SPRAWOZDANIE V

TEORIA WSPÓŁBIEŻNOŚCI

Problem czytelników i pisarzy oraz blokowanie drobnoziarniste

DAWID BIAŁKA

DATA LABORATORIUM 3.11.2020

DATA ODDANIA 10.11.2020

1.
Problem **czytelników i pisarzy** proszę rozwiązać przy pomocy: semaforów i zmiennych warunkowych

Proszę wykonać pomiary dla różnej ilości czytelników (10-100) i pisarzy (od 1 do 10). W sprawozdaniu proszę narysować 3D wykres czasu w zależności od liczby wątków i go zinterpretować.

2.

Proszę zaimplementować listę, w której każdy węzeł składa się z wartości typu Object, referencji do następnego węzła oraz zamka (lock).

Proszę zastosować metodę drobnoziarnistego blokowania do następujących metod listy:

```
boolean contains(Object o); //czy lista zawiera element o
boolean remove(Object o); //usuwa pierwsze wystąpienie elementu o
boolean add(Object o); //dodaje element o na koncu listy
```

Proszę porównać **wydajność** tego rozwiązania w stosunku do listy z jednym zamkiem blokującym dostęp do całości. Należy założyć, że koszt czasowy operacji na elemencie listy (porównanie, wstawianie obiektu) może być duży - proszę wykonać pomiary dla różnych wartości tego kosztu.

Zad 1.

Koncepcja

Tworzymy algorytm czytelników (10 – 100) i pisarzy (1-10) dla wariantu, gdzie czytelnicy mają pierwszeństwo. Liczymy, przez jaki czas wątki czekały (albo na dostęp do sekcji krytycznej albo do wejścia do czytelni (tyczy się to pisarzy)). Na potrzeby liczenia czasu i przekazywania go do klasy Main w celu zapisu do pliku implementujemy klasę TimeContainer. Czas liczymy przy pomocy metody System.NanoTime(), wynik ten dzielimy przez 1000 000 i uzyskujemy wynik w milisekundach.

Implementacja i wyniki

```
public static void main(String[] args) {
      int writersNumber = Integer.parseInt(args[0]);
      int readersNumber = Integer.parseInt(args[1]);


      ArrayList<Writer> writers = new ArrayList<>();
      ArrayList<Reader> readers = new ArrayList<>();

      TimeContainer timeContainer = new TimeContainer();
      final Library library = new Library(timeContainer);

      for (int i = 0; i < writersNumber; ++i) {
         writers.add(new Writer(i, library));
         writers.get(i).start();

      }
      for (int i = 0; i < readersNumber; ++i) {
         readers.add(new Reader(i, library));
         readers.get(i).start();
      }

      for (int i = 0; i < writersNumber; ++i) {
         try {
            writers.get(i).join();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      for (int i = 0; i < readersNumber; ++i) {
         try {
            readers.get(i).join();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      long meanReaderWaitingTime = timeContainer.getReaderWaitingTime() / readersNumber;
      long meanWriterWaitingTime = timeContainer.getWriterWaitingTime() / writersNumber;

      try {
         FileWriter myWriter = new FileWriter("results.txt", true);
         myWriter.write(String.format("%d %d %d %d", writersNumber, readersNumber,
meanWriterWaitingTime, meanReaderWaitingTime) + "\n");
         myWriter.close();
      } catch (IOException e) {
         System.out.println("An error occurred.");
         e.printStackTrace();
      }


   }
}
```

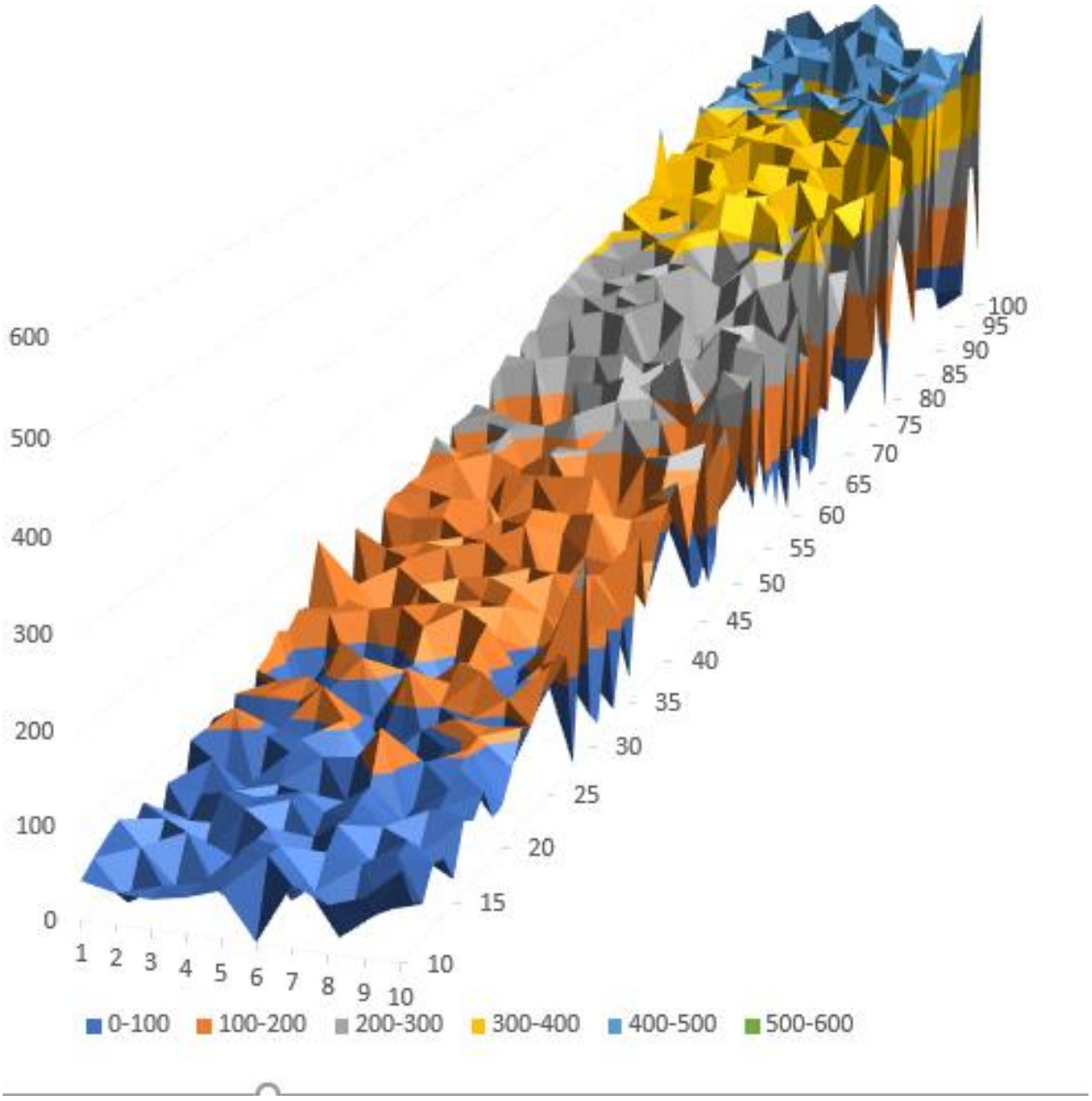
```
public class TimeContainer {
    private long readerWaitingTime = 0;
    private long writerWaitingTime = 0;

    public TimeContainer() {}

    public void updateReaderWaitingTime(long start, long end) {
        this.readerWaitingTime += (end - start);
    }

    public void updateWriterWaitingTime(long start, long end) {
        this.writerWaitingTime += (end - start);
    }

    public long getReaderWaitingTime() {
        return this.readerWaitingTime / 1000000;
    }

    public long getWriterWaitingTime() {
        return this.writerWaitingTime / 1000000;
    }
}
```

```
public class Reader extends Thread {
    private int ID;
    private Library library;

    public Reader(int ID, Library library) {
        this.ID = ID;
        this.library = library;
    }

    public void run() {
        for (int i = 0; i < 500; i++) {
            library.beginReading();
            library.endReading();
        }
    }
}
```

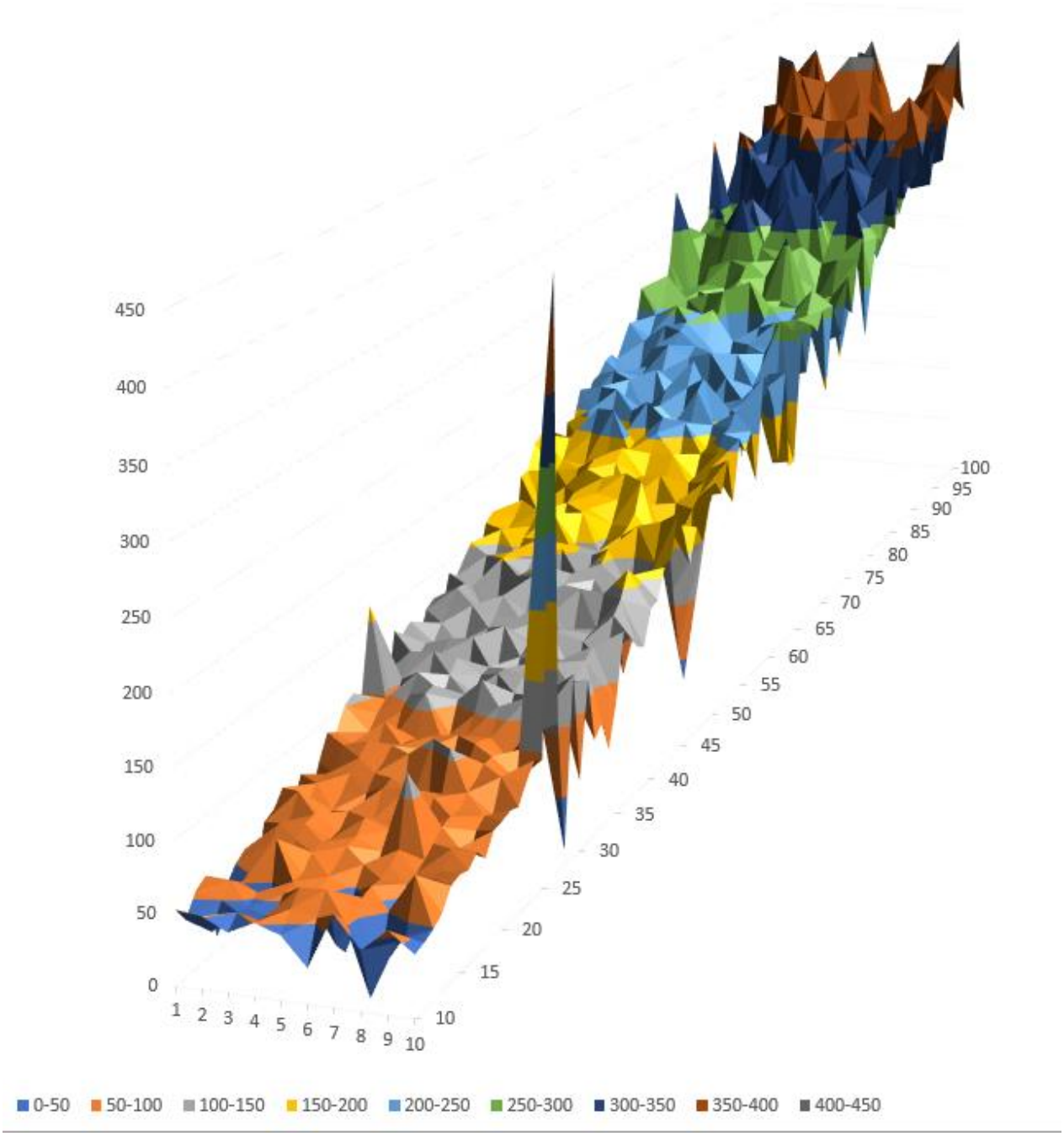
```
public class Writer extends Thread {
    private int ID;
    private Library library;

    public Writer(int ID, Library library) {
        this.ID = ID;
        this.library = library;
    }

    public void run() {
        for (int i = 0; i < 500; i++) {
            library.beginWriting();
            library.endWriting();
        }
    }
}
```

```
public class Library {
    Semaphore reading = new Semaphore(1);
    Semaphore writing = new Semaphore(1);

    TimeContainer timeContainer;

    private ReentrantLock libraryLock = new ReentrantLock();
    private Condition readers = libraryLock.newCondition();
    private Condition writers = libraryLock.newCondition();

    private int isReading = 0;
    private int isWriting = 0;
    private int writerWaiting = 0;
    private int readerWaiting = 0;

    int readersNumber = 0;

    public Library(TimeContainer timeContainer) {
        this.timeContainer = timeContainer;
    }

    public void beginReading() {
        try {
            long start = System.nanoTime();
            reading.acquire(1);

            readersNumber++;
            if(readersNumber == 1)
                writing.acquire();

            timeContainer.updateReaderWaitingTime(start, System.nanoTime());
            reading.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

    public void endReading() {
        try {
            long start = System.nanoTime();
            reading.acquire(1);
            timeContainer.updateReaderWaitingTime(start, System.nanoTime());
            readersNumber--;
            if(readersNumber == 0)
                writing.release();
            reading.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
```

```
public void beginWriting() {
      try {
         long start = System.nanoTime();
         writing.acquire(1);
         timeContainer.updateWriterWaitingTime(start, System.nanoTime());
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }

   public void endWriting() {
      writing.release();

   }
}
```



Wykres średniego czasu oczekiwania przez pisarza w zależności od liczby pisarzy i czytelników. Czas podany w milisekundach.



Wykres średniego czasu oczekiwania przez czytelnika w zależności od liczby pisarzy i czytelników. Czas podany w milisekundach.

Wnioski

Z obu wykresów widzimy, że wraz ze wzrostem liczby czytelników liniowo wzrasta średni czas oczekiwania zarówno dla pisarza i czytelnika.

Wzrost liczby pisarzy w przypadku średniego czasu oczekiwania dla pisarza powoduje wzrost, choć niewielki czasu oczekiwania.

Dla średniego czasu oczekiwania dla czytelnika wzrost liczby pisarzy nie powoduje zwiększenia czasu oczekiwania. Widzimy jednak, że maksymalny czas oczekiwania dla czytelnika był niższy, ok. 450 ms, dla pisarza było to ok. 600 ms. Jest tak, ponieważ czytelnicy mają wyższy priorytet.

Zdarzają się sytuacje, gdzie mamy pojedyncze bardzo niskie wartości czasu oczekiwania lub bardzo duże, wiele czynników może to powodować , np. kolejność uzyskiwania procesora przez wątki.

Zad. 2

Koncepcja

Tworzymy dwie klasy, FirstList – lista z blokowaniem drobnoziarnistym, SecondList – lista z blokowaniem ogólnym. Następnie na różnych czasów trwania poszczególnych operacji wykonujemy kilka operacji na danej liście i mierzymy całkowity czas wykonania wszystkich operacji.

Implementacja i wyniki

```
public class Main extends Thread {
   private Object[] o;
   private FirstList list;
   private SecondList list2;
   private static long sleepTime;

   public Main(Object[] o, FirstList list) {
      super();
      this.o = o;
      this.list = list;
   }

   public Main(Object[] o, SecondList list2) {
      super();
      this.o = o;
      this.list2 = list2;
   }

   @Override
   public void run() {
      /*for (int i = 0, n = o.length; i < 10; ++i) {
         try {
            list.add(o[i % n]);
            list.contains(o[(i + 1) % n]);
            list.remove(o[(i + 2) % n]);
            list.contains(o[(i + 3) % n]);
            list.add(o[(i + 4) % n]);
            list.remove(o[(i + 5) % n]);
            list.add(o[(i + 6) % n]);
            list.remove(o[(i + 7) % n]);
            list.contains(o[(i + 8) % n]);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }*/

      for (int i = 0, n = o.length; i < 10; ++i) {
         try {
            list2.add(o[i % n]);
            list2.contains(o[(i + 1) % n]);
            list2.remove(o[(i + 2) % n]);
            list2.contains(o[(i + 3) % n]);
            list2.add(o[(i + 4) % n]);
            list2.remove(o[(i + 5) % n]);
            list2.add(o[(i + 6) % n]);
            list2.remove(o[(i + 7) % n]);
            list2.contains(o[(i + 8) % n]);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public static void main(String[] args) {
      Object[] o = { "ala ma kota", 10L, 10.0, "ola ma kota", 11L, 11.0,
            "ala ma kata", 11L, 11.0 };
      FirstList list = new FirstList("ala ma kota", null);
```

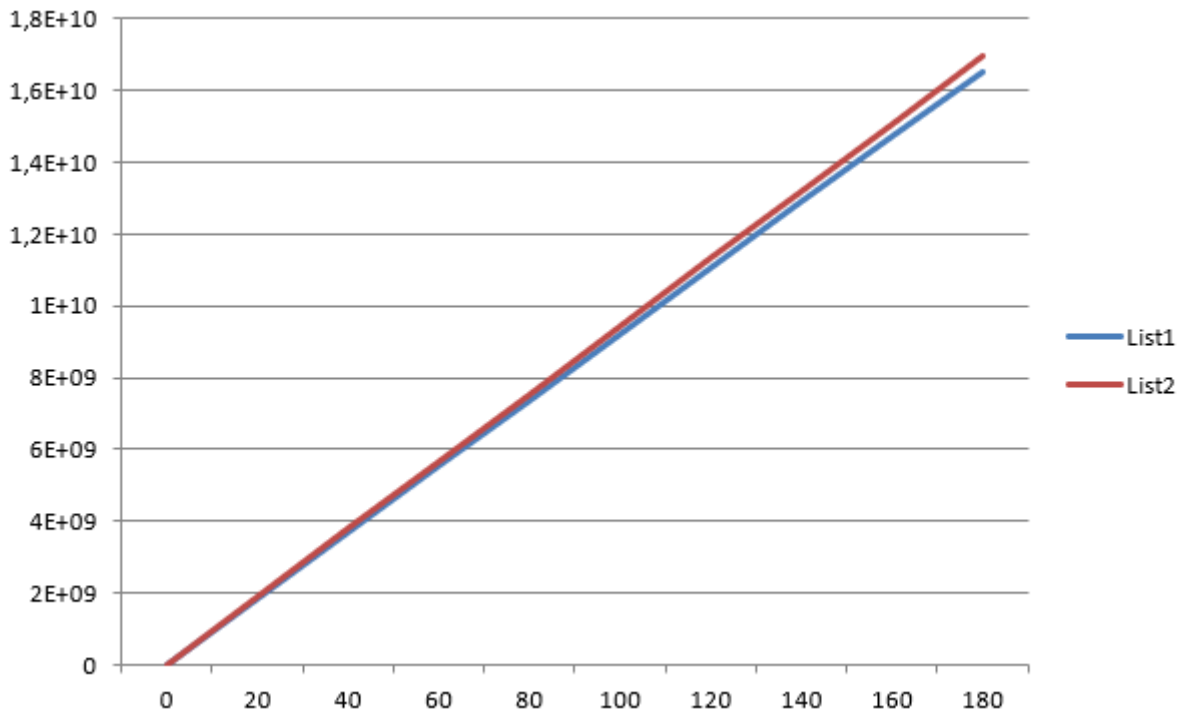
```
        SecondList list2 = new SecondList("ala ma kota", null);

        /*for (sleepTime = 0; sleepTime < 200; sleepTime += 20) {
            list.setSleepTime(sleepTime);
            long time = System.nanoTime();
            Thread[] t = {new Main(o, list), new Main(o, list), new Main(o, list)};
            for(int i = 0; i < t.length; ++i) {
                t[i].start();
            }
            for(int i = 0; i < t.length; ++i) {
                try {
                    t[i].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            time = System.nanoTime() - time;
            System.out.println(sleepTime + " " + time);
        }*/

        for (sleepTime = 0; sleepTime < 200; sleepTime += 20) {
            list2.setSleepTime(sleepTime);
            long time = System.nanoTime();
            Thread[] t = {new Main(o, list2), new Main(o, list2), new Main(o, list2)};
            for(int i = 0; i < t.length; ++i) {
                t[i].start();
            }
            for(int i = 0; i < t.length; ++i) {
                try {
                    t[i].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            time = System.nanoTime() - time;
            System.out.println(sleepTime + " " + time);
        }
    }
}
```

```
import java.util.concurrent.locks.ReentrantLock;

public class FirstList {
   private Object val;
   private FirstList next;
   private ReentrantLock lock;
   private static long sleepTime;

   public FirstList(Object val, FirstList next) {
      this.val = val;
      this.next = next;
      lock = new ReentrantLock();
   }

   public boolean contains(Object o) throws InterruptedException {
      FirstList previous = null, tmp = this;
      lock.lock();
      try {
         while (tmp != null) {
            if (val == o) {
               Thread.sleep(sleepTime / 10);
               return true;
            }
            previous = tmp;
            tmp = tmp.next;
            try {
               if (tmp != null) {
                  tmp.lock.lock();
               }
            } finally {
               previous.lock.unlock();
            }
         }
      } finally {
         if (tmp != null) {
            tmp.lock.unlock();
         }
      }
      return false;
   }


   public boolean remove(Object o) throws InterruptedException {
      FirstList prevprev = null, previous = null, tmp = this;
      lock.lock();
      try {
         while (tmp != null) {
            if (val == o) {
               if (previous != null) {
                  previous.next = tmp.next;
                  tmp.next = null;
               }
               Thread.sleep(sleepTime / 3);
               return true;
            }
            prevprev = previous;
            previous = tmp;
            tmp = tmp.next;
            try {
```

```
                if (tmp != null) {
                    tmp.lock.lock();
                }
            } finally {
                if (prevprev != null) {
                    prevprev.lock.unlock();
                }
            }
        }
    } finally {
        if (previous != prevprev) {
            previous.lock.unlock();
        }
        if (tmp != null) {
            tmp.lock.unlock();
        }
    }
    return false;
}

public boolean add(Object o) throws InterruptedException {
    if (o == null) {
        return false;
    }
    FirstList tmp = this, next = this.next;
    lock.lock();
    try {
        while (next != null) {
            try {
                next.lock.lock();
            } finally {
                tmp.lock.unlock();
            }
            tmp = next;
            next = next.next;
        }
        tmp.next = new FirstList(o, null);
        Thread.sleep(sleepTime);
        return true;
    } finally {
        tmp.lock.unlock();
        if (next != tmp && next != null) {
            next.lock.unlock();
        }
    }
}

public static void setSleepTime(long sleepTime) {
    FirstList.sleepTime = sleepTime;
}
}
```

```java
import java.util.concurrent.locks.ReentrantLock;

public class SecondList {
    private Object val;
    private SecondList next;
    private static ReentrantLock lock = new ReentrantLock();
    private static long sleepTime;

    public SecondList(Object val, SecondList next) {
        this.val = val;
        this.next = next;
    }

    public boolean contains(Object o) throws InterruptedException {
        SecondList tmp = this;
        lock.lock();
        try {
            while (tmp != null) {
                if (val == o) {
                    Thread.sleep(sleepTime / 10);
                    return true;
                }
                tmp = tmp.next;
            }
        } finally {
            lock.unlock();
        }
        return false;
    }

    public boolean remove(Object o) throws InterruptedException {
        SecondList previous = null, tmp = this;
        lock.lock();
        try {
            while (tmp != null) {
                if (val == o) {
                    if (previous != null) {
                        previous.next = tmp.next;
                        tmp.next = null;
                    }
                    Thread.sleep(sleepTime / 3);
                    return true;
                }
                previous = tmp;
                tmp = tmp.next;
            }
        } finally {
            lock.unlock();
        }
        return false;
    }

    public boolean add(Object o) throws InterruptedException {
        if (o == null) {
            return false;
        }
        SecondList tmp = this, next = this.next;
        lock.lock();
        try {
```

```
        while (next != null) {
          tmp = next;
          next = next.next;
        }
        tmp.next = new SecondList(o, null);
        Thread.sleep(sleepTime);
        return true;
      } finally {
        lock.unlock();
      }
    }

    public static void setSleepTime(long sleepTime) {
      SecondList.sleepTime = sleepTime;
    }
}
```



Zależność czasu wykonania wszystkich operacji od kosztu wykonania pojedynczej operacji. List1 – blokowanie drobnoziarniste, List2 – blokowanie ogólne. (Oś Y – czas w nanosekundach, Oś X – czas w milisekundach).

Wnioski

Z powyższego wykresu widzimy, że wraz ze wzrostem kosztu wykonania pojedynczej operacji czas wykonania wszystkich operacji zaczyna być większy dla listy z blokowaniem ogólnym niż dla listy z blokowaniem drobnoziarnistym. Oznacza to, że w przypadku, gdy czas wykonania pojedynczej operacji jest dość duży lepiej wykorzystywać listę z blokowaniem drobnoziarnistym. Dla niskich wartości różnica jest bardzo niewielka.

Bibliografia

Z. Weiss, T. Gruźlewski, Programowanie współbieżne i rozproszone. WNT, Warszawa 1993.

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

http://home.agh.edu.pl/~funika/tw/lab5/

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html