Teoria współbieżności

Andrzej Ratajczak

Laboratorium 5

1. Problem czytelników i pisarzy

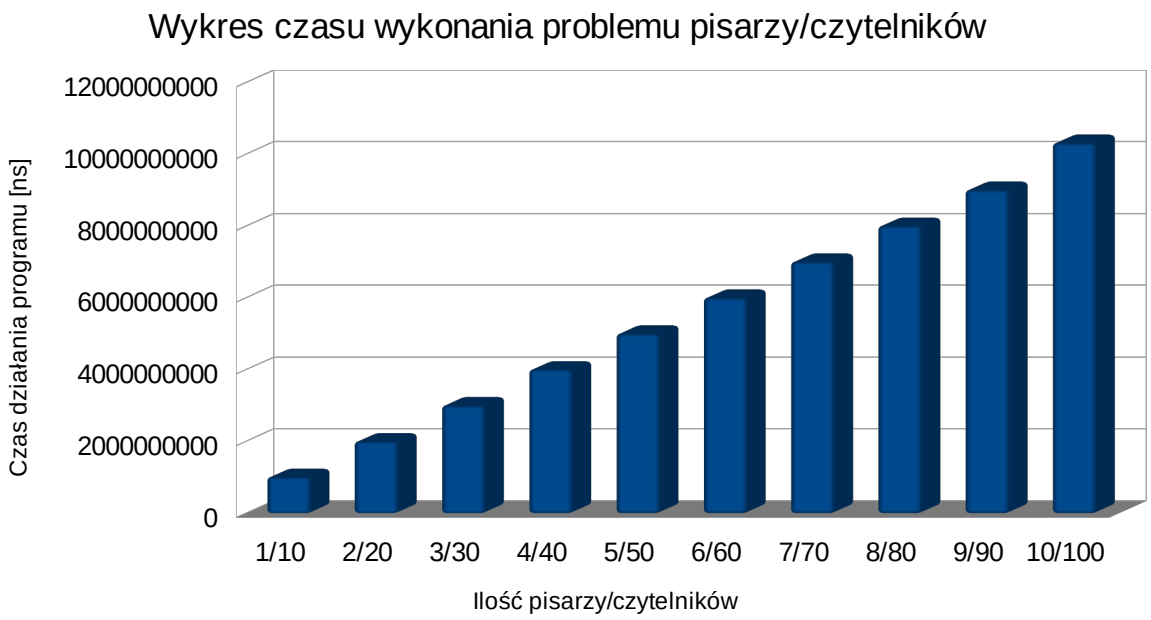
Do realizacji zadania wykorzystałem 3 semafory. Jeden semafor broni dostępu do samego zasobu. Drugi jest odpowiedzialny za kolejke procesów wspępujacych do sekcji krytycznej. Jeżeli jakiś pisarz zgłosi chęć pisania nie wchodzą kolejni czytelnicy. Trzeci pilnuje by operacje na liczbie aktualnych czytelników były atomiczne.

```
package lab5.src.ReadersWriters;

import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicInteger;

class Reader extends Thread {

    private Semaphore resourceAccess;
    private Semaphore readCountAccess;
    private Semaphore serviceQueue;
    private AtomicInteger readerCount;

    public Reader(Semaphore resourceAccess, Semaphore readCountAccess, Semaphore
serviceQueue, AtomicInteger readerCount) {
        this.resourceAccess = resourceAccess;
        this.readCountAccess = readCountAccess;
        this.serviceQueue = serviceQueue;
        this.readerCount = readerCount;
    }

    @Override
    public void run() {
        try {
            serviceQueue.acquire();
            readCountAccess.acquire();

            if (readerCount.get() == 0) {
                resourceAccess.acquire();
            }
            readerCount.addAndGet(1);
            serviceQueue.release();
            readCountAccess.release();
            //System.out.println("Thread "+Thread.currentThread().getName() + "
is READING");
            Thread.sleep(500);
            //System.out.println("Thread "+Thread.currentThread().getName() + "
leaves");

            readCountAccess.acquire();
            readerCount.addAndGet(-1);
            if(readerCount.get() == 0) {
                resourceAccess.release();
            }
            readCountAccess.release();
```

```
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

class Writer extends Thread {

    private Semaphore resourceAccess;
    private Semaphore readCountAccess;
    private Semaphore serviceQueue;

    public Writer(Semaphore resourceAccess, Semaphore readCountAccess, Semaphore
serviceQueue) {
        this.resourceAccess = resourceAccess;
        this.readCountAccess = readCountAccess;
        this.serviceQueue = serviceQueue;
    }

    @Override
    public void run() {
        try {
            serviceQueue.acquire();
            resourceAccess.acquire();
            serviceQueue.release();
            //System.out.println("Thread "+Thread.currentThread().getName() + "
is WRITING");
            Thread.sleep(500);
            //System.out.println("Thread "+Thread.currentThread().getName() + "
leaves");
            resourceAccess.release();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

public class Main {

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

        Semaphore resourceAccess = new Semaphore(1);
        Semaphore readCountAccess = new Semaphore(1);
        Semaphore serviceQueue = new Semaphore(1);
        AtomicInteger readCount = new AtomicInteger(0);


        for(int k = 1; k < 10; k++) {
            int readersCount = k*10;
            int writersCount = k;
            Reader[] readers = new Reader[readersCount];
            Writer[] writers = new Writer[writersCount];

            long start = System.nanoTime();

            for (int j = 0; j < k; j++) {
                for (int i = 0; i < readersCount / k; i++) {
                    readers[j * 10 + i] = new Reader(resourceAccess,
readCountAccess, serviceQueue, readCount);
                    readers[j * 10 + i].start();
                }

                for (int i = 0; i < writersCount / k; i++) {
```

```
                    writers[j] = new Writer(resourceAccess, readCountAccess,
serviceQueue);
                    writers[j].start();
                }
            }

            for (int i = 0; i < readersCount; i++) {
                readers[i].join();
            }

            for (int i = 0; i < writersCount; i++) {
                writers[i].join();
            }

            long end = System.nanoTime();

            System.out.println(end - start);
        }
    }
}
```

Kod źródłowy 1. Kod problemu pisarzy i czytelników



W moim schemacie widać że czytelnicy mają czytać przez 500 milisekund, natomiast pisarz piszę również 500 milisekund. Wchodzą partiami – 10 czytelników – 1 pisarz. Widać że czytelnicy czytają równolegle natomiast pisarz pisze pojedynczo. (500 milisekund + 500 milisekund = 1 sekunda = 1000000000 ns)

2. Problem blokowania drobnoziarnistego.

Poniżej znajduje się implementacja listy odsyłaczowej realizująca operacje dodawnia, usuwania i przeszukiwania listy metodą blokowania drobnoziarnistego.

```
class FineGrainedNode {
    private Object o;
    private FineGrainedNode nextNode;
    Lock lock = new ReentrantLock();
```

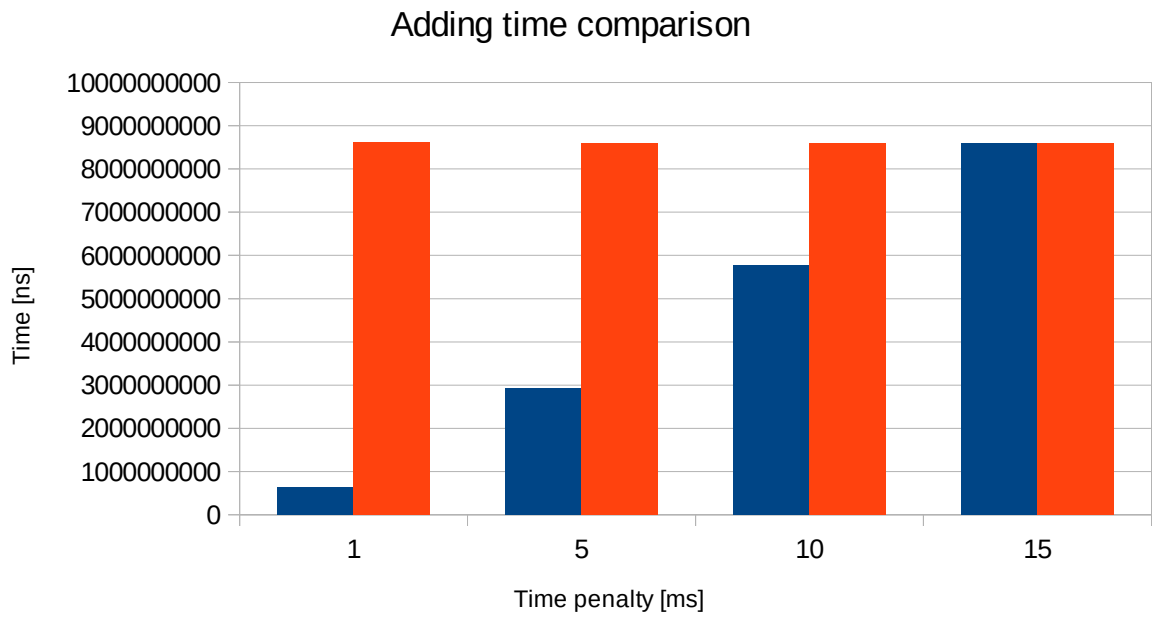
```
    FineGrainedNode(Object o) {
        this.o = o;
        this.nextNode = null;
    }

    boolean contains(Object o) {
        if(nextNode != null) {
            nextNode.lock.lock();
            if (this.nextNode.o == o) {
                this.lock.unlock();
                nextNode.lock.unlock();
                return true;
            } else {
                this.lock.unlock();
                return nextNode.contains(o);
            }
        } else {
            this.lock.unlock();
            return false;
        }
    }

    boolean remove(Object o) {
        if(this.nextNode != null) {
            this.nextNode.lock.lock();
            if (nextNode.o == o) {
                this.nextNode = nextNode.nextNode;
                this.lock.unlock();
                return true;
            } else {
                this.lock.unlock();
                return nextNode.remove(o);
            }
        } else {
            this.lock.unlock();
            return false;
        }
    }

    boolean add(Object o) {
        if(this.nextNode == null) {
            this.nextNode = new FineGrainedNode(o);
            this.lock.unlock();
            return true;
        } else {
            this.nextNode.lock.lock();
            this.lock.unlock();
            return nextNode.add(o);
        }
    }
}


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

        //Setup
        FineGrainedNode head = new FineGrainedNode(new Object());
        Object[] objects = new Object[1000];

        Thread[] runner = new Thread[100];

        //Adding to list
```

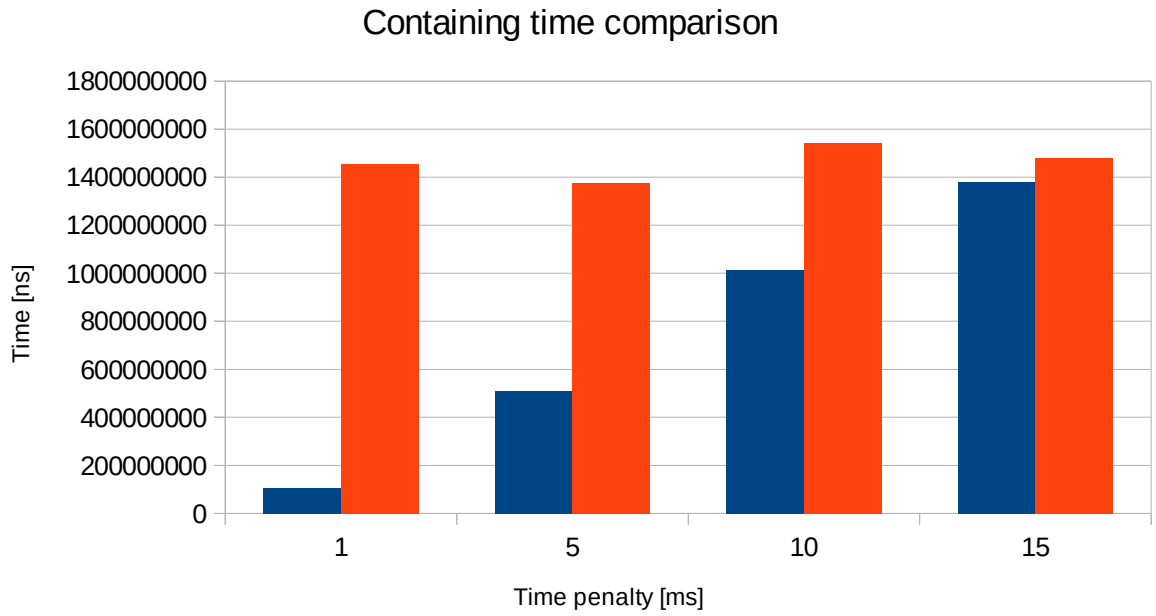
```
        long addingStart = System.nanoTime();

        for(int i = 0; i < 10; i++) {
            for(int j = 0; j < 100; j++) {
                objects[100 * i + j] = new Object();
            }
            final int bound = i;
            runner[i] = new Thread( () -> {
                for(int k = bound*100; k < ((bound+1) * 100); k++) {
                    head.lock.lock();
                    head.add(objects[k]);
                }
            });
            runner[i].start();
        }

        for(int i = 0; i < 10; i++) {
            runner[i].join();
        }

        long addingEnd = System.nanoTime();

        //Containing

        Random rand = new Random();

        long containingStart = System.nanoTime();

        for(int i = 0; i < 100; i++) {
            runner[i] = new Thread( () -> {
                head.lock.lock();
                head.contains(objects[rand.nextInt(999)]);
            });
            runner[i].start();
        }

        for(int i = 0; i < 100; i++) {
            runner[i].join();
        }

        long containingEnd = System.nanoTime();
        //Removing
        long removingStart = System.nanoTime();

        for(int i = 0; i < 100; i++) {
            runner[i] = new Thread( () -> {
                head.lock.lock();
                head.remove(objects[rand.nextInt(999)]);
            });
            runner[i].start();
        }

        for(int i = 0; i < 100; i++) {
            runner[i].join();
        }

        long removingEnd = System.nanoTime();

        // Output

        System.out.println("Adding time: " + (addingEnd - addingStart) + "ns");
        System.out.println("Containing time: " + (containingEnd -
containingStart) + "ns");
```

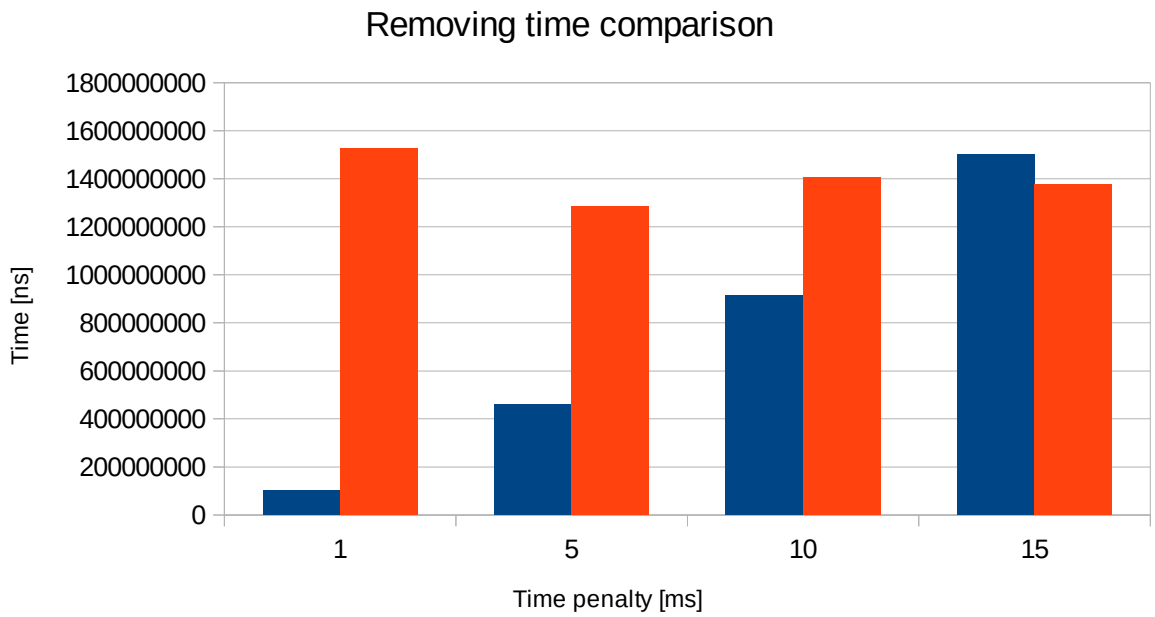
```
        System.out.println("Removing time: " + (removingEnd - removingStart) +
"ns");
    }
}
```

Bardzo podobną listę, ale bez zamków na każdym nodzie, natomiast jednym globalnym, okalającym dostęp do tablicy podałem testom czasowym. Poniższy wykrest prezentuje zebrane wyniki.

Dodanie 100 elementów, znalezienie 10, usunięcie 10 – 10 wątków







Czas dla blokowania drobnoziarnistego rośnie liniowo, jest to zrouzmiałe, jako że operacje praktycznie wykonują się niezależnienie od siebie. Czas blokowania całej listy jest w miarę stały. Jest to wina zbyt małego kosztu wykonywania operacji. Jednakże gdyby zwiększyć koszt operacji, wykres byłby nieczytelny jako że byłaby zbyt duża dysproporcja czasu blokowania drobnoziarnistego i blokowania całej listy.