Zad. 1

a) wzajemne wykluczanie:
Każdy wątek ustawia flag[i] = true i przechodzi przez pętlę, sprawdzając bity wszystkich wątków o niższych indeksach (for (int j = 0; j < i; j++)). Jeśli znajdzie wątek, który już ustawił flag[j] = true, zmienia swój bit z powrotem na ‘false’ i czeka, aż ten wątek zakończy swoją operację. Dzięki temu nie ma możliwości, aby dwa wątki jednocześnie ustawiły swoje bity i weszły do sekcji krytycznej.

b) niezakleszczenie:
W pierwszej pętli sprawdzane są tylko wątki o mniejszy id. Skoro wątki są różnowartościowe to istnieje dokładnie 1 najmniejszy wątek, który w ogóle nie będzie czekał w tej pętli. Czyli nie dojdzie do zakleszczenia. Analogicznie dla drugiej pętli, gdzie największy wątek wejdzie do sekcji krytycznej bez żadnego sprawdzania.

c) niezagłodzenie:
Załóżmy dwa wątki (0) i (1). Niech wątek (0) szybko wchodzi i opuszcza sekcje krytyczną, a wątek (1) późno reaguje na zmianę flag. Wątek (0) będzie ciągle wchodzić do sekcji krytycznej zanim zrobi to wątek (1), co będzie skutkowało zagłodzeniem (1).

Zad. 3

## Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is
  - **Linearizable™**

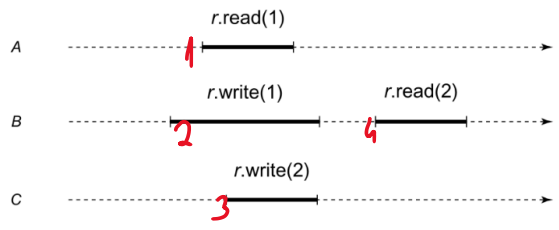


Diagram 1 (zapis/odczyt współdzielonej komórki pamięci r)

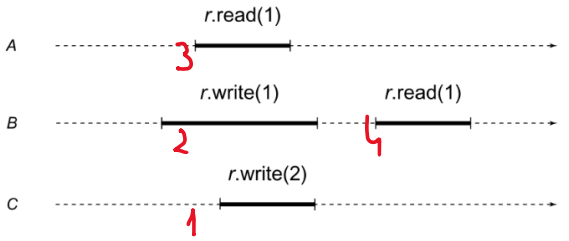


Diagram 2 (zapis/odczyt współdzielonej komórki pamięci r)

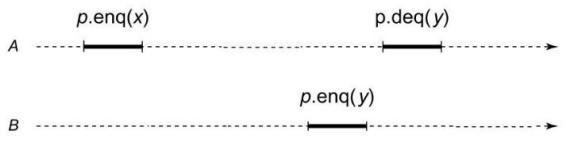


Diagram 3 (p jest kolejką FIFO)

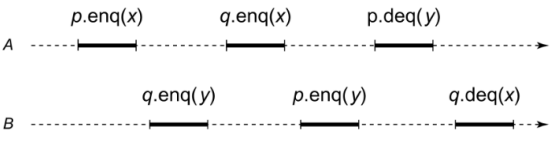


Diagram 4 (p i q są kolejkami FIFO)

1. Linearyzowane.
2. Linearyzowane.
3. Najpierw trzeba ściągnąć ‘x’ z p, żeby móc zdjąć ‘y’. Nielinearyzowane.
4. Analogiczna sytuacja jak w 3. – kolejka q jest symetryczna do p. Nielinearyzowane.

Zad. 4

## Linearizability

- History H is ***linearizable*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that G is equivalent to
  - Legal sequential history **S**
  - where $\rightarrow_G \subset \rightarrow_S$

1.
Historia G:
B: r.write(1)
A: r.read(1)
C: r.write(2)
A: r: 1
C: r: void
B: r: void
B: r.read(2)
B: r: 2

Historia S:
B: r.write(1)
B: r: void
A: r.read(1)
A: r: 1
C: r.write(2)
C: r: void
B: r.read(2)
B: r: 2

→G = {
B r.write(1) -> B r.read(2),
A r.read(1) -> B r.read(2),
C r.write(2) -> B r.read(2)}

→S = {
B r.write(1) -> A r.read(1) ->
C r.write(2) -> B r.read(2)}

2.
Historia G:
B: r.write(1)
A: r.read(1)
C: r.write(2)
A: r: 1
C: r: void
B: r: void
B: r.read(1)
B: r: 1

Historia S:
C: r.write(2)
C: r: void
B: r.write(1)
B: r: void
A: r.read(1)
A: r: 1
B: r.read(1)
B: r: 1

→G = {
C r.write(2) -> B r.read(1),
B r.write(1) -> B r.read(1),
A r.read(1) -> B r.read(1)}

→S = {
C r.write(2) -> B r.write(1) ->
A r.read(1) -> B r.read(1)}

3.
Historia G:
A: p.enq(x)
A: p: void
B: p.enq(y)
B: p: void
A: p.deq()
A: p: y
G nie odpowiada żadnej legalnej sekwencyjnej historii S, ponieważ p.enq(x) -> p.enq(y) -> p.deq(y), jest sprzeczne.

4.
Historia G:
A: p.enq(x)
A: p: void
B: q.enq(y)
B: q: void
A: q.enq(x)
A: q: void
B: p.enq(y)
B: p: void
A: p.deq()
A: p: y
B: q.deq()
B: q: x

Znów G nie odpowiada żadnej legalnej sekwencyjnej historii S, ponieważ mamy:
p.enq(x) -> p.enq(y) -> p.deq(y)
oraz
q.enq(y) -> q.enq(x) -> q.deq(x)
Co jest niemożliwe.

Zad. 5

```
public class MergeSort implements Runnable {
   protected int array[];
   protected int helperArray[];
   protected int left, right;

   MergeSort(int array[], int helperArray[], int left, int right) {
      this.array = array;
      this.helperArray = helperArray;
      this.left = left;
      this.right = right;
   }
   private void merge (int left, int middle, int right) {
          int leftIndex = left
          int rightIndex = middle + 1
          int index = left;

          for (int i = left; i <= right; i++) {
          helperArray[i] = array[i];
           }
          while (leftIndex < middle + 1 && rightIndex < right + 1) {
                    if (helperArray[leftIndex] <= helperArray[rightIndex]) {
                              array[index] = helperArray[leftIndex];
           leftIndex += 1;
        } else {
          array[index] = helperArray[rightIndex];
           rightIndex += 1;
        }
        index += 1;
      }
      while (leftIndex < middle + 1) {
           array[index] = helperArray[leftIndex];
        index += 1;
        leftIndex += 1;
      }
      while (rightIndex < right + 1) {
           array[index] = helperArray[rightIndex];
        index += 1;
        rightIndex += 1;
      }
   }

   @Override
   public void run() {
      if (this.left < this.right) {
         int middle = (this.left + this.right) / 2;
         MergeSort left = new MergeSort(array, helperArray, this.left, middle);
         MergeSort right = new MergeSort(array, helperArray, middle + 1, this.right);
```

```
            Thread t1 = new Thread(left);
            Thread t2 = new Thread(right);
            t1.start();
            t2.start();
            try {
               t1.join();
               t2.join();
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            this.merge(this.left, middle, this.right);
         }
      }
   }

public class RookieMergeSort {
   public static void main(String[] args) {
      int arr[] = {7, 6, 3, 1};
      MergeSort w = new MergeSort(arr, new int[arr.length], 0, arr.length-1);
      Thread t = new Thread(w);

      t.start();
      try {
         t.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      for (int i = 0; i < arr.length; i++)
         System.out.printf("%d ", arr[i]);    }
}
```

Zad. 6

```
public void run() {
     if (right - left < 4) {
          java.util.Arrays.sort(arr, left, right + 1);
     }
     else {
           int m = (this.l + this.r) / 2;
           MergeSort left = new MergeSort(arr, help_arr, this.l, m);
           MergeSort right = new MergeSort(arr, help_arr, m + 1, this.r);
           Thread t1 = new Thread(left);
           t1.start(); right.run();
           try {
              t1.join();
           } catch (InterruptedException e) {
              e.printStackTrace();}
           this.merge(this.l, m, this.r);}
```

Zad. 7

```
public class MergeSort implements Runnable {
   protected int array[];
   protected int helperArray[];
   protected int left, right;

   public static final int MAX_THREAD = 10;
   private static int activeThreads = 1;
   private final Object lock = new Object();

   MergeSort(int array[], int helperArray[], int left, int right) {
      this.array = array;
      this.helperArray = helperArray;
      this.left = left;
      this.right = right;
   }
public void sort(int left, int right) {
      if (left < right) {
         int middle = (left + right) / 2;
         sort(left, middle);
         sort(middle + 1, right);
         merge(left, middle, right);
      }
   }
public void run() {
      if (this.left < this.right) {
                  int middle = (this.left + this.right) / 2;
                  Thread t1 = null;

                  synchronized(lock) {
                        if(activeThreads + 1  <=  MAX_THREAD) {
                              MergeSort left = new MergeSort(array, helperArray, this.left, middle);
                              t1 = new Thread(left);
                              activeThreads += 1;
                        }
          }

                  Thread t2 = null;

                  synchronized(lock) {
                        if(activeThreads + 1  <=  MAX_THREAD) {
                              MergeSort right = new MergeSort(array, helperArray, middle + 1,
                              this.right);
                              t2 = new Thread(right);
                              activeThreads += 1;}}
         if(t1 != null) {
           t1.start();
         }else {
```

```
            sort(this.left, middle);
          }
          if(t2 != null) {
            t2.start();
          }else {
            sort(middle + 1, this.right);
          }
          try {
            if(t1 != null) {
                        t1.join();}
            if(t2 != null) {
                        t2.join(); }
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
          this.merge(this.left, middle, this.right);

      }          }
      }
      synchronized (lock) {
         activeThreads -= 1;
      }
    }
}
```

Zad. 8

TODO

Zad. 9

```
class BakeryOpt implements Lock {
            boolean flag[];
            int label[];
            int counter;
            public Bakery(int n) {
                        flag = new boolean[n];
                        label = new Label[n];
                        counter = 0;
                        for (int i = 0; i < n; i++) {
                                    flag[i] = false;
                                    label[i] = 0;
                        }
                        public void lock () {
                                    flag[i] = true; // i – numer bieżącego wątku
                                    label[i] = counter++;
                                    while (∃k flag[k] && (label[i], i) > (label[k], k)) { };
                        }
                        public void unlock () {
                                    flag[i] = false;
}}}
```

a) wzajemne wykluczanie:
   Instrukcja label[i] = counter++ nie jest atomowa. Istnieje ryzyko, że dwa wątki mogłyby wejść do sekcji krytycznej w tym samym czasie, jeśli przypisanie wartości z counter nie byłoby odpowiednio zsynchronizowane. Nie spełnia.
   1 i 0 uruchamiają lock jednocześnie. Oba odczytają ‘0’ z counter, przez co oba nie będą musiały czekać w pętli ‘while’.
b) niezakleszczenie:
   Z powodu potencjalnej niejednoznaczności etykiet, wątki mogłyby na siebie czekać bez końca. Nie spełnia.
c) niezagłodzenie:
   Ze względu na możliwość przypisania tej samej wartości etykiety wielu wątkom, może dojść do sytuacji, w której pewien wątek zostanie wielokrotnie pominięty, a inne będą przechodziły do sekcji krytycznej.