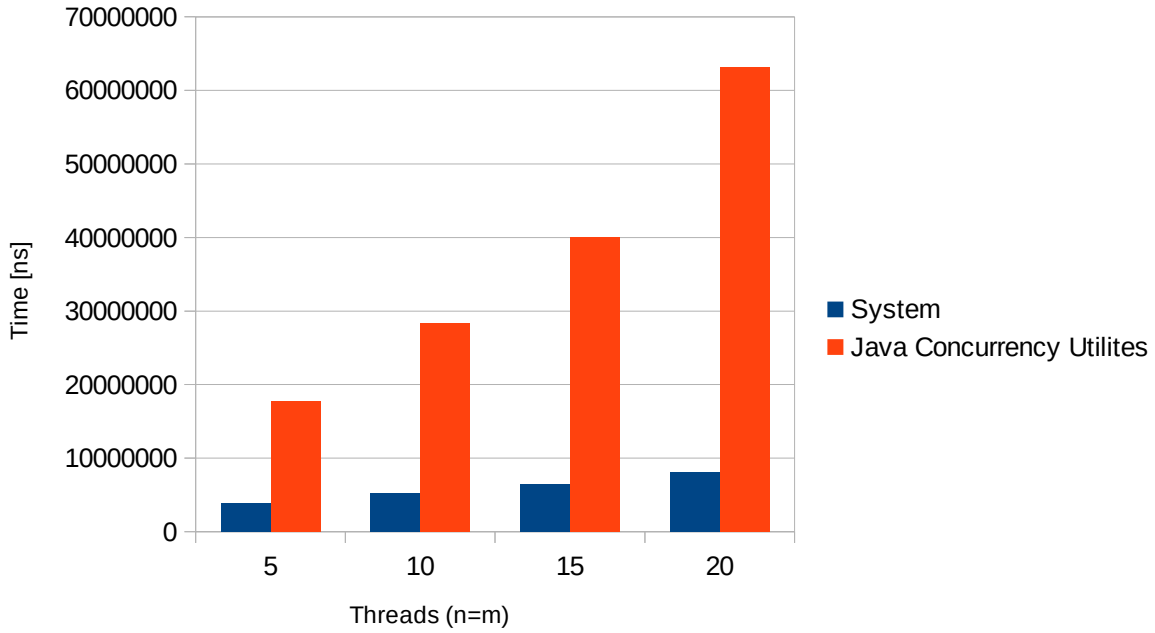
```
        lock.unlock();
        return k;
    }
}

public class Main {

    public static void main(String[] args) throws InterruptedException {

        int M = 100;
        int m = 5;
        int n = 5;

        Random rn = new Random();

        int checker = 0;
        int newValue;
        for(int res = 0; res < 100;) {
            checker = 0;
            Buffer buffer = new Buffer(2 * M);
            long begin = System.nanoTime();
            ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(m + n);
            for (int i = 0; i < m; i++) {
                newValue = rn.nextInt(M) + 1;
                checker += newValue;
                executor.execute(new Producer(buffer, M, newValue));
            }
            for (int i = 0; i < n; i++) {
                newValue = rn.nextInt(M) + 1;
                checker -= newValue;
                executor.execute(new Consumer(buffer, M, newValue));
            }
            executor.shutdown();
            if (checker < 0 || checker > 2 * M) {
                executor.awaitTermination(0, TimeUnit.SECONDS);
                executor.shutdownNow();
                continue;
            }

            long end = System.nanoTime();
            System.out.println(end - begin);
            executor.shutdownNow();
            res++;
        }
    }
}
```

Kod źródłowy 2. Zmiany w klasie Main oraz Buffer wykorzystujące mechanizmy Java Concurrency Utilities.

Zdecydowałem się zamienić zwykłe wątki na wątki zarządzane przez ExecutorService. Dodatkowo bloki synchronized zamieniłem na zwykłe funkcje z sekcją krytyczną zabezpieczoną ReentrantLockiem. Dodatkowo zautomatyzowałem testowanie każdego przypadku, tak by 100 razy liczyło czas wykonania, tak aby można było policzyć średnią. Zabezpieczyłem się przed zakleszczeniem. Jeżeli wylosowane wartości by nie spełniały warunku koniecznego zakończenia się poprawnego programu zabijałem wątki i powtarzałem iterację.

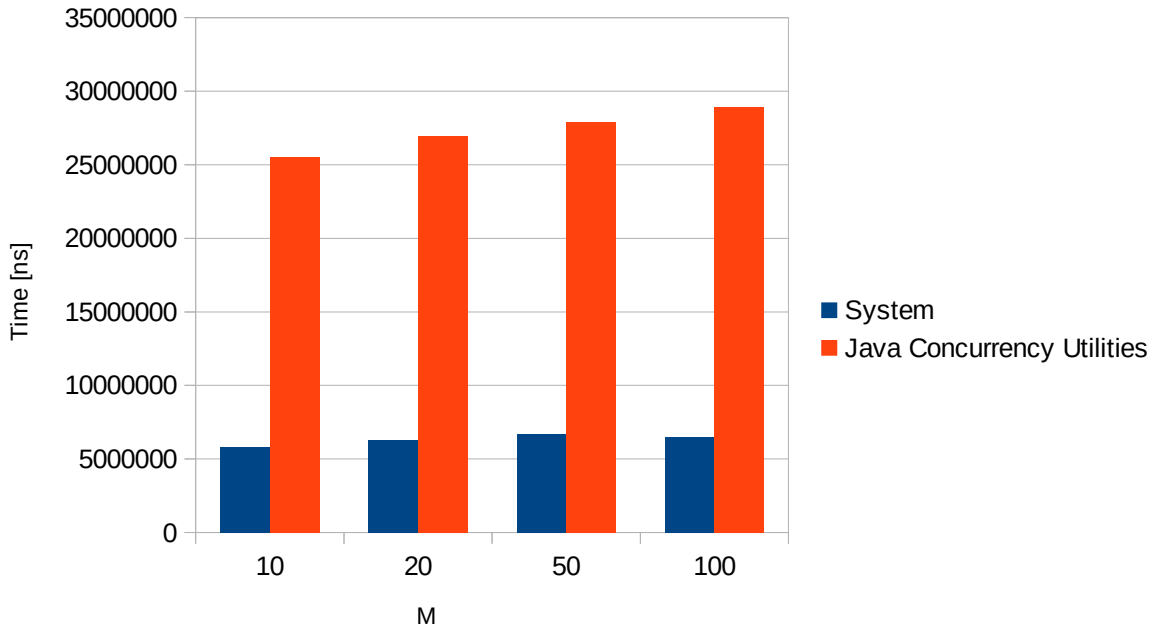
Poniższe wykresy prezentują już zebrane wyniki dla różnych n, m, M.

M = 100



Jak widać Java Concurrency wypadają o wiele gorzej w tym zestawieniu. Powodem może być ociężałość ExecutorSerwisu, który gwarantuje większą kontrolę nad wątkami kosztem wydajności.

n=m=10



Jak widać pojemność bufora nie ma za dużego znaczenia jeżeli chodzi efektywność działania programu. Parametr M determinuje również potencjalną ilość produkowanych/konsumowanych wartości dlatego jego zmiana nie wpływa za bardzo na działanie programu.