An Exploratory Investigation on High-School Students' Understanding of Threads — Survey Questions —

Emanuele Scapin[1][0000−0001−8384−8231], Nicola Dalla Pozza[1][0000−0003−2764−9603], and Claudio Mirolo[2][0000−0002−1462−8304]

[1] ITT G.Chilesotti, 36016 Thiene, Italy
`{escapin,ndallapozza}@chilesotti.it`

[2] University of Udine, 33100 Udine, Italy `claudio.mirolo@uniud.it`

Abstract. Students' difficulties to learn concurrent programming are well known amongst Computer Science instructors. While in the International Computing Education community it is still up to debate the extent to which such topic should be included in pre-university curricula, based on our country's Ministerial guidelines for technical high schools with a specialization in Computer Science, students are expected to acquire key concurrent programming skills. With the aim of getting insights about the nature of students' difficulties, as well as to identify possible pedagogical approaches to be adopted by teachers, we have undertaken an investigation on students' perception, proficiency and self-confidence when dealing with concurrency and synchronization tasks. We then present the results of a preliminary study carried out by submitting a survey in a couple of representative high schools of our area. The survey includes subjective perception questions as well as small program comprehension tasks addressing students' understanding of thread synchronization. Moreover, we also analyze students' self-confidence in connection with their actual performance in such tasks. A total of 68 high school students were engaged in the survey. Our findings indicate that students' perception of self-confidence tends to weakly correlate to their actual performance, although more in general they express a low self-confidence level in relation to the topic. In particular, the results clearly show that the concept of thread synchronization is especially difficult to master for a large majority of them.

Keywords: Informatics education · Programming learning · High school · Threads · Concurrent programming

Survey Questions

Approach to threads

- In general, how would you rate the difficulty of the thread topic?
  4–grade Likert scale (1=Not difficult – 4=Very difficult)

- How would you rate your performance when managing threaded applications?
  4–grade Likert scale (1=Not satisfied – 4=Very satisfied)

- In your opinion, is it adequate the amount of time that the teacher spends to introduce the thread topic?
  4–grade Likert scale (1=Not adequate – 4=Definitely adequate)

- In your opinion, are the examples and exercises that the teacher proposes to introduce the thread topic adequate?
  4–grade Likert scale (1=Not adequate – 4=Definitely adequate)

- Rate the level of difficulty you typically encounter when dealing with the following thread issues. (Mark only one option per row)
  Options: not known, usually simple, reasonably simple, difficult, very difficult.
  Topics: Class definition, Object shared between threads, Distinguishing shareable vs. non–shareable data, Thread "Run" method definition, Starting a thread, Closing a thread, Choice of class methods, Identification of shared class methods, Understand thread life cycle, Dealing with thread state, Synchronization (in general).

- Rate the level of difficulty you encounter when using the following methods for managing the state of a thread. (Mark only one option per row)
  Options: not known, usually simple, reasonably simple, difficult, very difficult.
  Methods: start, stop, sleep, suspend, wait, yield, join, resume, notify, notifyAll, synchronized.

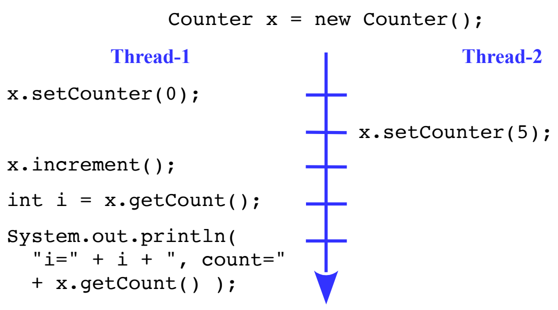
- Rate the level of difficulty you encounter when dealing with conditions between threads. (Mark only one option per row)
  Options: not known, usually simple, reasonably simple, difficult, very difficult.
  Operations: Read a shared resource, Write or modify a shared resource, Accidental resource sharing, Early release of a resource, Multiple Locks for the same resource, Missed protection of a shared resource, Synchronization of shared resources, Synchronization of methods that manage shared resources, Wait without wake–up notification (Notify).

Tasks

The code fragments formalized in Java for Task 1.a–d refer to the *Counter* class defined as follows:

```java
public class Counter {

  private int count = -1; // a negative value of count
                           // is interpreted as "undefined"

  public synchronized int getCount() {
    while ( count < 0 ) {
      try {
        wait();
      } catch ( Exception e ) {}
    }
    return count;
  }

  public synchronized void setCounter( int initValue ) {
    if ( initValue >= 0 ) {
      count = initValue;
      notify();
    }
  }

  public synchronized void increment() {
    while ( count < 0 ) {
      try {
        wait();
      } catch ( Exception e ) {}
    }
    count = count + 1;
  }

}  // Counter
```

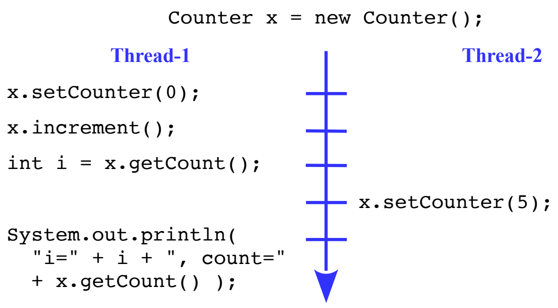
Task 1.a Analyze the execution of the following code snippets (Figure 1) for two separate threads, Thread-1 and Thread-2, operating on a shared instance x of the Counter class introduced above. The operations of each of the two threads are represented along opposite sides of the vertical axis, according to the time order (from top to bottom) in which the methods invoked in the instructions are executed; furthermore, no operations on x or i have been omitted in the reported flows. What are the output values printed during the execution of Thread-1? Mark only one option.
Options: i = 1, count = 1; i = 1, count = 5; i = 5, count = 5; i = 6, count = 6; The result cannot be predicted because there are several possibilities.



```
Counter x = new Counter();

Thread-1                              Thread-2
x.setCounter(0);
                                      x.setCounter(5);
x.increment();
int i = x.getCount();
System.out.println(
  "i=" + i + ", count="
  + x.getCount() );
```

Fig. 1: Task 1.a.

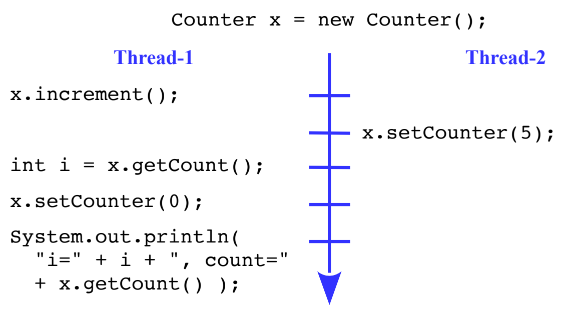
Task 1.b Analyze the execution of the following code snippets (Figure 2) for two separate threads, Thread-1 and Thread-2, operating on a shared instance x of the Counter class introduced above. The operations of each of the two threads are represented as described in question Task 1.a. What are the output values printed during the execution of Thread-1? Mark only one option.
Options: i = 1, count = 1; i = 1, count = 5; i = 5, count = 5; i = 5, count = 6; The result cannot be predicted because there are several possibilities.



```
Counter x = new Counter();

Thread-1                              Thread-2
x.setCounter(0);
x.increment();
int i = x.getCount();
                                      x.setCounter(5);
System.out.println(
  "i=" + i + ", count="
  + x.getCount() );
```

Fig. 2: Task 1.b

Task 1.c Analyze the execution of the following code snippets (Figure 3) for two separate threads, Thread-1 and Thread-2, operating on a shared instance x of the Counter class introduced above. The operations of each of the two threads are represented as described in question Task 1.a. What are the output values printed during the execution of Thread-1? Mark only one option.
Options: i = 0, count = 0; i = 5, count = 0; i = 6, count = 0; i = 6, count = 6; The result cannot be predicted because there are several possibilities.



```
Counter x = new Counter();
```

Thread-1:

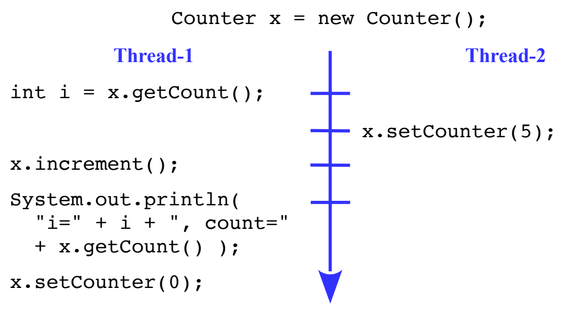
```
x.increment();

int i = x.getCount();
x.setCounter(0);
System.out.println(
  "i=" + i + ", count="
  + x.getCount() );
```

Thread-2:

```
x.setCounter(5);
```

Fig. 3: Task 1.c

Task 1.d Analyze the execution of the following code snippets (Figure 4) for two separate threads, Thread-1 and Thread-2, operating on a shared instance x of the Counter class introduced above. The operations of each of the two threads are represented as described in question Task 1.a. What are the output values printed during the execution of Thread-1? Mark only one option.
Options: i = 0, count = 0; i = 5, count = 6; i = 6, count = 6; i = -1, count = 6; The result cannot be predicted because there are several possibilities.



```
Counter x = new Counter();
```

Thread-1:

```
int i = x.getCount();

x.increment();
System.out.println(
  "i=" + i + ", count="
  + x.getCount() );
x.setCounter(0);
```

Thread-2:

```
x.setCounter(5);
```

Fig. 4: Task 1.d

Task 1: self-confidence level With regard to the previous questions (Task 1.a–d), rate your degree of confidence in the correctness of the solutions you have chosen on a scale from 1 to 4.
4–grade Likert scale (1=Not confident at all – 4=Fully confident)

Task 2: Consider the classes defined below (Figure 5) and assume to start the program through the *main* method of the Task2 class. Which of the proposed sequences will be printed at the end of the execution? Mark only one option.
Options: P3P7P5; P3PP7PP5P; PP3P5P7; PP3PP7PP5; PPP375; PPPPPP375; The program hangs in a deadlock; The result cannot be predicted because there are several possibilities.

```java
class Adder extends Thread {

  private int loops;
  private Vector<Integer> buffer;  // integer sequence

  public Adder( int loops, Vector<Integer> buffer ) {

    this.loops = loops;
    this.buffer = buffer;
  }

  public void run() {

    for ( int i=0; i<loops; i=i+1 ) {

      synchronized ( buffer ) {
        while ( buffer.size() < 2 ) {
          try {
            buffer.wait();
          } catch ( Exception e ) {}
        }
        // n: adds the first two buffer elements
        int n = buffer.get(0) + buffer.get(1);
        buffer.clear();  // buffer is emptied
        System.out.print( ""+n );
        buffer.notify();
    }}
    System.out.println();
  }

}  // Adder
```

```java
class Provider extends Thread {

  private int[] stream;
  private Vector<Integer> buffer;

  public Provider( int[] stream, Vector<Integer> buffer ) {

    this.stream = stream;
    this.buffer = buffer;
  }

  public void run() {

    for ( int i=0; i<stream.length; i=i+1 ) {

      int x = stream[i];
      synchronized ( buffer ) {
        while ( buffer.size() == 2 ) {
          try {
            buffer.wait();
          } catch ( Exception e ) {}
        }
        System.out.print( "P" );
        buffer.add(x);  // new element into buffer
        buffer.notify();
      }
    }
  }
}  // Provider
```

```java
public class Task2 {

  public static void main( String[] args ) {

    int[] stream = new int[] { 1, 2, 3, 4, 3, 2 };

    // buffer initially empty
    Vector<Integer> buffer = new Vector<Integer>();
    Adder adder = new Adder( 3, buffer );
    Provider provider = new Provider( stream, buffer );

    adder.start();
    provider.start();
  }

}  // Task2
```

Fig. 5: Task 2

Task 2: self-confidence level With regard to the previous question (Task 2), rate your degree of confidence in the correctness of the solution you have chosen on a scale from 1 to 4.
4–grade Likert scale (1=Not confident at all – 4=Fully confident)

Task 3: Within a class describing the implementation of a shared resource, which of the following methods' definitions (Figure 6) can help to avoid conflicts in the management of the resource itself?

```
public synchronized int getValue() {
  return value;
}
public void setValue(int someValue) {
  value = someValue;
}
public void increment() {
  value++
}
```

Option 1

```
public synchronized int getValue() {
  return value;
}
public synchronized void
         setValue(int someValue) {
  value = someValue;
}
public synchronized void increment() {
  value++
}
```

Option 3

```
public int getValue() {
  synchronized (this) {
    return value;
  }
}
public void
  setValue(int someValue) {
  synchronized (this) {
    value = someValue;
  }
}
public void increment() {
  synchronized (this) {
    value++
  }
}
```

Option 5

```
public int getValue() {
  return value;
}
public synchronized void
         setValue(int someValue) {
  value = someValue;
}
public void increment() {
  value++
}
```

Option 2

```
public int getValue() {
  return value;
}
public void setValue(int someValue) {
  value = someValue;
}
public void increment() {
  value++
}
```

Option 4

Fig. 6: Task 3. Equivalence: *Select all applicable items.*

Task 3: self-confidence level With regard to the previous question (Task 3), rate your degree of confidence in the correctness of the solution you have chosen on a scale from 1 to 4.
4–grade Likert scale (1=Not confident at all – 4=Fully confident)

Task 4: Consider an instance of the *Bouncer* class defined below. The synchronization modes of the *from1to2* and *from2to1* methods can lead to deadlock situations.

```
public class Bouncer {

  private Vector<Integer> seq1;
  private Vector<Integer> seq2;

  public Bouncer( Vector<Integer> seq1, Vector<Integer> seq2 ) {

    this.seq1 = seq1;
    this.seq2 = seq2;
  }

  public void from1to2() {
    synchronized ( seq1 ) {
      if ( seq1.size() == 0 ) {
        try {
          seq1.wait();
        } catch ( Exception e ) {}
      }
      int item = seq1.elementAt(0);
      seq1.removeElementAt(0);
      synchronized ( seq2 ) {
        seq2.add( item );
        seq2.notify();
      }
    }
  }

  public void from2to1() {
    synchronized ( seq2 ) {
      if ( seq2.size() == 0 ) {
        try {
          seq2.wait();
        } catch ( Exception e ) {}
      }
      int item = seq2.elementAt(0);
      seq2.removeElementAt(0);
      synchronized ( seq1 ) {
        seq1.add( item );
        seq1.notify();
      }
    }
  }

}  // Bouncer
```

Which of the following workarounds will fix the code to prevent the occurrence of a deadlock (while still ensuring proper synchronization)?
Mark only one option.
Options: delete all synchronized; eliminate nested synchronized; drop synchronized by either method; drop the outer synchronized from one of the methods and the nested one from the other; transform nested synchronized into sequenced synchronized (one after the other rather than one within the other); reverse seq1 and seq2 in all synchronized constructs; none of the previous solutions.

Task 4: self-confidence level With regard to the previous question (Task 4), rate your degree of confidence in the correctness of the solution you have chosen on a scale from 1 to 4.
4–grade Likert scale (1=Not confident at all – 4=Fully confident)

Possible help tools

- Have you ever thought about a graphical representation of thread working principles, in order to ease its understanding?
  4–grade Likert scale (1=Never – 4=Often)

- Do you think a graphical representation of how threads work could be effective in improving your understanding?
  4–grade Likert scale (1=Not effective – 4=Very effective)

- How would you rate the following graphing tools in the context of threads? (Mark only one option per row)
  Options: I don't know this type of representation, not very useful, partially useful, quite useful, very helpful.
  Tools: flow–charts, Petri nets, finite state automata, Cartesian diagrams as a function of time, Unified Modeling Language (UML), Holt graphs, block diagrams.

Final open question

- What would you suggest to make the lessons on threads more interesting and clearer? (Open answer)