גיליון תשובות מספר נבחן:_____

על תשובות ריקות לשאלות הפתוחות יינתן 20% מהניקוד!

מחק את התשובה הנכונה ביותר. יש למחוק אך ורק תשובה אחת. מחיקה של יותר מתשובה אחת או השארת תושבה ריקה, לא תזכה בנקודות.

שאלה 1 (30 נקודות)

א - 1
ב - 3
ג - 5
ד - 5
ה - 2

שאלה 2 (30 נקודות)

להלן פתרון שאלה 2 - מומלץ לעיין בהערות לפני שמגישים ערעור.

<u>סעיף א</u>

0 <= size() <= capacity()

הערות:

- התכונה הנשמרת (invariant) מאפיינת את <u>המצב של האובייקט</u>, בקביעת אילוץ על הערכים של השדות <u>כפי שניתן ע"י השאילתות של הממשק</u>.
- לא ניתן לתאר את סמנטיקת ה LIFO של המחסנית בתכונה הנשמרת, כי היא קשורה לפעולות על המחסנית ולא על תוכן המחסנית ברגע נתון. (זה ייעשה בתנאי ההתחלה והסיום של push ו-pop).
- את התכונה הנשמרת יש לתאר אך ורק ע"פ השאילתות בממשק (size() ו capacity() במקרה שלנו). אם נדרש מידע נוסף לשם תיאור התכונה הנשמרת, יש להוסיף שאילתות (לא נדרש במקרה שלנו).

סעיף ב,ג באו לבדוק הבנה של מושג הבטיחות והנכונות, לא רק זיהוי סתמי ש'משהו לא בסדר'.

<u>סעיף ב</u>

מצב לא בטוח של אובייקט הוא מצב בו האבוייקט לא שומר על האינווריאנטה שלו.
ריצה לא בטוחה היא ריצה בה אחד האובייקטים אינו בטוח.

במקרה שלנו:

- T1 ו T2 מכירים אובייקט משותף מסוג LinkedStack, כאשר **size() == 1, capacity() == 2**
- T1 ו T2 מבצעים במקביל **push()**, ועוברים לסירוגין את הבדיקה **if (size < capacity)**
- T1 ו T2 מבצעים את שאר המתודה, תוך העלאת תכולת המחסנית, ו size() בהתאם, ל-3, כך ש size() > capacity(), בניגוד לתכונה הנשמרת. כך שהאובייקט אינו בטוח והריצה אינה בטוחה.

סעיף ג

הרצה מקבילת לא נכונה של קבוצת משימות נתונה הינה הרצה המובילה לתוצאה שאינה אפשרית בשום הרצה סדרתית של המשימות בסדר כלשהו.

נדרש לתאר תרחיש של הרצה בטוחה אך לא נכונה, כלומר הרצה בה האינווריאנטה נשמרת אך התוצאה אינה אפשרית בהרצה סדרתית של המשימות בסדר כל שהוא:

- נתון אובייקט מסוג **LinkedStack<Integer>**, כאשר **size() == 0**, **capacity() == 10**
- נתונות שתי משימות: משימה א' – ביצוע **push(1)**, משימה ב' – ביצוע **push(2)**.
- בהרצה סדרתית של משימות א,ב יהיו בראש המחסנית 1,2; ובהרצה סדרתית בסדר הפוך של משימות ב,א יהיו בראש המחסנית 2,1.

בהרצה מקבילית, בה כל ת'רד מבצע את אחת המשימות, ייתכן מצב שבו בראש המחסנית יהיה רק 1 או רק 2 - אם שני הת'רדים מגיעים במקביל לשורה **head = new Link<>(head, data)**, כך שהת'רד השני 'דורס' את ה head של הת'רד הראשון (הריצה בטוחה כי **size()** לא חרג מ **capacity()**)

הערה:

- תשובות שהתבססו על קריטריון נכונות בו ההרצה המקבילית חייבת להתאים להרצה סדרתית בסדר אחד בלבד של המשימות (כמו ביצוע push ואח"כ pop) הינן שגויות. אם נדרש לבצע push ורק אח"כ pop אין כל טעם להריץ שתי פעולות אלו כשתי משימות מקבילות. ואם עושים זאת.
- ירדו 4 נקודות למי שנתן תרחיש כזה, כאשר כל ת'רד משלים לבד את ביצוע ה push/pop שלו ללא הפרעה (מאחר ומדובר בחוסר הבנה מוחלט של קריטריון הנכונות). במידה והיתה התייחסות לכך ששני הת'רדים נכנסים יחד למתודה ירדה, לפנים משורת הדין, רק נקודה אחת.
- מי שציין בעיה באי קיום תנאי הסיום של המתודה, קיבל את מלוא הנקודות.

סעיף ד

```
class Link<T> {

  public Link<T> next;
  public T data;
  public int size;

  public Link(Link<T> next, T data) {
    this.next = next;
    this.data = data;
    this.size = 0;
  }

  public Link(Link<T> next, T data, int size) {
    this.next = next;
    this.data = data;
    this.size = size;
  }
```

```
}

class ConcurrentLinkedStack<T> implements Stack<T> {

   private AtomicReference<Link<T>> head;
   private final int capacity;

   ConcurrentLinkedStack(int capacity) throws Exception {
      if (capacity < 1)
         throw new Exception("Illegal capacity: " + capacity);
      this.capacity = capacity;
      this.head = null;
    }

   public void push(T data) {
      Link<T> localHead;
      Link<T> newHead = new Link<>(null, data);
      do {
         localHead = head.get();
         if (localHead != null && localHead.size == capacity)
            return;
         newHead.next = localHead;
         newHead.size = localHead.size + 1;
      } while (!head.compareAndSet(localHead, newHead));
   }

   public T pop() {
      Link<T> localHead = null;
      Link<T> newHead = null;
      do {
         if (head == null || head.get().size == 0)
            return null;
         localHead = head.get();
         newHead = head.get().next;
      } while (!head.compareAndSet(localHead, newHead));
      return  localHead.data;
   }

   public int size () {
     return  head  == null ? 0 : head.get().size;
   }

   public int capacity () {
      return capacity;
   }
}
```

הערות:

- מחלקות ה Lock של Java מעבירות את הת'רד למצב blocked אם האובייקט נעול. ההחלטה האם לעשות זאת עם synchronized או עם מנגנון אחר היא תלויית מימוש. לפנים משורת הדין, ירדו רק 5 נקודות למי שהשתמש במחלקות אלו (במידה והשתמש בהם כיאות, ולא כתב סתם Lock(stack), או דברים מוזרים מעין אלו).
- wait/notify דורשים סנכרון. מי שהשתמש בהם לא מילא למעשה אחר הדרישה הבסיסית בשאלה.
- יש לדאוג לכך ש size יהיה תואם למספר האובייקטים במחסנית (בפיתרון המוצע זה ממומש ע"י הוספת השדה size למחלקה Link, כך שהוא מתעדכן בפעולה אטומית אחת עם הוספת/הסרת החוליה).
- יש לדאוג לכך שהאובייקט יתווסף/יוסר מהמחסנית בעקבות ביצוע push/pop (כפי שנלמד בכיתה), לא ניתן לדלג על הפעולה אם המחסנית מטופלת כרגע על ידי ת'רד אחר.

שאלה 4 (10 נקודות)

א - 3
ב - 4

פתרון לשאלה 3:

א. פתרונו של שרגא לא יעבוד. מכיוון שאובחן שהעומס נובע מהתעבורה ניתן להסיק שהפעולות הבעייתיות שהסלקטור מבצע הן פעולות הIO ובעיקר פעולות הקריאה והכתיבה. הוספה של workers וליבות חדשות בשבילם לא יפתרו את בעיה זו כיוון שהם לא מבצעים פעולות IO כלל ולכן הטרד (היחיד) של הסלקטור ישאר עם לפחות אותה כמות של עבודה.

ב. פתרונו של נחום יעבוד שכן הוא מחלק את הפעולות הבעייתיות - הקריאות והכתיבות על פני 2 טרדים ולא אחד

ג. ישנם מספר דרכים לפתור את הבעיה - קיבלתי המון פתרונות לא מושלמים (בלשון המעטה), לא קיבלתי פתרונות שלא הראו הבנה מינימלית בשימוש בטרדים ותקשורת

להלן פתרון אפשרי אחד שהיה מקבל את מלוא הנקודות:

```
public class Reactor {
... the reactor's fields and constructor

//Change: Constructor added
public Reactor(ExecutorService pool, Selector selector) {
  this.pool = pool;
  this.port = -1;
  this.protocolFactory = null;
  this.readerFactory = null;
  this.selector = selector;
}

public void serve() {
  selectorThread = Thread.currentThread();
  Selector selector2 = null; //Change: local variables added
  ServerSocketChannel serverSock = null;

  try {
     if (port >= 0) { //Change: remove try with resources, manage the resources manually and
start another thread
        selector2 = Selector.open();
        selector = Selector.open();
        serverSock = ServerSocketChannel.open();
        serverSock.bind(new InetSocketAddress(port));
        serverSock.configureBlocking(false);
        serverSock.register(selector, SelectionKey.OP_ACCEPT);

        Reactor second = new Reactor(pool, selector2);
        new Thread(() -> second.serve()).start();
     }
```

```
    while (!selectorThread.isInterrupted()) {

        selector.select();
        runSelectionThreadTasks();

        for (SelectionKey key : selector.selectedKeys()) {

            if (!key.isValid()) {
                continue;
            } else if (key.isAcceptable()) {
                //Change: work split between selectors
                handleAccept(serverSock, selector.keys().size() > selector2.keys().size() ?
selector2 : selector);
            } else {
                handleReadWrite(key);
            }
        }

        selector.selectedKeys().clear();      }
  } catch (ClosedSelectorException ex) {
      //do nothing - server was requested to be closed
  } catch (IOException ex) {
      //this is an error
      ex.printStackTrace();
  }


  System.out.println("server closed!!!");
  pool.shutdown();

  //Change: Manually close the selectors
  try {
      selector.close();
      if (selector2 != null) selector2.close();
  } catch (IOException err) {
      err.printStackTrace();
  }
}
```

... rest of the Reactor functions with no additional changes