A Student's View of Concurrency: A Study of Common Mistakes in Introductory Courses on Concurrency

Filip Strömbäck, Linda Mannila, Mikael Asplund and Mariam Kamkar

A Student's View of Concurrency – A Study of Common Mistakes in Introductory Courses on Concurrency

Filip Strömbäck
Department of Computer and Information Science
Linköping University
Linköping, Sweden
filip.stromback@liu.se

Linda Mannila
Department of Computer and Information Science
Linköping University
Linköping, Sweden
linda.mannila@liu.se

Mikael Asplund
Department of Computer and Information Science
Linköping University
Linköping, Sweden
mikael.asplund@liu.se

Mariam Kamkar
Department of Computer and Information Science
Linköping University
Linköping, Sweden
mariam.kamkar@liu.se

ABSTRACT

This paper investigates common misconceptions held by students regarding concurrency in order to better understand how concurrency education can be improved in the future. As a part of the exam in two courses on concurrency and operating systems, students were asked to identify and eliminate any concurrency issues in a piece of code as a part of their final exam. Different types of mistakes were identified and the 216 answers were sorted into categories accordingly. The results presented in this paper show that while most students were able to identify the cause of an issue given its symptoms, only approximately half manage to successfully eliminate the concurrency issues. Many of the incorrect solutions fail to associate shared data with a synchronization primitive, e.g. using one lock to protect multiple instances of a data structure, or multiple locks to protect the same instance in different situations. This suggests that students may not only have trouble dealing with concepts related to concurrency, but also more fundamental concepts related to the underlying computational model. Finally, this paper proposes possible explanations for the students' mistakes in terms of improper mental models, and suggests types of problems that highlight the issues with these mental models to improve students' understanding of the subject.

CCS CONCEPTS

• **Applied computing** → **Education**; • **General and reference** → *Empirical studies*; • **Theory of computation** → *Concurrency*.

KEYWORDS

computer science education, concurrency, synchronization, mental models

ACM Reference Format:
Filip Strömbäck, Linda Mannila, Mikael Asplund, and Mariam Kamkar. 2019. A Student's View of Concurrency – A Study of Common Mistakes in Introductory Courses on Concurrency. In *International Computing Education Research Conference (ICER'19), August 12–14, 2019, Toronto, Ontario, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3291279.3339415

1 INTRODUCTION

In recent years, performance improvements in computers are mostly due to an increase in the number of cores rather than to performance improvements in a single core. Thus, as most platforms contain multiple cores, the ability to construct concurrent programs is becoming an essential skill in order to utilize the increasing computational power provided by the hardware [23]. As the importance of concurrency increases, high quality education of the subject is essential. Concurrency is usually perceived as a difficult subject by students, and as such many visualizations, languages and other tools have been proposed to aid education. However, Moström [24] points out that the educational impact of many of the proposed tools have not been properly evaluated, and that little is known in general on how to teach concurrency well. Moström does, however, highlight the work by Kolikant [15, 16, 17] and Lönnberg [19], who investigate common mistakes made by students and misconceptions that could cause the mistakes. This knowledge is relevant to education in the context of constructivism, as these mistakes represent instances where students use a mental model not suitable for the task, i.e. a *non-viable* mental model. Being aware of these non-viable mental models is useful to construct problems or examples that illustrate the shortcomings of these models, which may in turn help students to realize the problems with the mental models, and allow them to revise the model or abandon it in favor of a better model.

In this paper, we aim to continue the exploration of non-viable mental models by examining students' mistakes when working with concurrency as suggested by Lönnberg. Data was collected by examining the answers to a question that appeared on the final exam in two courses on *concurrency and operating systems* given at Linköping University in order to collect a large amount of data, resulting in 216 answers. In addition to the large data set used, we extend the scope of previous work by Kolikant by examining issues that arise when allowing multiple instances of shared data structures, and relate the issues found not only to non-viable mental models of concepts taught in concurrency, but also in the underlying computational model.

As such, this paper aims to answer the following questions:

- What kind of synchronization mistakes are common in an introductory course in concurrency, and how common are they?
- What non-viable mental models could cause these mistakes?
- What kind of problems or examples could be used to highlight these non-viable mental models?

In this paper, we first introduce concurrency, constructivism and related research in the field of object oriented programming in Section 2, followed by related work in concurrency in Section 3. In Section 4 the data collection and the exam exercise is described, followed by the results in Section 5. The implications of the results are discussed in Section 6, followed by a conclusion in Section 7.

2 BACKGROUND

In this section, we briefly introduce the aspects of concurrency relevant to this paper followed by a short survey of constructivism in computer science education, specifically aimed at the importance of mental models.

2.1 Concurrency

Most modern operating system allow programs to create multiple *threads* to represent concurrent execution of different parts of the program. Each thread represents a single stream of instructions that may be executed in parallel. In general, few assumptions can be made about the relative execution speed of different threads or the ordering of read and write operations between multiple threads [2].

In order to allow multiple threads to communicate safely, the courses examined in this paper introduce *locks* to achieve *mutual exclusion* as well as *semaphores* and *condition variables* for tasks where locks are not sufficient. Another benefit of using synchronization primitives emphasized in the courses is that they avoid *busy-wait* by suspending waiting threads when necessary.

2.2 Constructivism and Mental Models

Constructivism claims that knowledge is constructed by the learner based on previously acquired knowledge rather than being transferred from a teacher to a learner [4]. This implies that each learner constructs a mental model of their own while learning a subject. For this reason, Ben-Ari [3] suggests teaching a suitable mental model explicitly in order to help students construct a viable mental model and thereby avoid common misconceptions. Alternatively, as suggested by Hadjerrouit [11], constructivism can be used to guide the didactic approaches used to guide the students' creation of relevant and viable mental models. Furthermore, Boyer et al. [7] note that failure to adhere to these principles, e.g. by focusing on concepts and definitions first rather than connecting the new concepts to students' previous knowledge, leads to programming being perceived as "using and adapting others' programs" [7] which in turn encourages a shallow understanding and a trial-and-error approach that is not suitable when working with concurrency.

Since the constructed mental model plays an important role in constructivism, much work has been done to understand which mental models are used by students, and whether they are viable or not [1, 7, 11, 14, 22]. For example, Ma et al. [22] examined the mental models of references and assignments in Java held by novice programmers using a questionnaire and found that 54 out of 65 students held at least one inappropriate mental model, called a *non-viable* mental model. Similar results were found in a similar study was conducted regarding conditionals, loop, scope and parameter passing [21]. The importance of mental models was also highlighted by Reges [27], who noted that students' success on a particular relatively simple question was strongly correlated with success on the entire exam. The author argues that this particular question was particularly good at indicating whether the student used a viable mental model or not, as it required emulating the behavior of the computer, and suggests that this kind of questions could be useful to highlight non-viable models used by students during a course.

2.3 Object Oriented Programming

Even though it is not necessary to know object oriented programming (OOP) when working with concurrency, mental models of OOP concepts and their shortcomings have been explored extensively [11, 14, 26]. For example, one of the most important concepts in OOP, the distinction between a *class* and an *object* [10] are often confused or seen as the same by students in introductory courses [12, 25, 26, 29, 30]. Additionally, Paul and Vahrenhold [25] noted difficulties in understanding object identity and the difference between values and references. These concepts are highly relevant to concurrency, as non-viable mental models regarding these concept leads to difficulties in identifying shared data, which could result in incorrect synchronization of concurrent programs.

3 RELATED WORK

As noted by Moström [24], much work has been done in the area of concurrency education. A large number of visualizations, languages, and other tools, e.g. Linda [8] and Multi-Logo [28], have been created to aid education. However, Moström points out that much of the work regarding tools in the area focuses on the tools themselves rather than than the impact they have on education, leading to the conclusion that not much is known about concurrency education.

One area pointed out as a promising part towards improved concurrency education is studying programming errors made by students, which is done by Lönnberg and Kolikant among others. Lönnberg [19] analyzes students' solutions to three concurrency assignments in a course and concludes that a large part of errors in the tasks can be attributed to misunderstanding either some part of the task itself, synchronization goals of the task or the computational model in use (i.e. using a non-viable mental model). Lönnberg et al. [20] continue to investigate the reason behind the misunderstandings through a phenomenographic study (i.e. a study exploring different ways in which respondents understand or experience a given phenomenon [6]) regarding one of the tasks in the course, and find that some students experience the task as a requirement to get a grade or as an idealized problem that will not work in practice, which could explain at least some of the misunderstandings. The study also covers how students believe concurrent programs should be tested, where the authors find categories ranging from unplanned testing to partial correctness proofs.

Similar ideas are also explored by Kolikant [18], who examined high school students' views of correctness using questionnaires. Kolikant found that students' views of correctness often differ from teacher's views, which explains why students find some solutions satisfactory while the teacher does not. In another study, Kolikant explores another cause of errors found by Lönnberg: misunderstanding the computation model. Kolikant [16] observed students' mistakes during laboratory sessions and interviewed one of the students. From this, the author notes that some students initially experience the concurrent system as a user, without trying to understand what happens inside the system, and that students need to change their perspective to properly understand the models and solve problems. A change in their perspective was also observed by Ben-Ari and Kolikant [5] in a study of an introductory course on concurrency for high school students. In the course, some students pointed out that in contrast to sequential programming, which could be solved by trial and error, concurrency requires a more formal approach and a deeper understanding to properly solve problems. This, along with further observations from the same study [17], highlights another necessary change in their perspective similar to the previous one.

As previously mentioned, this paper continues on the path suggested by Moström [24] and investigates common concurrency issues by examining mistakes made by students when writing concurrent programs. Several aspects of this area has been explored in similar ways by Kolikant. When studying students' ability to identify and correct synchronization errors in a small program using a simulator, most students found the two synchronization goals (one order of execution- and one mutual exclusion problem), and most students successfully solved the problem [16, 17]. In another study Kolikant examines the high-level approach taken by students when solving problems involving concurrency [15]. Compared to the work by Kolikant, this paper analyzes a slightly more complex task that involves multiple instances of a data structure, which allows analyzing misconceptions regarding a larger part of the computational model in addition to concepts purely related to concurrency. Additionally, this paper uses a larger data set, which gives a better indication of the frequency of the types of the encountered mistakes.

4 METHOD

Data was collected from two similar courses on the subject of *concurrency and operating systems* given in the second year of two computer science programs at Linköping University. Both courses introduce students to concurrency and synchronization and let the students explore the concepts in computer assignments where they implement functionality in the educational operating system Pintos[1]. During the assignments, students implement various system calls (for example, `exec`, `wait` and `exit` which are used to synchronize processes), and synchronize an existing file system implementation. The main difference between the two courses is that one course (course A) assumes that students are already familiar with operating system concepts on a theoretical level (i.e. they are assumed to know about virtual memory, file systems, etc.) and only introduces concurrency in addition to the practical assignments. The other course (course B), on the other hand, does not assume prior knowledge and introduces all relevant concepts, including concurrency and the practical assignments. Course A is conducted in Swedish for students attending the second year of a 3 year CS program while course B is given in English for students attending the second year of a 5 year CS program and master students.

Both courses are concluded with a final exam where students demonstrate, among other things, their ability to find and eliminate potential concurrency issues in a given piece of code using the synchronization primitives introduced during the course. One such question is presented in Section 4.1 and analyzed in this paper as described in Section 4.2. The exams in the two courses differ slightly: in course A students prepare and submit their answers on a computer while students in course B works on paper. This allows students in course A to edit the code more easily and even compile and execute their solutions if they desire. Even though this is an option, neither course requires submitted solutions to be syntactically correct. The ability to test the solution before it is submitted does not necessarily provide a large advantage for students in course A, since it is difficult, if not impossible, to test whether a solution is correct or not.

Since the question was initially designed for course A, where students have no issue editing the code, the code was altered slightly to better suit the written format used in course B. These changes are noted in Section 4.1.

4.1 Exam Question

The exam question analyzed in this paper is a typical exam question for the courses; the student is given a piece of unsynchronized code along with a description of the expected behavior. The student is then asked to identify any problems in the code, and finally to resolve the issues using suitable synchronization primitives. The question analyzed in this paper is presented below:

> As a teacher, you are constantly on the hunt for good ideas for exam exercises. The main problem, however, is that it is easy to forget the good ideas before they are actually used to produce a good question. To solve this problem, one teacher implemented a data structure to keep track of them. The implementation of the data structure is shown in Listing 1 on page 4. It has the following operations:
>
> - `idea_init`: Initializes the idea buffer.
> - `idea_add`: Adds an idea (a string) to the buffer. If the buffer is full and the idea could not be added `false` should be returned, otherwise `true` should be returned.
> - `idea_get`: Randomly selects and returns an idea from the buffer. The idea is also removed to ensure it is not used for another exam. If no ideas are present, `idea_get` shall wait until a new idea is added with `idea_add`.
>
> During the exam periods, `idea_add` and `idea_get` are used frequently by many teachers. Therefore, it is important that they are usable from multiple threads simultaneously as far as possible.
>
> (a) Is *busy-wait* used somewhere in the implementation? If so, where?
>
> (b) Use suitable synchronization primitives to eliminate any occurrences of *busy-wait* you found.

[1] https://web.stanford.edu/class/cs140/projects/pintos/pintos.html

```
 1      #define BUFFER_SIZE 32
 2
 3      struct idea_buffer {
 4          // All ideas in the buffer. Empty elements are
 5          // set to NULL.
 6          const char *ideas[BUFFER_SIZE];
 7          // Number of ideas in the buffer.
 8          int count;
 9      };
10      // Initialize the buffer.
11      void idea_init(struct idea_buffer *buffer) {
12          for (int i = 0; i < BUFFER_SIZE; i++)
13              buffer->ideas[i] = NULL;
14          buffer->count = 0;
15      }
16      // Add a new idea to an empty location in the
17      // buffer. Returns 'false' if the buffer is full.
18      bool idea_add(struct idea_buffer *buffer,
19                    const char *idea) {
20          // Find an empty location.
21          int found = BUFFER_SIZE;
22          for (int i = 0; i < BUFFER_SIZE; i++) {
23    A{        if (buffer->ideas[i] == NULL) {
24                  found = i;
25                  break;
26              }
27          }
28          // Full?
29          if (found >= BUFFER_SIZE)
30              return false;
31          // Insert into the buffer.
32    B{    buffer->ideas[found] = idea;
33    B{    buffer->count++;
34          return true;
35      }
36      // Get and remove a random element from the
37      // buffer. If no elements are present, the
38      // function waits for an element to be added.
39      const char *idea_get(struct idea_buffer *buffer) {
40    C{    while (buffer->count == 0)
41    C{        ;
42          buffer->count--;
43          // Find an element. Start from a random index,
44          // and look through the array until we find a
45          // non-empty element.
46          int pos = rand() % BUFFER_SIZE;
47   D{E{   while (buffer->ideas[pos] == NULL) {
48   D{         pos = (pos + 1) % BUFFER_SIZE;
49   D{     }
50          // Remove it.
51    F{    const char *result = buffer->ideas[pos];
52    F{    buffer->ideas[pos] = NULL;
53          return result;
54      }
```

Listing 1: Code for the exam exercise. The marked regions were not present in the original question.

(c) After using the data structure for a while, some users notice that the same idea has been used multiple times (i.e. multiple calls to `idea_get` returned the same idea). Furthermore, ideas sometimes disappear from the buffer, even though `idea_add` indicated success by returning `true`. Explain with an example what could have happened when...
  1: ...the same idea was used multiple times.
  2: ...the buffer "lost" one or more ideas.

(d) Mark any critical sections present in the functions `idea_add` and `idea_get`. Also note the resource(s) that need protection.

(e) Use suitable synchronization primitives to synchronize the code based on the critical sections you found.
Note: Strive from a solution that allows maximum theoretical concurrency, even though that solution might perform worse in practice due to synchronization overheads (please note if you think this is the case).
Note: Points may be deducted for excessive locking.

In this question, students are first asked to identify (a) and correct (b) any instances of busy-wait. In this case, `get` uses busy-wait to wait for the buffer to contain at least one element in loop C. Since the variable `count` is not used otherwise, it could be removed entirely and replaced with a semaphore to solve the problem.

After the issues with busy-wait, the student is presented with two issues and is asked to present a possible cause for them (c), followed by identifying critical sections in the code (d) and protecting them as appropriate (e). Part (d) is used to encourage students to show the data associated with each critical section, which sometimes shows misconceptions that would lead to otherwise correct synchronization. Furthermore, as mentioned in the exercise text, students are encouraged to strive for a solution that maximizes the theoretical available concurrency, even though it might not be the best solution in practice. This is also to encourage students to minimize their critical sections, which also helps highlighting misconceptions, but most importantly discourages solutions that synchronizes the two functions in their entirety with a single lock, which is mostly uninteresting both for the exam and for this paper.

As previously mentioned, the code in Listing 1 was altered slightly for the exam in course B in order to limit the amount of refactoring required, which is impractical on a written exam. First and foremost, the `count` variable was removed entirely, making loop D a busy-wait loop instead of loop C. Aside from that, lines 31-34 were moved inside the loop, replacing the `break` on line 25, and line 29 was removed. These changes eliminate some confusion about the `count` variable, and makes a solution that uses one lock for each element more practical to implement on paper without a large impact on the aspects studied in this paper, as presented in Section 4.2.

4.2 Data Analysis

The answers to the question presented in Section 4.1 were analyzed using a method inspired by *content analysis* [13]. While initially correcting the exams for course A, the first author noted the issues present in the students' solutions and compiled a list of aspects to be recorded for each answer. Then, answers from both courses were analyzed by the first author according to the aspects found previously. For each answer, it was recorded whether or not it was correct, and the types of any mistakes. A number of additional aspects were also recorded for each part, regardless of the correctness of the answer. The criteria for correctness and the recorded aspects are as follows:

(a) A correct answer highlights loop C as a busy-wait loop and nothing else (loop D for course B). Any other locations mentioned were recorded.
(b) A correct answer successfully removes the previously found busy-wait, possibly by removing the `count` variable. The synchronization primitive used, whether it was declared inside the struct or at global scope, as well as any mistakes in initialization or usage were recorded.
(c) A correct answer illustrates a scenario that causes the problem in the question. If the answer was incorrect, the kind of issue was recorded as well.
(d) This part was not analyzed in isolation; the critical section and the variables protected were used for better categorization in (e).
(e) A correct answer eliminates data races in the code. Forgetting to release a lock in the failure case of `add` was not deemed an error (i.e. missing the early return on line 30 for course A). Furthermore, errors in synchronizing the `count` variable were ignored as the variable is missing entirely from course B, and was expected to be removed from some solutions in course A. In addition to correctness, the following aspects were recorded:
    - The synchronization granularity of the solution (e.g. global or local locks, one lock for each element).
    - Any synchronization errors remaining.
    - Any variables protected by the critical section in (d) that do not need protection.

After recording the aspects mentioned above for each answer, the answers were categorized according to the recorded data. Each category represents answers that solve the problem in a similar way and contain the same kind of mistakes. The categories found, and the number of answers in each category are presented in Section 5.

5 RESULTS

In total, 216 students answered at least one part of the question, 67 in course A and 149 in course B. The results from the analysis described in Section 4.2 are presented below. The two courses are presented separately to highlight any differences between them. All percentages are relative to the total number of students in each course.

5.1 Identifying Busy-Wait (Part A)

As can be seen in Table 1, all students that answered (a) realized that the `get` function contains busy-wait. However, as indicated by *mentioned get*, a number of students in course B pointed to the entire `get` function rather than the loop in particular. Furthermore, as shown by the category *additional loop*, a number of students also pointed to additional loops in the program. In course A, this was mainly due to additionally identifying loop D, while the 3 students in course B mistakenly identified the loops in the `add` function as well. Note that since the code was simplified for course B, the `get` function only contained one loop.

Category	Course A		Course B		Total	
Correct	56	84%	126	85%	182	84%
Mentioned `get`	0	0%	16	11%	16	7%
Additional loop	11	16%	3	2%	14	6%
No answer	0	0%	4	3%	4	2%
Total	67	100%	149	100%	216	100%

Table 1: Results for (a).

5.2 Eliminating Busy-Wait (Part B)

Category	Course A		Course B		Total	
Semaphore	59	88%	134	90%	193	89%
instance	46	69%	83	56%	129	60%
global	12	18%	13	9%	25	12%
unspecified	1	1%	38	26%	39	18%
Condition	6	9%	-	-	6	3%
No answer	2	3%	15	10%	17	8%
Total	67	100%	149	100%	216	100%

Table 2: Type and location of the primitive used in (b).

Table 2 shows which primitive was used to eliminate busy-wait, and if it was created at *global* scope or inside the `idea_buffer` struct, using one for each *instance*. Using a global primitive was surprisingly common, even though it will not work properly if multiple instances of the buffer are used simultaneously. These solutions may still be deemed correct in Table 3. Due to the structure of the exam in course B, a large number of students did not specify the location of the semaphore, which is represented by the *unspecified* category. Condition variables were only used in course A as they were disallowed in course B. All students who used condition variables declared it inside the struct, and provided mostly correct solutions except for failing to protect the *signal* operation with a lock as required by the implementation used in the courses.

Category	Course A		Course B		Total	
Correct	43	64%	91	61%	134	62%
Incorrect usage	10	19%	23	15%	36	17%
Incorrect initialization	3	4%	4	3%	7	3%
Used as a lock	3	4%	16	11%	19	9%
Total	59	88%	134	90%	193	89%

Table 3: Types of solutions using semaphores in (b).

Table 3 shows the mistakes present when eliminating the busy-wait using a semaphore. Even though most students were able to identify the busy-wait, far fewer successfully used a semaphore to properly eliminate it. The most common mistake was *incorrect usage*, i.e. not calling *up* and *down* at appropriate times or reversing the *up* and *down* operations. A common example from course A is shown in Listing 2. Here, *down* is only called when the buffer is empty. This solution will not behave correctly when paired with an implementation of add that always calls *up* on success as it will sometimes fail to wait for elements to be inserted into the buffer. The implementation is however correct if paired with an implementation that only calls *up* when an element is inserted into an empty buffer provided that the count variable is properly synchronized, which was not the case. In course B, it was common to call *down* inside the loop, as shown in Listing 3. This solution is also incorrect, as it calls *down* for every empty slot it finds, and is therefore likely to wait even if there are elements in the buffer.

```
const char *idea_get(struct idea_buffer *buffer) {
    if (--buffer->count == 0)
        sema_down(&buffer->sema);
    // ...
}
```

Listing 2: Incorrect semaphore usage in course A

```
const char *idea_get(struct idea_buffer *buffer) {
    int pos = rand() % BUFFER_SIZE;
    while (buffer->ideas[pos] == NULL) {
        sema_down(&buffer->sema);
        pos = (pos + 1) % BUFFER_SIZE;
    }
    // ...
}
```

Listing 3: Incorrect semaphore usage in course B

The related category *incorrect initialization* represents solutions where the only issue was failing to initialize the semaphore to an appropriate value. Finally, a number of students attempted to *use the semaphore as a lock* or use a regular lock to eliminate the busy-wait, which removes data races but does not eliminate busy-wait.

5.3 Illustrate Synchronization Issues (Part C)

Category	Course A		Course B		Total	
Correct	58	87%	121	81%	179	83%
Shared locals	4	6%	2	1%	6	3%
Vague	2	3%	10	7%	12	6%
Other	3	4%	6	4%	9	4%
No answer	0	0%	10	7%	10	5%
Total	67	100%	149	100%	216	100%

Table 4: Results for the first issue in (c).

Tables 4 and 5 show that the majority of students managed to give an example illustrating the issues outlined in (c). A number of incorrect examples are represented by the *shared locals* categories, where the student provided an example where one or more local variables were altered when multiple threads executed the same function, erroneously suggesting that local variables are shared between threads. The *vague* categories represent students whose description was not detailed enough to determine whether or not it was correct. Finally, the category *misunderstood semantics* represents students who provide an example of a situation that is acceptable according to the semantics of the data structure, for example where an element added by add was removed by get before add returns true. The *other* categories represent answers that describe other concurrency issues than the ones outlined in (c), or attributes the issues to other supposed defects in the implementation, such as a buffer overflow causing strings to be overwritten.

Category	Course A		Course B		Total	
Correct	53	79%	122	82%	175	81%
Misunderstood semantics	2	3%	4	3%	6	3%
Shared locals	2	3%	2	1%	4	2%
Vague	4	6%	6	4%	10	5%
Other	5	7%	6	4%	11	5%
No answer	1	1%	9	6%	10	5%
Total	67	100%	149	100%	216	100%

Table 5: Results for the second issue in (c).

5.4 Synchronize the Code (Parts D and E)

As previously mentioned, (d) and (e) were analyzed together. Answers to (d) were used to highlight possible misconceptions in the data protected by the critical sections for correct solutions.

Granularity	Course A		Course B		Total	
One per element	11	16%	15	10%	26	12%
One per instance	40	60%	69	46%	109	50%
Global	13	19%	19	13%	32	15%
One per call	0	0%	3	2%	3	1%
Unclear/inconsistent	3	4%	34	23%	37	17%
No answer	0	0%	9	6%	9	4%
Total	67	100%	149	100%	216	100%

Table 6: Synchronization granularity in (e).

Table 6 shows the synchronization granularity of the solutions to (e). The best solution according to the exercise is using *one lock per element*, as it allows maximum concurrency. Even though this was the best solution, only a relatively small amount of students attempted this solution. Most students who attempted this solution did, however, succeed in providing a correct solution (as indicated by the rows *one lock per element* in Table 7). The most common solution, and most probably the most efficient in practice, is to use *one lock per instance* of the data structure, allowing different instances to operate independently of each other while execution is serialized within each instance. A surprisingly large number of students also opted to use *global* locks, which is correct but less efficient than the other solutions. Another surprising solution was to declare the locks as local variables inside get and add, effectively using *one lock per call*, which essentially makes the locks useless. The final category, *unclear/inconsistent* covers answers that do not indicate where locks are declared, or where the declaration and the usage is inconsistent.

Table 7 shows the types of mistakes present in the answers to (e). Note that a single answer may contain multiple mistakes, and may therefore appear in multiple categories. The *correct* category, as well as subcategories are, however, mutually exclusive. Even though most students provided a correct answer to (c), only 45%

Category	Course A		Course B		Total	
Correct	37	55%	59	40%	96	44%
one lock per element	10	15%	12	8%	22	10%
shared locals	4	6%	4	3%	8	4%
Separate add and get	13	19%	46	31%	59	27%
one lock per element	0	0%	3	2%	3	1%
motivation	0	0%	9	6%	9	4%
Incorrect CS in get	21	31%	32	21%	53	25%
too small	15	22%	31	21%	46	21%
split	6	9%	1	1%	7	3%
Incorrect CS in add	15	22%	12	8%	27	12%
too small	9	13%	12	8%	21	10%
split	6	9%	0	0%	6	3%
Shared locals	2	3%	5	3%	7	3%
Local locks	0	0%	3	2%	3	1%
No answer	0	0%	9	6%	9	4%

Table 7: Results for (d) and (e).

of the students provided a solution that correctly eliminates data races. However, 8% of those indicated that at least one local variable needs to be protected by the critical section in (d), as shown by the subcategory *shared locals*. Additionally, 3% of all students indicated shared locals while making other mistakes, as shown by the separate category *shared locals*, meaning that a total of 7% of all students mistakenly believed that local variables need protection.

The most common mistake was to synchronize add and get with a separate set of locks, allowing concurrent operations on the shared buffer contents. Students in the *motivation* subcategory as well as all students who used *one lock per element* motivated why this is safe. However, the motivation incorrectly assumes that reads and writes are not reordered by the compiler, and that concurrent reads and writes to a variable are safe, which is not true in general. Another common issue was incorrect critical sections in the two functions. The most common mistake here was a too small critical section in the get function, only including the lines marked F. Additionally, students split the critical section into two separate critical sections, marked with E and F. Similarly, a common issue was to only consider the region marked B as critical in the add function rather than also including the comparison marked A. The critical section was also commonly split around the if statement in course A, as marked by A and B. However, since the code altered for course B, the issue with a split critical section in add was not present there.

6 DISCUSSION

The results in Section 5 show that students from the two courses performed similarly, suggesting that the differences outlined in Section 4 did not have a major impact on the results. There were, however, some notable differences between the courses, which are highlighted in Section 6.1. Additionally, Table 3 shows that 63% of students successfully used a semaphore to eliminate busy-wait, which is similar to a similar question studied by Kolikant [17] where 51% of the students correctly solved a similar problem with a semaphore, suggesting that the observations from this paper are relevant in other contexts as well. Finally, in Section 6.2, we suggest possible misconceptions that could be the cause of the mistakes found.

6.1 Differences Between the Courses

As previously mentioned, students from the two courses performed similarly with some notable differences. The most notable difference is that a larger portion of students in course B did not note where the used synchronization primitive was declared, which is reflected by the *unspecified* category in Table 2 and *unclear* in Table 6. This is probably due to the fact that course A used computer exams, which encouraged students to modify the code rather than describing their changes in text, which in turn made it easier to determine where the primitive was declared. Additionally, Table 7 shows that a larger portion of students in course A (55%) provided correct answers compared to course B (40%), and that the category *separate add and get* was more common in course B while the categories representing incorrect critical sections were more common in course A than in course B. The slight difference in correct solutions could be attributed to the fact that course A gives students more time to experiment with the concepts compared to course B, since it only covers concurrency and synchronization while course B also covers other aspects of operating systems. Another possible cause of these differences is the removal of of the count variable in course B, which could affect the approach selected by students when attempting to understand the program.

6.2 Mistakes and Non-Viable Mental Models

In Table 1 we can see that students generally manage to identify instances of busy-wait, even though some students also incorrectly identified other loops in the programs as busy-wait. These mistakes could be caused by not properly understanding the intent of the code, or over-reliance on pattern matching, which is common in concurrency [17], as well as in programming in general [7]. Furthermore, Table 3 shows that even though most students correctly identified the busy-wait in (a), only 65% of all students managed to provide a correct solution. This differs from what Kolikant [16, 17] observed when studying students' solutions to similar problems, namely that once the synchronization goals are identified, applying the correct synchronization primitive is straightforward. This difference could be explained by considering that a correct answer to (a) does not necessarily imply that the synchronization goals are understood, which is likely the case for the answers in the *used as a lock* category. The remaining categories, *incorrect usage* and *incorrect initialization*, however, represent solutions where a semaphore was used, but used incorrectly in some way. These categories likely point to a non-viable mental model of a semaphore being used rather than missing synchronization goals. For example, the semaphore may be incorrectly considered to count remaining resources rather than available resources, which causes incorrect initialization and reversing *up* and *down*. Furthermore, failing to realize that the semaphore counts some resource, perhaps limiting the counter to 0 and 1, likely cause solutions attempting to only call *down* in some cases.

The results for (c) in Tables 4 and 5 are similar to the results for (a): a majority of students (83% and 81%) managed to present an explanation for the issues in the question, but only 44% provided a suitable solution, once again contrary to the observation by Kolikant [16, 17]. In this case, however, the categories *incorrect CS in get* and *incorrect CS in add* clearly illustrate that a correct answer to (c) does not imply understanding the extent of the critical sections in the code. A large portion of students only synchronized the lines directly surrounding the issue identified in (c), failing to realize other similar problems. Another non-viable mental model is shown clearly by *shared locals* in Tables 4 and 5 and Table 7 is a model where local variables in functions are shared between multiple threads. This could be due to an incorrect model of the scope of variables, which was highlighted as difficult by Goldman et al. [10], either in general or only regarding concurrency.

The misconception that local variables are shared by different threads is likely caused by a failure to distinguish between different *instances* of a function call, not realizing that each instance has its own set of local variables, which is similar to the difficulties differentiating between a *class* and an *instance* noted by Sanders and Thomas [29]. Eckerdal and Thuné [9] suggests that this could be caused by students experiencing the program in terms of the program *text* rather than its dynamic execution. In this model, it makes sense for local variables to be shared since there is only one instance of each variable, since they only appear once in the program text.

A related issue is shown in Tables 3 and 6. A fair number of students used a global semaphore (12%) or a global lock (15%) to synchronize the `idea_buffer`. As previously mentioned, using a global semaphore in (b) will not work properly if multiple instances of the buffer is used concurrently, since the semaphore will count the number of elements added globally rather than in one particular instance. In (e), however, using a global lock only disallows multiple instances to be used concurrently, but does not introduce any synchronization issues. This could once again be caused by students failing to realize that multiple instances may be created from a single definition, which was observed by Sanders and Thomas [29] who noted that students create copies of the same class with different names instead of creating multiple instances. Using this non-viable mental model, there is no difference between global and local scope, and as such the location of the primitive does not matter. Another non-viable mental model that explains this issue is a model where locks and semaphores protect the *code* rather than a particular *resource*, or where the CPU is treated as a global resource. In both of these cases, the focus is mainly on the code rather than on the data, meaning that global locks makes sense. This mental model could also justify the category *local locks* in Table 7, since declaring locks locally makes sense if locks are protecting the code rather than the data, or if local variables are shared between threads.

The misconception that locks protect the code or some other global resource could also explain the category *separate add and get*, where the two functions were protected by a different set of locks. Additionally, if an non-viable model of scope and references is used, highlighted as a difficult subject by Goldman et al. [10], Ragonis and Ben-Ari [26], the fact that these functions operate on the same data might not be realized. However, as shown by the subcategory *motivation*, some students in course B actively made the decision to improve concurrency of their solution by using separate locks, and argued for the correctness of their solution. The argument, while correct and showing a high proficiency in concurrency, was made using an incorrect model where reads and writes to shared variables are assumed to occur in the same order as in the program text and where write operations are atomic. Sadly, this is not necessarily true, as languages or the hardware generally provide few guarantees on the ordering of memory accesses and the atomicity of reads and writes [2]. As we can see, using an idealized computational model when working with concurrency is sometimes dangerous, and not using the proper computational model could be a common cause of these mistakes, as noted by Lönnberg [19], even though we are unable to conclude to what extent the proposed mental models cause the observed issues from the data in this paper.

7 CONCLUSION

The results presented in Section 5 show that the majority of students were capable of correctly identifying *busy-wait* and specifying a cause for the specified concurrency issues. However, far fewer were able to correctly eliminate the identified busy-wait (65%) and properly synchronize the code using locks (44%). Most mistakes (15%) in eliminating the busy-wait originate from mistakes in using the semaphore. When synchronizing code using locks, many students (27%) used separate locks for the two different functions, even though they may operate on the same data. This could be due to a mental model focusing on the *code* rather than the *data* being used, or using a simplified computational model. This misconception could also explain the large amount of students using global instances of synchronization primitives to synchronize the code (12% for semaphores and 15% for locks). Finally, many students fail to see the full extent of critical sections (25% for `get` and 12% for `add`), and only synchronize the code identified earlier as problematic even though other issues exist.

These results suggest that an increased focus on mental models is beneficial to improve concurrency education. It is important to be aware of the mistakes outlined above not only in order to teach viable mental models, but also to illustrate cases where common non-viable mental models are inappropriate. This can be highlighted by illustrating the following cases:

- Comparing code synchronized with too short, or multiple separate critical sections with correctly synchronized code.
- Using multiple instances of a data structure to illustrate that they are independent, and thus should be synchronized separately.
- Accessing the same data from multiple functions in multiple threads concurrently to emphasize that the *data* rather than the *code* needs synchronization.
- Calling the same function from multiple threads to illustrate that local variables are not shared between threads.
- Illustrate the limitations of simplified computational models by using compiler optimizations to break code that would be safe in the simplified model.

To summarize, students are generally able to identify some concurrency issues, at least when knowing what to look for. Correctly solving the problem, however, is not as common, as all aspects of the problem need to be identified. Finally, the suboptimal usage of locks suggests that students use a non-viable mental model of either the underlying model of scope and references, or a simplified model of concurrency that provides too strong guarantees.

REFERENCES

[1] Dan Aharoni. 2000. Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures. *SIGCSE Bull.* 32, 1 (March 2000), 26–30. https://doi.org/10.1145/331795.331804

[2] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11).* ACM, 55–66. https://doi.org/10.1145/1926385.1926394

[3] Mordechai Ben-Ari. 1998. Constructivism in Computer Science Education. *SIGCSE Bull.* 30, 1 (March 1998), 257–261. https://doi.org/10.1145/274790.274308

[4] Mordechai Ben-Ari. 2002. From Theory to Experiment to Practice in CS Education. In *2nd Annual Finnish/Baltic Sea Conference on Computer Science Education.*

[5] Mordechai Ben-Ari and Yifat Ben-David Kolikant. 1999. Thinking Parallel: The Process of Learning Concurrency. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '99).* ACM, 13–16. https://doi.org/10.1145/305786.305831

[6] Anders Berglund. 2006. Phenomenography as a way to research learning in computing. *Bulletin of Applied Computing and Information Technology* 4, 1 (2006). http://www.citrenz.ac.nz/bacit/0401/2006Berglund_Phenomenography.htm

[7] Naomi R. Boyer, Sara Langevin, and Alessio Gaspar. 2008. Self Direction & Constructivism in Programming Education. In *Proceedings of the 9th ACM SIGITE Conference on Information Technology Education (SIGITE '08).* ACM, 89–94. https://doi.org/10.1145/1414558.1414585

[8] Nicholas Carriero and David Gelernter. 1989. Linda in Context. *Commun. ACM* 32, 4 (April 1989), 444–458. https://doi.org/10.1145/63334.63337

[9] Anna Eckerdal and Michael Thuné. 2005. Novice Java Programmers' Conceptions of "Object" and "Class", and Variation Theory. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05).* ACM, 89–93. https://doi.org/10.1145/1067445.1067473

[10] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2008. Identifying Important and Difficult Concepts in Introductory Computing Courses Using a Delphi Process. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08).* ACM, 256–260. https://doi.org/10.1145/1352135.1352226

[11] Said Hadjerrouit. 1999. A Constructivist Approach to Object-oriented Design and Programming. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '99).* ACM, 171–174. https://doi.org/10.1145/305786.305910

[12] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding Object Misconceptions. In *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '97).* ACM, 131–134. https://doi.org/10.1145/268084.268132

[13] Hsiu-Fang Hsieh and Sarah E. Shannon. 2005. Three Approaches to Qualitative Content Analysis. *Qualitative Health Research* 15, 9 (2005), 1277–1288. https://doi.org/10.1177/1049732305276687

[14] Peter Hubwieser and Andreas Mühling. 2011. What Students (Should) Know About Object Oriented Programming. In *Proceedings of the Seventh International Workshop on Computing Education Research (ICER '11).* ACM, 77–84. https://doi.org/10.1145/2016911.2016929

[15] Yifat Ben-David Kolikant. 2001. Gardeners and Cinema Tickets: High School Students' Preconceptions of Concurrency. *Computer Science Education* 11, 3 (2001), 221–245.

[16] Yifat Ben-David Kolikant. 2004. Learning Concurrency as an Entry Point to the Community of Computer Science Practitioners. *Journal of Computers in Mathematics and Science Teaching* 23, 1 (2004), 21–46.

[17] Yifat Ben-David Kolikant. 2004. Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies* 60, 2 (2004), 243–268. https://doi.org/10.1016/j.ijhcs.2003.10.005

[18] Yifat Ben-David Kolikant. 2005. Students' Alternative Standards for Correctness. In *Proceedings of the First International Workshop on Computing Education Research (ICER '05).* ACM, 37–43. https://doi.org/10.1145/1089786.1089790

[19] Jan Lönnberg. 2006. Student Errors in Concurrent Programming Assignments. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006 (Baltic Sea '06).* ACM, 145–146. https://doi.org/10.1145/1315803.1315833

[20] Jan Lönnberg, Anders Berglund, and Lauri Malmi. 2009. How Students Develop Concurrent Programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95 (ACE '09).* Australian Computer Society, Inc., 129–138. http://dl.acm.org/citation.cfm?id=1862712.1862732

[21] Linxiao Ma, John Ferguson, Marc Roper, Isla Ross, and Murray Wood. 2009. Improving the Mental Models Held by Novice Programmers Using Cognitive Conflict and Jeliot Visualisations. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09).* ACM, 166–170. https://doi.org/10.1145/1562877.1562931

[22] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the Viability of Mental Models Held by Novice Programmers. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07).* ACM, 499–503. https://doi.org/10.1145/1227310.1227481

[23] Tim Mattson and Michael Wrinn. 2008. Parallel Programming: Can We PLEASE Get It Right This Time?. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08).* ACM, 7–11. https://doi.org/10.1145/1391469.1391474

[24] Jan Erik Moström. 2011. Learning concurrency - What's the problem? In *A study of student problems in learning to program.* Umeå, Sweden, Chapter VII. http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-48216

[25] Wolfgang Paul and Jan Vahrenhold. 2013. Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13).* ACM, 29–34. https://doi.org/10.1145/2445196.2445212

[26] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (2005), 203–221. https://doi.org/10.1080/08993400500224310

[27] Stuart Reges. 2008. The Mystery of "B := (B = False)". In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08).* ACM, 21–25. https://doi.org/10.1145/1352135.1352147

[28] Mitchel Resnick. 1990. MultiLogo: A Study of Children and Concurrent Programming. *Interactive Learning Environments* 1, 3 (1990), 153–170. https://doi.org/10.1080/104948290010301

[29] Kate Sanders and Lynda Thomas. 2007. Checklists for Grading Object-oriented CS1 Programs: Concepts and Misconceptions. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '07).* ACM, 166–170. https://doi.org/10.1145/1268784.1268834

[30] Ewan Tempero, Paul Denny, Andrew Luxton-Reilly, and Paul Ralph. 2018. Objects Count So Count Objects!. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18).* ACM, 187–195. https://doi.org/10.1145/3230977.3230985