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

Emanuele Scapin[1(✉)], Nicola Dalla Pozza[1], and Claudio Mirolo[2,3]

[1] ITT G.Chilesotti, 36016 Thiene, Italy
{escapin,ndallapozza}@chilesotti.it
[2] University of Udine, 33100 Udine, Italy
claudio.mirolo@uniud.it
[3] Lab. CINI "Informatica e Scuola", Rome, Italy

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

1 Introduction

Over the last decades, in order to achieve ever increasingly powerful architectures, the processor industry has shifted its manufactures towards multi–core

J.-P. Pellet and G. Parriaux (Eds.): ISSEP 2023, LNCS 14296, pp. 93–110, 2023.
https://doi.org/10.1007/978-3-031-44900-0_8

and many–core designs. This trend has become so widely adopted that nowadays such hardware is employed not only for industrial and academic purposes [8], but it is also readily available in desktop machines in school laboratories [2]. Alongside to the electronic improvements, to harness the computational power of these machines it is necessary to design software solutions that exploit the numerous cores with concurrent programs.

There is therefore the need for experienced programmers leveraging this paradigm, as indicated, in particular, by the ACM/IEEE Task Force in the Computer Science Curricula 2013 Report [12]. Academic institutions often introduce concurrent programming in advanced dedicated courses or in conjunction with related topics, e.g. [6,11,21], but less commonly in introductory programming modules [3,13]. As a matter of fact, it is still up to debate when and how to approach the subject, and even more open to debate is the feasibility to cover concurrent programming in the high school curriculum [1,2,14,23]. Students' difficulties to learn concurrent programming are indeed well known among both high school teachers and university instructors, and are also confirmed by a handful of empirical investigations, such as [2,10,14].

According to the Italian Ministerial guidelines, the students opting for a technical specialization in Computer Science are expected to develop parallel and concurrent programming skills by the end of their fourth high school year (grade 12).[1] Unfortunately, however, there is a lack of empirical evidence to set realistic curricular goals. As pointed out by Brođanac et al. in a recent paper, "[i]n the case of teaching parallel programming before university level, the research appears to be scarce" [2, p. 2].

Within the outlined framework, this work is meant as an initial, exploratory investigation on students' proficiency and self-confidence when dealing with concurrency and synchronization tasks. More specifically, we are trying to find at least some preliminary answers to the following research questions:

Q1. To what extent are students at ease with some basic concepts of concurrent programming?
Q2. To what extent does students' perception of self-confidence correlate with their actual performance in simple concurrent programming tasks?
Q3. What are their major difficulties when learning concurrent programming?

In order to address the above questions, we have administered a survey including subjective perception questions as well as four program comprehension tasks involving basic aspects of thread synchronization. A total of 68 students attending the last year (grade 13) in representative high schools of our area were engaged in the survey. The students were introduced to concurrent programming in the previous school year, by different teachers who may have used diversified approaches.

The paper is organized as follows. In Sect. 2 we summarize the background of this work. Section 3 is about the survey structure and the rationale underlying its design. The main findings of our exploratory investigation are then outlined

[1] https://www.gazzettaufficiale.it/eli/gu/2012/03/30/76/so/60/sg/pdf.

in Sect. 4 and discussed in Sect. 5. Finally, in Sect. 6 we draw some conclusions and mention possible future developments of the present work.

2 Background

At least anecdotally, students' difficulties with concurrent programming are acknowledged by technical high school Computer Science teachers, as well as by lecturer at the undergraduate and graduate levels. Nevertheless, this subject has hardly been given significant attention in the context of introductory programming. For instance, in their systematic study [17] reviewing over 700 research papers published between 2003 and 2018, Luxton-Reilly and colleagues mention only two contributions addressing parallel or concurrent programming.

In the following two subsections we will mention a selection of works specifically focused on the topic, respectively at the tertiary and secondary instruction levels, whereas for a broader literature analysis the reader is referred to the recent review discussed in Brođanac et al.'s paper [2]. We will then conclude this background section with a couple of notes about the Ministerial guidelines that apply to our school system.

2.1 Parallel and Concurrent Programming at the University Levels

A range of works on the learning of concurrent programming at the undergraduate or graduate levels examine students' performance, their common difficulties, their understanding of and misconceptions on the subject. For instance, Choi and Lewis [5] analyze and classify the pitfalls in a collection of simple multi-threaded programs written by students in order to improve instruction and develop learning aids. While focusing on thread-safe Java classes, Fekete [7] identifies suitable learning outcomes and discusses related pedagogical difficulties, also proposing examples that could help students avoid common misconceptions. Lönnberg & Berglund [16] investigate the defects of concurrent programs produced by students from a program development perspective.

A recurrent theme at the undergraduate level is whether these topics should be taught in a dedicated course [16,21], split into multiple units [11,13], or covered in an introductory programming course [3,13]. In this respect, Zhu et al. [25] argue that the conceptual shift from sequential programming to concurrent and parallel programming is notoriously difficult to make. The authors present their results of using the educational game Parallel, focusing on the learners' self-efficacy and how they learn concurrency concepts, and show that undergraduates' self-efficacy correlates with the time students spend in multi-threaded problem-solving. Formerly, Bruce et al. [3] had suggested making use of graphics and animations in order to facilitate student learning through visual feedback.

No matter how challenging the subject is, Gardner argues that, in light of the current developments of multi-core architectures, refraining "from teaching parallel programming to CS undergraduates is a kind of educational malpractice,"

since computer technology cannot be expected "to turn back to the old, comfortable path of ever-increasing uniprocessor clock speeds [...]. To be prepared for careers in this emerging environment, our students need to be furnished with the knowledge and practice of parallel programming" [9, p. 3:6]. And Rivoire [21] reported that upper-level undergraduates were indeed interested in and satisfied with the contents of an introductory course on multi-core programming models. However, according to Ko et al.'s teaching experiences in multiple courses [13], 1st and 2nd year students can recognize parallelism in programming tasks and are generally aware of synchronization issues (although would prefer to approach this subject in later years). In addition, Conte et al. [6] claim that the achievements in parallel programming of novices without previous exposure to computing concepts appear to be comparable to those of more advanced students.

2.2 Parallel and Concurrent Programming in High School

Although concurrent programming has usually been considered exceedingly challenging in a pre-tertiary context, sporadic attempts to introduce this topic in high school date back to the mid '90s. According to Rifkin, for instance, "it is never too early to teach so-called 'hard' concepts" such as basic principles of parallel algorithms and software engineering, providing the ideas are presented "in a manner that is simple, fun and suited to the audience" [20, p. 26].

Much work on the teaching and learning of concurrent programming in the upper secondary school has been done by the Israeli CS Education community. In particular, in the context of a high school unit in concurrent and distributed computing, Ben-Ari & Kolikant [1] explored the evolution of students' conceptions and attitudes, and found that they were eventually able to develop parallel algorithms and to prove their correctness. Although the involved students initially felt extremely challenged, they then came to appreciate the relevance of the topic and its contribution to improving their cognitive skills. Later, Kolikant observed that, although high school students are able to gain a "rich" understanding of various synchronization tasks, quite often their successful solutions to synchronization problems are achieved by a pattern-based approach "exempting them from dealing with the dynamics of the synchronization mechanisms," so that the underlying "concepts become inert" [14, p. 243].

Also in Tobert et al.'s view [23] there is ample evidence that parallel algorithmic thinking and multi-threading can be taught—and should be broadly covered in CS education—as early as high school. Moreover, Brođanac et al. [2] have recently conducted an investigation in 3 Croatian high schools where parallel programming was included in the informatics curriculum. They report getting positive feedback from students as to the interest and usefulness of the learnt content, even though it was perceived as more difficult compared to the other topics. The authors conclude that parallel programming, including synchronization issues, can be taught in a high school context at least as an optional subject.

2.3 High School Ministerial Guidelines in Italy

We conclude the background section by briefly summarizing the implications of the Ministerial guidelines framing our Country's secondary school system. Rather than prescribing a detailed syllabus, such recommendations are meant to be a reference for the teaching of each covered subject. In the case of technical high schools with specialization in Computer Science, the guidelines are articulated in terms of *knowledge*, *skills* and *competencies* to be achieved in the second biennium (grades 11 and 12) and in the last year (grade 13).

1. General information (2 questions) gender; subjective preference of programming languages for use with threads
2. Thread-related concepts in general (7 subjective perception questions) difficulty of the subject in general; self-efficacy in general; adequacy of time spent on the subject; adequacy of program examples; difficulty with specific thread-related concepts/issues; difficulty with specific thread-related (technical) program tools; difficulty with thread synchronization
3. Tasks (7 program comprehension questions and 4 evaluations of self-confidence on the provided answers) Task1 – mutual exclusion and race conditions (4 related questions); Task2 – producer-consumer scheme; Task3 – synchronization analysis; Task4 – deadlock analysis
4. Possible learning aids (3 subjective perception questions) whether graphical representations have ever been considered; expected potential of graphical representations; perception of the use of specific graphical representations
5. Open suggestions (1 question) suggestions to make the lessons on threads more interesting and clearer

Fig. 1. Structure of the survey. More details in Appendix: Survey Questions

As to the second biennium, in terms of knowledge students are expected to learn the components of operating systems, including techniques and technologies for concurrent programming and for the synchronized access to shared resources; in terms of skills, they should eventually be able to design and develop applications that interact with the operating system, using whenever appropriate concurrent programming strategies. In their last high school year, students should acquire knowledge on methods and technologies for network programming, as well as on protocols and communication languages of the application layer; in terms of skills, they are trained to develop applications that leverage network communication, such as client-server applications through simple communications protocols.

Within this framework, while teachers still have the freedom to personalize the course organization, concurrent programming is nonetheless considered an essential part of the high school curricula in Computer Science.

3 Instrument

In this section we outline the survey structure and the rationale underlying its design. The general organization, summarized in Fig. 1, is partly drawn from a similar instrument developed and used by the authors to get insight about the learning of other programming concepts [22]. Overall, it includes 24 items, 11 of which are based on 4 small tasks—7 program comprehension questions and 4 evaluations of self-confidence on the provided answers in a 4-grade Likert scale. The full survey is reported in the Appendix: Survey Questions.

The questions addressing the learning of concurrency concepts elaborate on Choi and Lewis' [5] "catalog" of errors that students typically make when approaching multi-thread programs, particularly in connection with data race conditions, deadlocks and other synchronization issues. To accommodate for the common practice in the involved high schools, the code is Java-like. The first two tasks draw inspiration from programs proposed by Meyer at al. in *"Concurrent Programming with Java Threads"*,[2] and are intended to test students' ability to anticipate the outcomes of concurrent threads. The other two tasks elaborate on Fekete's work [7]: the third task is meant to see if students are able to identify, among 5 options, both (equivalent) appropriate code fragments to deal with a shared resource; the fourth one requires to recognize deadlock-prone code and figure out suitable corrections.

For the solutions of each of the four tasks, the students were also asked to indicate their perceived level of self-confidence in a Likert scale ranging from 1 (not confident at all) to 4 (fully confident). We included these items because of the potential pedagogical implications pointed out, e.g., in [18,19], as well as in order to assess the extent to which students' subjective perception of difficulty correlates with their actual performance on a task. The concluding survey sections concern potential graphical/visual tools that could help understand concurrent programming and possible additional suggestions to improve the instructional practice. A particular interest in graphical/visual tools is motivated by recent research findings suggesting that students of STEM domains are more likely to exhibit a visual-spatial cognitive style [24].

4 Results

Data Collection

The survey was administered to 68 fifth-year students (grade 13: age range 18–19; 60 boys and 8 girls) attending a CS curriculum in two technical high schools in the North-East of Italy. The students, who were taught concurrent programming in the previous school year, engaged in the task in a controlled situation, under the supervision of their teachers. They were expected to complete the survey

[2] Course material available at the link https://se.inf.ethz.ch/courses/2011a_spring/soft_arch/lectures/old/13_softarch_self_study_threads.pdf.

within an hour. The answers to the survey were registered only anonymously, and then could not be used for formative or summative assessment.[3]

General Thread-Related Concepts

Here we summarize the results of survey Sect. 2 (see Fig. 1). To begin with, more than two thirds of the students find thread programming difficult (57%) or very difficult (12%) and report to be unsatisfied (54%) or completely unsatisfied (13%) with their performance in thread-related tasks. Conversely, most of them consider the time spent to deal with the subject (67%) as well as the proposed programming tasks (63%) to be both adequate to the purpose.

At a finer level of granularity, the bar chart in Fig. 2 shows students' perceived difficulty for diverse thread-related concepts. As can be seen, *synchronization*

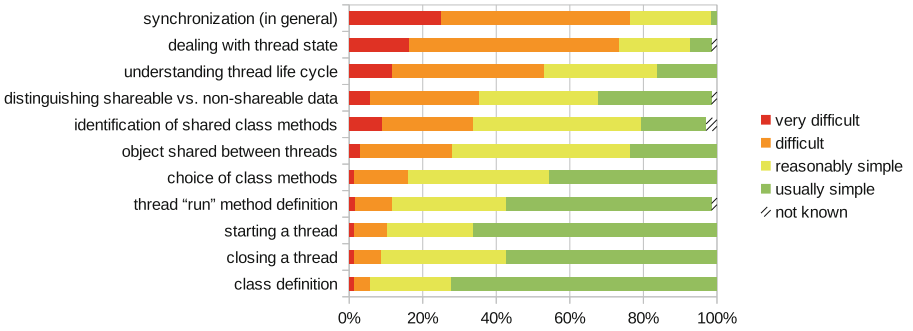

Fig. 2. Students' difficulties with a range of thread-related concepts.

and dealing with *thread states* are regarded as difficult to learn by 77% and 74% of them, respectively, whereas several aspects in connection with the organization of the classes implementing threads seem less critical. Although to a lesser degree, the answers to the following question, using a similar Likert scale and concerning a range of specific technical program tools, essentially confirm this picture: looking at the hardest side, the Java keyword `synchronized` turns out to be difficult for about 40% students and the methods `wait` and `notify`/`notifyAll`, governing the thread state, for roughly one third of them. The last question of survey Sect. 2 focuses on managing shared resources. All such related aspects are perceived as difficult by a significant percentage of students, ranging from about 30% (reading/writing operations) to about 60% (synchronization operations).

[3] In particular, since no personal information was shared with any third party, the Italian research policies do not require the approval by an ethics commission.

Program Comprehension Tasks

In summary, the first task presented a very simple class aimed at synchronizing the access to a shared resource, based on availability of data. Four temporal sequences of methods invocations by two concurrent threads were then presented, with a request to identify the resulting outcomes (questions a–d). The second, more complex task was about an instance of the *producer-consumer* scheme; again, students were asked to identify the correct output. In the third task, the (two) sound implementations of a straightforward synchronization scheme were to be recognized among five options. Finally, the last problem asked to choose a suitable strategy, informally described in words, to fix a given deadlock-prone code. (The full text of these tasks can be found in the appendix.)

In Fig. 3a are reported the percentages of (fully) correct answers to the seven questions asked for tasks 1–4. As we can see, only the solution of subtasks 1a and 1b are correct for a large majority of students; in all the other cases, less than half of them was successful, with the worst performances taking place for the apparently easy subtasks 1c and 1d. Figure 3b depicts the overall distribution of self-confidence levels between correct and incorrect answers. In this case, since there was one such Likert evaluation for each task, the solution of task 1 is considered correct when all four related answers are correct. Besides evidencing that less than one third of the students are more confident than not about their answers, what once again confirms their difficulties with concurrent programming, the diagram clearly shows that students' self-confidence in the provided solutions is only weakly connected with their actual achievements.

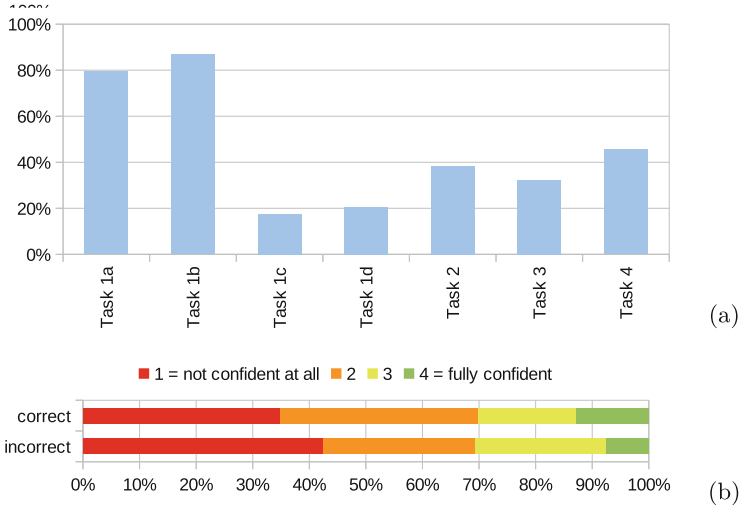

Fig. 3. Students' performance in the proposed tasks and perceived self-confidence in connection with their correct and incorrect answers.

We also tried to look in more depth at the relationships between quality of students' answers and perceived levels of self-confidence. To this aim, we scored the performance on task 1 in terms of number of correct solutions to questions a–d, and the performances on the other tasks by distinguishing three quality levels: 1 (= severely incorrect), 2 (= incorrect) and 4 (= correct) for tasks 2 and 4; 1 (= incorrect), 3 (= partly correct) and 4 (= fully correct) for task 3. The correlations and the corresponding statistical significance are summarized in Table 1, which should be self-explanatory.

Table 1. Correlation between students' performance in the tasks and their perception of self-confidence in the provided answers (in a Likert scale 1–4); the correlation cannot be taken as statistically significant if p-value > 0.05.

Task	correlation	p-value
Task 1 (number of correct answers to questions a–d)	0.278	0.0215
Task 2 (4 = correct, 2 = incorrect, 1 = severely incorrect)	0.440	< 0.0002
Task 3 (4 = correct, 3 = partly correct, 1 = incorrect)	0.150	0.2234
Task 4 (4 = correct, 2 = incorrect, 1 = severely incorrect)	0.017	0.8917
Task 1–4 (average score vs. average self-confidence level)	0.365	0.0022

Possible Learning Aids and Additional Suggestions

The last section of multiple-choice questions focused on possible graphical/visual learning aids. 59% students reported having thought about using graphical diagrams, and as many as 88% believe that a graphical representation could be effective to improve their understanding of concurrent programming. However, fewer students have a clear idea about which type of tool would be best suited to deal with thread-related concepts. After all, the most effective tools that may perhaps be used are not widely known in school contexts: in particular, several students did not know about Petri nets (74%), Wait-for/Holt graphs (51%), or finite-state automata (50%). Thus, the tools considered most useful for understanding were flow-charts (51% of positive ratings) and block diagrams (43%), even though they are not the best suited to the purpose.

The final open answer was answered by 45 students and 12 of them suggested to introduce examples of increasing difficulty more gradually (e.g. *"exercises and examples of more gradual difficulty"*). Other recurrent proposals include the use of concrete, real-world examples (6 – e.g. *"examples and exercises drawn from the real world [...]"*); the use of graphical/animation aids (6 – e.g. *"make more use of graphical representations [...]"*); a deeper theoretical discussion.

5 Discussion

First of all, we discuss the research questions raised in the introduction.

Q1 – To what extent are students at ease with some basic concepts of concurrent programming? All findings summarized in the previous section—answers to subjective perception questions, actual performance in the proposed tasks and self-confidence in the provided solutions—consistently indicate that concurrent programming is a rather challenging subject for high school students. This is not surprising, in that it corroborates what other educators have observed, by analysing both learners' performance (more often at the tertiary instruction level, e.g. [5,23]) as well as their subjective perception. In the latter respect, in particular, Brođanac and colleagues [2] report that concurrent programming is perceived by the high school students involved in their investigation as more difficult than several other programming topics.

Q2 – To what extent does students' perception of self-confidence correlate with their actual performance in simple concurrent programming tasks? To the best of our knowledge, this kind of analysis does not appear in previous studies specifically addressed to concurrency. The statistics listed in Table 1 indicate that, overall, students' self-confidence in the provided solutions tends to only roughly correlate to their actual performance in the tasks at hand—and, more in general, they express a low self-confidence level in relation to the considered subject. The discrepancy between self-confidence and performance is especially marked relative to the first task, where higher levels of self-confidence align with wrong answers for subtasks 1c and 1d—so suggesting some lack of awareness about their difficulties, even for a simple problem.

Q3 – What are their major difficulties when learning concurrent programming? As pointed out in Sect. 4, dealing with *synchronization* and with thread state transitions (via *wait/notify* operations) represent major challenges. Such difficulties emerge both from students' perception and from their performance in synchronization tasks. Once again, this confirms previous results that synchronization mechanisms are common sources of students' mistakes, see e.g. [5,7]. In connection with synchronization tasks, it may also be worth observing that students' performance in subtasks 1c and 1d is significantly worse than that in task 2 (see Fig. 3), although the program in the latter case is far more complex. A conceivable explanation of a similar phenomenon is that envisaged by Kolikant [14] and mentioned in Sect. 2: a successful solution for task 2 may be achieved by analogy with a stereotypical producer-consumer pattern, without being concerned with the details of the underlying mechanisms.

Limitations. The present study has been conceived with an exploratory character, in order to gather preliminary insight into a range of aspects in connection with the learning of concurrent programming in the context of our school system. Of course, each such aspect would be worth a specific, more focused investigation, possibly involving larger student samples in a wider geographic area.

Implications for Educators. Mastery of basic concepts of concurrent programming is a cognitively demanding endeavor that, in order to nurture meaningful learning, requires much pedagogical effort and time spent on the subject.

As also pointed out by multiple students, teachers should be particularly careful to choose an appropriate set of examples of gradually increasing complexity. An additional issue worth being considered is the use of graphical/visual aids, especially to support the integration of spatial abilities into the learning process [4,24]. A range of existing tools has been reviewed, e.g., by Libert & Vanhoof [15]. Finally, the low correlation between self-confidence and performance suggests that more attention needs to be paid to students' metacognitive skills [18], possibly by offering them "opportunities for empirical validation of their knowledge" and explicit instruction in this respect [19, p. 148].

6 Conclusions

In this paper we have presented the results of an exploratory investigation, carried out via a survey, addressing high school students' perception, proficiency and self-confidence when dealing with concurrent programming tasks. While the main implications of our findings are discussed in the previous section, appropriate decisions about the potential role of this subject in a high school context, and specifically in our school system, are bound to find a reasonable trade-off between two opposite poles: on the one hand, the relevance of the topic from a professional perspectives [9]; on the other, the high cognitive challenge [2] in light of learners' maturity and teaching time available to develop the subject.

Future Perspectives. Besides designing and planning more focused investigations to overcome the limitations mentioned above, a shorter-term goal could be to administer the current version of the survey in other technical high schools following heterogeneous approaches to the teaching of concurrent programming, with the aim of assessing the extent to which different instructional approaches influence students' perceptions and/or achievements. Further research could be devised in order to evaluate and compare the effectiveness of different graphical/visual tools to improve the understanding of thread-related concepts.

Appendix: Survey Questions

An easier-to-read version of the survey questions is also available online at the link:
http://nid.dimi.uniud.it/additional_material/thread_survey/thread_survey.pdf

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:

```
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 (Fig. 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 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.

Task 1.b Analyze the execution of the following code snippets (Fig. 5) 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.

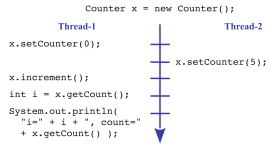

Fig. 4. Task 1.a.

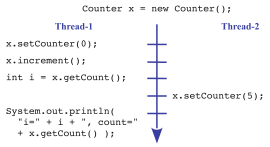

Fig. 5. Task 1.b

Task 1.c Analyze the execution of the following code snippets (Fig. 6) 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.

Task 1.d Analyze the execution of the following code snippets (Fig. 7) 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.

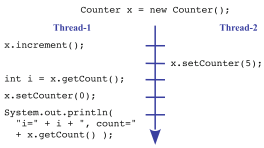

Fig. 6. Task 1.c

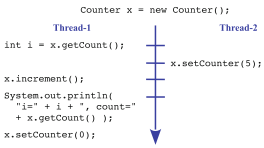

Fig. 7. Task 1.d

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 (Fig. 8) 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; PPPP PP375; The program hangs in a deadlock; The result cannot be predicted because there are several possibilities.

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 (Fig. 9) can help to avoid conflicts in the management of the resource itself?

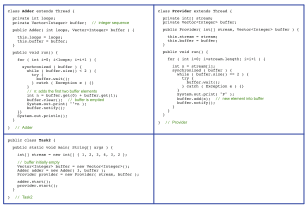

Fig. 8. Task 2

Fig. 9. 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.
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.: 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.

```
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
```

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.–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? –grade Likert scale (1 = Never – 4 = Often)
- Do you think a graphical representation of how threads work could be effective in improving your understanding? –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) : I don't know this type of representation, not very useful, partially useful, quite useful, very helpful.: 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)

References

1. Ben-Ari, M., Kolikant, Y.B.D.: Thinking parallel: the process of learning concurrency. SIGCSE Bull. **31**(3), 13–16 (1999)
2. Brođanac, P., Novak, J., Boljat, I.: Has the time come to teach parallel programming to secondary school students? Heliyon **8**(1), e08662 (2022)
3. Bruce, K.B., Danyluk, A., Murtagh, T.: Introducing concurrency in CS1. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE 2010, pp. 224–228. ACM, New York, NY, USA (2010)
4. Buckley, J., Seery, N., Canty, D.: Spatial cognition in engineering education: developing a spatial ability framework to support the translation of theory into practice. Eur. J. Eng. Educ. **44**(1–2), 164–178 (2019)
5. Choi, S.E., Lewis, E.C.: A study of common pitfalls in simple multi-threaded programs. In: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, SIGCSE, pp. 325–329. ACM, New York, NY, USA (2000)
6. Conte, D.J., de Souza, P.S.L., Martins, G., Bruschi, S.M.: Teaching parallel programming for beginners in computer science. In: 2020 IEEE Frontiers in Education Conference (FIE), pp. 1–9 (2020)
7. Fekete, A.D.: Teaching students to develop thread-safe java classes. In: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008, pp. 119–123. ACM, New York, NY, USA (2008)
8. Fernández, A., Fernández, C., Miguel-Dávila, J.A., Conde, M.A., Matellán, V.: Supercomputers to improve the performance in higher education: a review of the literature. Comput. Educ. **128**, 353–364 (2019)
9. Gardner, W.B.: Should we be teaching parallel programming? In: Proceedings of the 22nd Western Canadian Conference on Computing Education, WCCCE 2017, ACM, New York, NY, USA (2017)
10. Hartley, S.J.: Alfonse, wait here for my signal!. In: Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1999, pp. 58–62. ACM, New York, NY, USA (1999)

11. John, D.J., Thomas, S.J.: Parallel and distributed computing across the computer science curriculum. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 1085–1090 (2014)
12. Joint Task Force on Computing Curricula: Association for Computing Machinery (ACM) and IEEE Computer Society: Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. Association for Computing Machinery, New York, NY, USA (2013)
13. Ko, Y., Burgstaller, B., Scholz, B.: Parallel from the beginning: the case for multicore programming in the computer science undergraduate curriculum. In: Proceedings of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, pp. 415–420. ACM, New York, NY, USA (2013)
14. Kolikant, Y.B.D.: Learning concurrency: evolution of students' understanding of synchronization. Int. J. Hum.-Comput. Stud. **60**(2), 243–268 (2004)
15. Libert, C., Vanhoof, W.: Survey of software visualization systems to teach message-passing concurrency in secondary school. In: Bajo, J., et al. (eds.) PAAMS 2017, vol. 722, pp. 386–397. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-60285-1_33
16. Lönnberg, J., Berglund, A., Malmi, L.: How students develop concurrent programs. In: Proceedings of the 11th Australasian Conference on Computing Education, ACE 2009, vol. 95. pp. 129–138. Australian Computer Society, Inc. (2009)
17. Luxton-Reilly, A., Simon et al.: Introductory programming: a systematic literature review. In: Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2018, pp. 55–106. ACM, New York, NY, USA (2018)
18. Mani, M., Mazumder, Q.: Incorporating metacognition into learning. In: Proceedings of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE 2013, pp. 53–58. ACM, New York, USA (2013)
19. Murphy, L., Tenenberg, J.: Do computer science students know what they know? A calibration study of data structure knowledge. In: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2005, pp. 148–152. ACM, New York, USA (2005)
20. Rifkin, A.: Teaching parallel programming and software engineering concepts to high school students. In: Proceedings of the 25th SIGCSE Symposium on Computer Science Education, SIGCSE 1994, pp. 26–30. ACM, New York, NY, USA (1994)
21. Rivoire, S.: A breadth-first course in multicore and manycore programming. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE 2010, pp. 214–218. ACM, New York, NY, USA (2010)
22. Scapin, E., Mirolo, C.: An exploration of teachers' perspective about the learning of iteration-control constructs. In: Pozdniakov, S.N., Dagienė, V. (eds.) ISSEP 2019. LNCS, vol. 11913, pp. 15–27. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33759-9_2
23. Torbert, S., Vishkin, U., Tzur, R., Ellison, D.J.: Is teaching parallel algorithmic thinking to high school students possible? One teacher's experience. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education, SIGCSE 2010, pp. 290–294. ACM, New York, NY, USA (2010)

24. Wai, J., Lubinski, D., Benbow, C.P.: Spatial ability for STEM domains: aligning over 50 years of cumulative psychological knowledge solidifies its importance. J. Educ. Psychol. **101**(4), 817–835 (2009)
25. Zhu, J., Alderfer, K., Smith, B., Char, B., Ontañón, S.: Understanding learners' problem-solving strategies in concurrent and parallel programming: A game-based approach (2020). arXiv.org (ARXIV2005.04789)