The Non-Deterministic Path to Concurrency – Exploring how Students Understand the Abstractions of Concurrency

Filip STRÖMBÄCK, Linda MANNILA, Mariam KAMKAR
Linköping University, Sweden
e-mail: filip.stromback@liu.se, linda.mannila@liu.se, mariam.kamkar@liu.se

Abstract. Concurrency is often perceived as difficult by students. One reason for this may be due to the fact that abstractions used in concurrent programs leave more situations undefined compared to sequential programs (e.g., in what order statements are executed), which makes it harder to create a proper mental model of the execution environment. Students who aim to explore the abstractions through testing are further hindered by the non-determinism of concurrent programs since even incorrect programs may seem to work properly most of the time. In this paper we aim to explore how students' understanding these abstractions by examining 137 solutions to two concurrency questions given on the final exam in two years of an introductory concurrency course. To highlight problematic areas of these abstractions, we present alternative abstractions under which each incorrect solution would be correct.

Keywords: abstraction, concurrency, memory model, synchronization, locks.

1. Introduction

One of the main differences between sequential programs and concurrent programs is the environment in which they are executed. While sequential programs are executed in a deterministic environment, concurrent programs are executed in a non-deterministic one and may thus behave differently each time they are executed, even on the same machine. This non-determinism means that it is difficult to learn about the underlying abstractions by running a program and observing its behavior, as is largely possible with sequential programs. Thus, a student who has misunderstood some part of an abstraction is not likely to find this out through trial and error, which Lönnberg *et al.* (2009) found to be how some students develop concurrent programs.

Non-determinism is often a difficult concept for students (Alexandron *et al.*, 2016). This is especially true when working with concurrent programs, as it is necessary to

consider a large set of possible interleavings between threads when reasoning about the program, which students often struggle with (Xie *et al.*, 2007). Furthermore, Lawson and Kraemer (2020) found that students used various *sleep*-functions to avoid concurrency issues in Java. In order to make it feasible to reason about concurrent programs and to avoid the belief that the issues can be avoided by slowing down the program using sleep functions, it is necessary to use a suitable abstraction of the concurrent execution environment. In addition, one also needs to be aware of how synchronization primitives (which are often implemented as abstract data types) affect the execution of the concurrent program. All of these abstractions do, however, look very different from the ones used in sequential programs due to the non-determinism of the environment. In particular, an abstraction of a concurrent environment typically leaves a large portion of its behavior undefined compared to sequential ones. For example, it is often possible to find a single order in which statements are executed in a sequential program (assuming the input is known). The same is not true for concurrent programs since statements executed by different threads may be executed in any order relative to each other, and may even appear to have been executed out of order. This example illustrates a situation where it is important to pay close attention to what is left unspecified by the abstraction of the concurrent execution environment, as it is otherwise easy to assume that the abstraction provides stronger guarantees than it actually does.

These observations may at least partially be a reason for why students struggle with concurrent programming: students need to learn new abstractions, both of the execution environment, and the abstractions used to affect the execution environment (i.e., abstract data types, such as locks, semaphores, etc.). While doing this, students need to pay close attention to what properties that could previously be taken for granted are no longer true due to the non-determinism of the system. Finally, students are not able to verify their understanding of the new abstractions through testing. As noted by Strömbäck *et al.* (2019), the difficulties in testing solutions also means that any weaknesses in prerequisite skills, such as pointers or references, may also affect a student's ability to reason about concurrent programs without the student realizing it.

In this paper, we continue the work by Strömbäck *et al.* (2019), Lawson and Kraemer (2020), and others, who studied common errors in student solutions to a concurrency question. In order to better understand the underlying reasons as to why a student makes a particular error, we focus on what part of the underlying abstractions students might have misunderstood rather than the errors themselves. These insights can then be used to better highlight the problematic areas in concurrency education, perhaps through examples or visualizations, in order to better aid students who learn concurrency in the future. In particular, we aim to answer the following research questions:

RQ1 What incorrect assumptions do students make regarding abstractions used in concurrent programming in their first course on concurrency and operating systems?

RQ2 To what extent do these students understand the concurrency constraints imposed by common abstractions in the C language?

RQ3 How well do these students understand the concurrency constraints imposed by simple data structures implemented in the C language?

The remainder of this paper is structured as follows: In Section 2 we introduce the relevant abstractions used in concurrency. In Section 3 we present related work on teaching concurrency. In Section 4, we introduce the method used in this paper and the two questions that were examined. We then present the results in Section 5, and discuss the method and the results in Section 6. Finally, we provide a conclusion in Section 7.

2. Abstractions of a Concurrent Environment

In this section we provide a brief introduction to concurrency, with a particular focus on the different abstractions that are used to describe concurrent systems and the abstract data types that are used to synchronize concurrent programs (synchronization primitives).

2.1. *Concurrency in the Shared Memory Model*

In this paper, we only consider the shared memory model. In this model a concurrent program consists of one or more *threads* (Silberschatz *et al.*, 2010, ch. 3–5). Each thread represents a sequential stream of instructions that may be executed concurrently with other threads. All threads belonging to the same *process* share memory and are thus able to communicate with each other.

As with sequential programming, it is not feasible to consider all details of the execution environment when writing concurrent programs. Therefore, programmers use an abstraction of the concurrent execution environment when writing concurrent programs. This abstraction defines a computational model that holds for a large set of implementations, regardless of how concurrency is achieved there (e.g., preemption or hardware parallelism). An abstraction of a concurrent execution environment essentially consists of two parts: a description of when threads are executed, and the semantics of the shared memory, which is typically called a *memory model*.

Typically, few guarantees are provided regarding when threads are executed. Often it is only safe to assume that each thread will get the chance to be executed eventually. In particular, it is not safe to assume that threads are executed at (approximately) the same speed or even at the same time, even if this is may be the case in practice for some systems.

The memory model used in the C language, which we use in this paper, is usually considered a weak memory model as it leaves many situations undefined. Essentially, the C (and C++) memory model assumes that no data is shared between threads unless access to it is explicitly protected in some manner (Batty *et al.*, 2011). This assumption gives the compiler a large amount of freedom to perform optimizations, but might cause surprising results for students who accidentally assume that a stronger memory model, such as *sequential consistency* or *total store ordering*, is used. The sequential consistency (Lamport, 1979) memory model guarantees that all threads observe that reads and writes happen in program order. As this model closely resembles how sequential

programs work, it is a natural first step to incorrectly assume that sequential consistency holds in concurrent systems. Total store ordering (Sewell *et al.*, 2010) is weaker than sequential consistency, but still stronger than the C memory model. In this model, writes occur in program order, but they may not be visible to other threads immediately. There are, of course, many other memory models that a programmer might need to take into account in low-level programming (Alglave *et al.*, 2010), but as they are not relevant to this paper, we will not cover them here.

2.2. *Synchronization Primitives*

One way of protecting shared data in a concurrent program is to use *synchronization primitives*, which are often implemented as abstract data types. These abstract data types provide a convenient and platform-independent abstraction of the low-level details provided by the current platform and operating system to control threads in a concurrent program.

The questions analyzed in this paper involve *locks*, *semaphores* and *condition variables*. Locks are exclusively used to ensure that two or more threads do not access the same data concurrently by ensuring *mutual exclusion* for the *critical sections* in the program. Before entering a critical section, a thread needs to *acquire* the lock, and then *release* the lock after leaving the critical section. Semaphores and condition variables can be used to ensure mutual exclusion as well, but they also allow waiting for arbitrary events (e.g., a task being completed) without repeatedly checking if a condition is true (i.e., *busy-wait*). A semaphore can be described as a counter which can be incremented and decremented. If a thread attempts to decrement the counter below zero, it is put to sleep until another thread increments the counter. Similarly, a condition variable can be described as a queue of threads that are waiting for some condition to become true. Other threads may then wake the waiting threads as necessary.

3. Related Work

In this section we first present related work on the importance of fundamental programming skills when learning more advanced topics, such as concurrency. We then present existing research in teaching and learning concurrency, a selection of tools that aim to help students to learn concurrency, and finally recent efforts in structuring the curriculum to better teach concurrency in a CS program.

3.1. *The Impact of Fundamentals*

Even though the focus of this paper is abstractions related to concurrency, it is not possible to entirely ignore abstractions typically taught in earlier courses. As shown by Nelson *et al.* (2020) many assessments, such as the ones used in this paper, rely on the

student being familiar with the abstractions and concepts taught in earlier courses as well as the new ones. Furthermore, Valstar *et al.* (2019) found a correlation between students' performance on their CS1 and CS2 courses. These observations suggest that at least some problems students face are due to a lacking understanding of fundamentals.

Difficult concepts in introductory courses have been studied extensively. For example, Ma *et al.* (2007) showed that only 17% of students held a viable mental model of reference assignments at the end of a Java programming course. Difficulties with pointers and references have been identified as difficult by others as well, for example by Goldman *et al.* (2008) using a Delphi process, and by Valstar *et al.* (2019) as still being problematic in a CS2 course. A good understanding of pointers and references in particular are important when working with concurrency, for example to identify shared data, as pointed out by Strömbäck *et al.* (2019).

3.2. *Learning Concurrency*

Much research has been done on how students learn concurrency. One notable line of inquiry explored by Kolikant is to explore mistakes made by students, both by examining incorrect answers and interviews (Kolikant, 2004). The results indicate that the difficult part is to correctly identify the synchronization goals, after which solving the problem is relatively easy. Kolikant (2001) also argues that problems with a strong connection to computing (e.g., a distributed system for selling tickets) is better than problems based on events in the real word (e.g., coordinating gardeners) since the limitations imposed by the computerized setting better reflect what is actually possible in concurrent programs, especially for distributed systems. The first problem from this work, which is a text description of multiple sellers selling tickets to a concert, has been further studied in other contexts. One such example is Lewandowski *et al.* (2007), who used the problem to study what students know about concurrency before having formal instruction and found that students see the problems, even though they are not always able to find suitable solutions. Another example is Lawson *et al.* (2019), who studied what types of solutions are suggested by undergraduate students in four different years of their education. The authors found that students at all levels were able to identify the issues and found a number of different approaches to solve the problem. Kolikant (2005) has also shown that at least some part of incorrect solutions to concurrency problems are due to students having an alternative definition of correctness, for example believing that it is enough for concurrent programs to produce the correct answer *most* of the time. Others have also examined students' solutions to find and quantify errors. One such example is Strömbäck *et al.* (2019), who examined 216 solutions and quantified the types of errors present. Another example is Lawson and Kraemer (2020) who examined 24 students' solutions to a concurrency problem along with students' reflections. Each student solved the problem both in the shared memory model used in this paper, and in the *actor model*. The authors found a number of issues, perhaps most notably that students often use *sleep*-functions to avoid concurrency issues. Choi and Lewis (2000) also suggested using a tool to automatically find concurrency issues

in student solutions. Using this method, the authors found a number of solutions containing race conditions, even in simple concurrent programs. Finally, Lönnberg *et al.* (2008) suggests using these common errors to guide the feedback given to students to better prepare students for future similar problems.

A partially separate, but yet related, line of inquiry is how students experience the subject of concurrency and the various abstractions therein. This has largely been done using Phenomenography, a method to find different aspects of a subject of learning through interviews (Marton, 2014). Using Phenomenography, Lönnberg *et al.* has explored how students perceive an abstraction called *tuple spaces*, which can be used to synchronize concurrent programs (Lönnberg and Berglund, 2007). They have also studied how students perceive correctness, developing and debugging concurrent programs (Lönnberg *et al.*, 2009). Strömbäck *et al.* (2020) has also studied how students perceive concurrency itself as well as critical sections.

Another important line of research is the design and evaluation of notations and methods to help students reason about and design concurrent programs. One early such example is Xie *et al.* (2007). The authors noted that instructors often use ad-hoc sketches to describe concurrent programs and that the large set of possible interleavings are difficult for students to reason about. To help students, they developed an extension of UML sequence diagrams to help formalize the ad-hoc sketches, and to help students reason about the concurrent programs. Another approach was explored by Bijlsma *et al.* (2017), who suggested a 5-step design method for concurrent programs with the aim of scaffolding the often difficult design process of concurrent programs. The authors later evaluated the design process (Bijlsma *et al.*, 2019), and found that while the method was useful, *micro-steps* should be available on demand to further aid scaffolding the difficult stages. The authors also pointed out the importance of clear instructions as to what parts of a program are expected to be executed in parallel, and highlighted the benefits and dangers of using real-world analogies for objects in the program.

This paper takes an approach that lies between the first two lines of inquiry. We use student solutions to concurrency problems as a data source. We do, however, not use the data to find out which errors are common, but rather to find what abstractions in concurrency are problematic when certain errors occur. This aspect of our approach is thus more in line with the second line of inquiry, as it is more qualitative in nature.

3.3. *Tools to Aid Students*

Due to the difficulty of teaching and learning concurrency, a number of different tools has been proposed to aid students. One category of such tools propose an alternative concurrency model where communication between threads is more explicit than in the shared memory model. One example of this is to use tuple spaces to highlight the communication between different threads (Lin and Tatar, 2011). Another approach is to use a language like Salsa (Desell, 2013) which uses the *actor model*, where threads are only able to communicate by exchanging messages.

There are also a number of tools that aim to visualize some aspects of concurrency. There are a large number of visualizations of classical concurrency problems, such as the *dining philosophers* problem (Adams *et al.*, 2019). Another example is a tool called The Deadlock Empire[1] which provides the user with two or more pieces of source code and asks the user to interleave execution of the pieces (by single-stepping a piece) in a way that exposes a concurrency issue. While this is an excellent introduction to concurrency issues, it does not allow users to modify the code or test their own code. It also does not include references or pointers, which is important in larger programs. A similar tool, called ConEE, was proposed by Offenwanger and Lucet (2014). This tool does allow loading arbitrary programs and automatically proving whether or not the solution is correct. Again, it does not address the issues introduced by pointers or references. Another approach is explored by Eludicate (Exton, 2000), which provides execution traces that users may use to reason about the program. Lönnberg (2012) has also examined a number of tools. One example is a train simulation to introduce semaphores, and another is a tool called Atropos (Lönnberg *et al.*, 2011b), which visualizes the program flow as a graph and allows executing it both forwards and in reverse. Lönnberg *et al.* (2011a) also examined how students utilize Atropos when debugging concurrent programs. Finally, Alexandron *et al.* (2016) suggested using a visual language called *live sequence charts* to introduce students to nondeterminism, partly to help understanding nondeterministic state machines, and partly to help understanding of concurrency, as the authors argue that nondeterminism is an important part of both topics.

3.4. *Integrating Concurrency into the CS Curriculum*

Alongside the efforts to help students learn concurrency in various ways, much work has been done to better integrate concurrency into the CS curriculum. Among others, Ernst and Stevenson (2008) argue that since multicore systems become increasingly common, it is vital to educate CS students in how to properly utilize the available concurrency. Furthermore, they suggest integrating concurrency in CS1, CS2 and algorithms courses, and suggest a number of suitable projects at these levels. Concurrency was later formally included in the Computer Science Curricula 2013 (ACM/IEEE Joint Task Force, 2013), which motivated the CDER Book Project to provide detailed guidelines for integrating the ACM 2013 Curriculum Guidelines into programs, and to provide suitable material both for students and instructors (Prasad *et al.*, 2015, 2018). Recently, a working group also explored further extending the curriculum to also include high performance computing, which also covers distributed systems (Raj *et al.*, 2020).

Alongside this work, others have made various attempts at integrating concurrency into their curriculum and evaluated their approaches. For example, Burtscher *et al.* (2015) explores how concurrency can be introduced into an already full curriculum using an early-and-often approach, similarly to what Ernst and Stevenson (2008) suggested earlier. Qasem *et al.* (2021) also explored an early-and-often approach, and put further

---

[1] https://deadlockempire.github.io/

emphasis on introducing concepts at a proper level of abstraction at the different stages of the education. Others, such as Bogaerts (2017), have examined how concurrency can be introduced in a CS1 course and evaluated the approach across multiple years. The author concluded that it is more effective to cover a small set of topics in depth rather than a larger set of topics shallowly.

4. Method

In order to explore how students understand the concurrency abstractions and how they interact with other abstractions, we examine students' solutions to synchronization problems that appeared on the final exam in a course on concurrency and operating systems. In particular, we examine any incorrect solutions and suggest one or more alternative abstractions of the execution environment that would be needed to make the solution correct, as this gives an insight into the incorrect abstractions that students might believe to be true. In cases where this is not possible, we suggest different semantics of the data structure described by the question which would make the solution correct.

4.1. *Data Collection*

Data was collected from the final exam of a course in concurrency and operating systems given at Linköping University. This course is given towards the end of the second year for students pursuing a bachelor's degree in computer science. The course only introduces concurrency in a shared memory context (see Section 2) since it assumes that students are already familiar with the fundamentals of operating systems from an earlier course. As the course does not expect any prior experience with concurrency, the goal is to teach students why concurrent programs need synchronization and how to apply synchronization correctly rather than how to utilize concurrency to maximize the performance of a program. Students practice these skills along with their previously gained knowledge on operating systems in a series of lab assignments using the educational operating system Pintos.[2] The assignments involve implementing a number of system calls (e.g., `read`, `write`, `exec` and `wait`) and ensure that they behave correctly in the concurrent environment of an operating system kernel. In particular, students need to ensure that the `wait` system call behaves correctly (typically by using semaphores), and to make sure that file system operations behave atomically by synchronizing the existing file system implementation.

The final exam of the course also focuses on concurrency and thus involves one or more exercises where students are given a piece of code that is executed concurrently. Students are then asked to identify and eliminate any concurrency issues in the code. This process is scaffolded by providing one or two scenarios that highlight some symptoms of the concurrency issues present and asking students to find the cause of these

---

[2] `http://www.scs.stanford.edu/07au-cs140/pintos/pintos.html`

issues. Most students successfully find a cause for the highlighted issues, and we will therefore not focus on this part of the question in this paper. After this, students are asked to highlight the critical sections present in the code, and to eliminate concurrency issues using synchronization primitives. Students are also asked to maximize the theoretically possible concurrency in their solution (i.e., to minimize critical sections, and allow unrelated operations to execute concurrently as far as possible). This pushes students away from simply acquiring a lock in the beginning of each function and releasing it in the end. Rather, students have to closely consider what data is shared and where the critical sections are. As such, this reveals many misunderstandings of the underlying abstractions. The exam is given as a computer exam, where students are able to edit the code in a text editor and are able to compile and test their solutions using standard UNIX tools. Whether or not the solutions compile or not is, however, not a part of the grading.

Data was collected from the final exam of two subsequent years (year 1 and year 2). Two different questions were used for the two years as the exam questions are published after each exam. As such, it is not possible to directly compare the students' results from the two years. This is, however, not a problem for this paper since our goal is to find a large number of incorrect abstractions used by students. Thus, having two different questions increases the likelihood that more of these incorrect abstractions become visible, both by increasing the sample size and through the different nature of the questions.

4.2. *Data Analysis*

All solutions from the two final exams were collected and then analyzed using a method similar to the one used by Strömbäck *et al.* (2019): We examined each solution individually, determined whether or not the solution was correct and the granularity of the synchronization. For each incorrect solution, we also recorded a short description of the issue. After examining all solutions individually, we categorized all incorrect solutions based on the description so that similar errors end up in the same category. As some solutions contained multiple errors, a single solution may appear in multiple categories.

In order to answer the research questions of this paper we need to find out which incorrect abstractions are used by students. As such, we are interested in finding the answer to the question "What abstractions did this student use when arriving at this solution?" This question is, however, difficult to answer without interviewing each student, ideally during a think-aloud. If we assume that students reason logically within their current (but maybe incorrect or incomplete) understanding of the abstractions provided by the system, which Marton (2014, ch. 4) argues is typically the case, we can approximate the answer to the above question with a different question: "Which abstractions need to be altered in order to make this solution correct?" This is a question we are able to answer only from the submitted solutions by finding alternative abstractions that would make the solution correct. Since this is an approximation it is, however, necessary to treat the results accordingly. Since each incorrect solution may have multiple sets of alternative abstractions that makes it correct, we can not be sure of which corresponds to the ones actually used by the student. Furthermore, Albrecht and Grabowski (2020) found that

17% of incorrect solutions to programming exercises were due to unintentional mistakes (e.g., syntax errors, missing semicolons). We take this into account by not requiring that the submitted code should compile. It does, however, mean that we have to account for the fact that students make mistakes, especially during the stress many feel during an exam. Both the approximation and the fact that students may make mistakes in their reasoning means that we can not use the results to make quantitative statements about these alternative abstractions. Rather, the results are more similar to a Phenomenographic inquiry (Marton, 2014) which aims to qualitatively describe some aspects of an object of learning. Similarly, our results describe important aspects of the abstractions in a concurrent system by highlighting possible misinterpretations of them. Since the results are qualitative in nature, having a large and diverse sample size increases the likelihood that all relevant aspects are found, rather than allowing us to make statements about which of them are most common.

We use this approximation by analyzing the solutions in each category in turn and for each of them we propose one or more abstractions that need to be altered (e.g., using a stricter memory model) for the solution to be correct. In case multiple combinations of relaxations were needed, we noted all of them. In this paper, we consider modifying the following abstractions:

- Abstractions regarding the memory model.
- Abstractions regarding the implementation of concurrency.
- Abstractions used in the C language and expectations regarding certain constructs.
- The abstractions created in the questions (i.e., altering the semantics in the problem statement).

4.3. *The First Question (Year 1)*

The question that appeared on the exam in year 1 is presented in Fig. 1. It involved synchronizing a data structure, presented in Listing 1, that stores a number of strings in a C array with a fixed size. Insertions are done into the first empty element in the array and are expected to fail if the array is full. Removals pick a random element to remove and are expected to wait until an element is present in the data structure rather than failing. This is the same question as the one used in our previous study (Strömbäck *et al.*, 2019).

As can be seen in Fig. 1, the question aims to assess if students are able to use locks to synchronize access to shared data and use either semaphores or condition variables to wait for the data structure to contain at least one element. The question scaffolds the process of finding and eliminating synchronization issues in (a), which asks where busy-wait occurs, in (c), which asks the student to highlight why some problems occur, and in (d), which asks students to mark critical sections in the code.

A correct solution to this question would be to first find that the loop marked A in Listing 1 act as a busy-wait loop in part (a), and remove that issue by, for example, replacing the loop and the variable `count` with a semaphore in part (b). Next, in part (c),

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 in the file `exam_ideas.c` (see Listing 1). 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.
(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...
    (c1) ...the same idea was used multiple times.
    (c2) ...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 for 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.

Fig. 1. The first question. The code in Listing 1 was provided in a separate file.

the solution provides an example where two threads pick the same position in the array and read the element before removing it on the line marked B (c1), and another example where two threads pick the same empty position before inserting a new element on the line marked C (c2). The solution then identifies two critical sections in the code, marked D and E in Listing 1, (assuming `count` was removed) that needs to be protected for part (d). The critical data is the array `ideas` inside `idea_buffer`, or more precisely, the individual element that the algorithm is currently examining. Finally, in part (e), the solution introduces one or more locks to protect the critical sections. Ideally, the solution utilizes one lock for each element in the array to maximize theoretical concurrency, but a solution that uses one lock for each instance of the `idea_buffer` struct is also acceptable for our purposes.

In the analysis of this question, we focus on parts (d) and (e) since they are the ones that involve identifying critical sections and protecting them with locks, and therefore typically provide most insights into students' understanding of the subject.

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

Listing 1. Code provided alongside the first question in a separate file. The markers were not present in the original code. The code also contained the main-function presented in Appendix A.

4.4. *The Second Question (Year 2)*

The question that appeared on the exam in year 2 is presented in Fig. 2. In this question, two threads try to acquire ingredients from a shared inventory. The code presented in Listing 2 implements the logic to ensure that the correct amount of each ingredient is removed from the inventory, or reports a failure without removing any ingredient. In order to aid readability, only the important parts of the code are presented in Listing 2. The remainder of the implementation, including a simple `main` function, is provided in Appendix B.

As with the first question, this question aims to assess if students are able to use locks to synchronize access to shared data. This question does not, however, require threads to wait and does therefore not cover semaphores and condition variables. Instead, it covers the possibility of deadlocks in (d). Again, this question scaffolds the process of finding and eliminating synchronization issues by asking students to find the cause of an issue in (a), and then asking students to mark critical sections in (b) before asking students to solve the issues in the implementation.

In a small and remote village, the villagers like to spend their Friday nights on the only bar in the village drinking the excellent milk beverages served there. Understandably, nobody is eager to tend to the bar since everyone is tired from a hard week at work. After much consideration, the villagers decided to solve the problem by building two robots to tend the bar.

The robots are controlled by a computer running the program in the given file. To manage both robots at the same time, the program runs a thread for each robot. The program keeps track of the bar's inventory (the data type `ingredient`) and the recipes for the available drinks (the data type `recipe`). When the system is started it initializes the inventory before the threads are started. Thus, we can assume that the initialization is executed in a single thread, and not during normal operation. The threads controlling the robots then call `make_drink` to check whether enough ingredients are available to make a drink that has been ordered, and updates the amount available in the inventory.

 (a) After using the system for a couple of nights a problem surfaces. Sometimes, often near closing time when there are few ingredients remaining, the robots make drinks without having sufficient ingredients available. This is even though there are explicit checks for this particular case in the code!

   Explain with an example what could have happened in this case (i.e., when `make_drink` returns `true` even though there are not enough of both ingredients).

 (b) Mark any critical sections you find in the `make_drink` function. Also note the resource(s) that need protection.

 (c) Use suitable synchronization primitives to solve the issue based on the critical sections you found in (b).

   Note: Strive for 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).

 (d) Is there risk for deadlock in your solution to (c)? Motivate your answer using the four conditions for deadlock.

Fig. 2. The second question. The code in Listing 2 was provided in a separate file.

```
1  // One ingredient available in the bar.
2  struct ingredient {
3      // Name
4      const char *name;
5      // Amount.
6      int amount;
7      // The ingredient's index in the array.
8      int index;
9  };
10 // A recipe. Each recipe consists of exactly two ingredients.
11 struct recipe {
12     // Name.
13     const char *name;
14     // Name and amount required for the first ingredient.
15     const char *ingredient1;
16     int amount1;
17     // Name and amount required for the second ingredient.
18     const char *ingredient2;
19     int amount2;
20 };
21 // Find an ingredient. Returns NULL if none was found.
22 struct ingredient *find_ingredient(const char *name);
23 // Find a recipe. Returns NULL if none was found.
24 struct recipe *find_recipe(const char *name);
25
26 // Make a drink. Return true on success, otherwise false.
27 // This function may be called concurrently from multiple threads
28 // since we have multiple robots in the bar!
29 bool make_drink(const char *name) {
30     struct recipe *recipe = find_recipe(name);
31     // Does the recipe exist?
32     if (recipe == NULL)
33         return false;
34     // Do the ingredients exist?
35     struct ingredient *ing1 = find_ingredient(recipe->ingredient1);
36     struct ingredient *ing2 = find_ingredient(recipe->ingredient2);
37     if (ing1 == NULL || ing2 == NULL)
38         return false;
39     // Note: We assume that ing1 != ing2.
40     // Is there enough of everything?
41     if (ing1->amount < recipe->amount1)
42         return false;
43     if (ing2->amount < recipe->amount2)
44         return false;
45     // Everything seems to be good. We can make the drink!
46     ing2->amount -= recipe->amount2;
47     ing1->amount -= recipe->amount1;
48     return true;
49 }
```

Listing 2. Code provided alongside the second question in a separate file. The highlights were not present in the original code. The implementation of the unlisted functions are provided in Appendix B. The two functions `find_ingredient` and `find_recipe` search for elements in global arrays.

A correct solution to this question identifies that the function `make_drink` may incorrectly return `true` when two threads conclude that there is enough of both ingredients before any of them has modified the amount in stock (lines marked A in Listing 2) as a solution to part (a). For part (b), the solution then identifies either two smaller critical sections marked as B and C in Listing 2 associated with `ing1` and `ing2` respectively, or a single critical section that spans the entirety of B and C associated with both `ing1` and `ing2`. Then, for part (c), the solution introduces one lock for each ingredient, and acquires these locks according to the identified critical sections. In this case it is, however, necessary to be aware of the possibility for a deadlock to occur. There are a number of ways to avoid this problem. For example, one may make sure to always acquire the two locks in the same order (based on the `index`, for example). Another possibility is to perform the two comparisons and subtractions separately to avoid the need to acquire two locks at the same time. Such solutions need to take care to undo any changes to the first ingredient in case the second ingredient turns out not to be available. Finally, the solution argues that deadlocks are not possible in the proposed solution using the four conditions for deadlock in part (d). In case a deadlock is possible, the solution should of course argue for that instead (but full credits would not be given for part (c) in that case).

In this question, we focus on parts (b) and (c). For solutions in which a deadlock may happen we also use part (d) to understand if the students were aware of this issue after it was pointed out to them, or if the student believed that a deadlock was not possible.

5. Results

A total of 137 answers were collected and analyzed, 67 in the first year and 70 in the second year. Out of all solutions, 37 solutions (55%) were deemed correct in the first year, and 39 (56%) in the second year. Fig. 3 provides an overview of the remaining
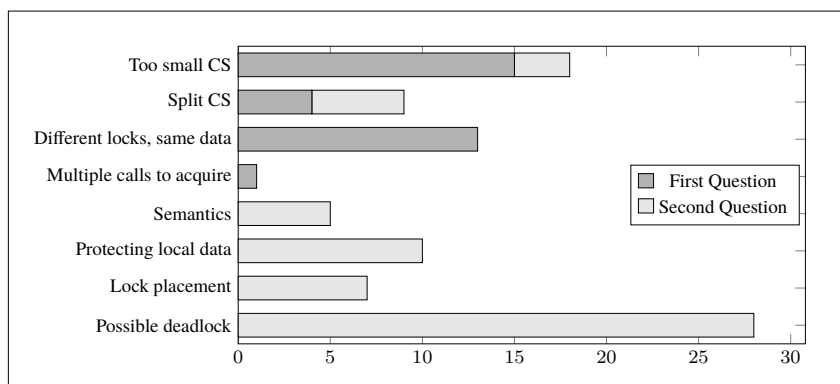

Fig. 3. Overview of the types of errors found in the solutions to the two questions.
Note that only the first two categories were present in both questions.

incorrect solutions. Note that a single solution may contain multiple errors and may therefore appear in more than one category. Unsurprisingly, the different nature of the two questions made different types of errors visible. Therefore, we first present the two categories that describe errors in both questions (Section 5.1), followed by the categories that describe errors in only one of the questions (Sections 5.2 and 5.3). In the remainder of this section we will present these categories in further detail and provide alternative abstractions that make solutions correct. All of these alternative abstractions are written in *italics*, followed by a number in parenthesis that refers to the summary in Section 5.4.

5.1. *Common Errors*

In this section, we present the categories that contain errors from both questions.

5.1.1. *Too Small Critical Section*

This category contains solutions where at least one critical section was smaller than necessary. This is problematic since the behavior of concurrent reads and writes is undefined according to the memory model of the C language. Even in a stricter memory model, any invariants established by the code in the critical section can not be assumed to hold outside of the critical section in general since other threads are free to interfere.

In the first question we found a number of solutions with too small critical sections in both the `idea_add` and the `idea_get` functions, as shown in Listing 3. In the `idea_add` function, the most common error was to only lock the assignment to the array (B), and sometimes also the `if`-statement before it. Another related error was to only lock the `for`-loop before the assignment (A). Both types of solutions would only be correct if *using a lock inside a function prevents the entire function from being executed concurrently (4)*. The first solution (locking only B) is also correct if *a function is only called concurrently with other functions, not with itself (7)*, in this case disallowing concurrent calls to `idea_add` and also assuming *sequential consistency (1)*. Since these solutions often locked the lines where the solution to part (c) indicated that preemption would lead to errors, students likely understood that locks would prevent this issue, but failed to see other possibilities that cause the same issue.

In `idea_get` many solutions only locked around the two lines that read an element from the array and sets it to `NULL` (D). Some solutions instead only locked the `while`-loop that finds a suitable element (C). Once again, these solutions would be correct if *using a lock inside a function prevents the entire function from being executed concurrently (4)*. Additionally, only locking the read and write to the array (D) is correct if *a function is allowed to fail sporadically (9)*, in this case if `idea_get` is allowed to return `NULL` even if the buffer contains an element. Another possibility is if *a function is only called concurrently with other functions, not with itself (7)*, in this case disallowing concurrent calls to `idea_get`. Both of these also require assuming *sequential consistency (1)*.

Similar errors were found in the solutions to the second question, as shown in Listing 4. Some solutions only locked around the subtractions (B), and others only around the comparisons (A). Once again, these solutions are only correct if *using a lock inside a function prevents the entire function from being executed concurrently (4)*. They may also be correct if *sequential consistency (1)* is assumed and the ingredients in stock are allowed to become (a bit) negative, which is essentially a variant of *a function is allowed to fail sporadically (9)*, but applied to invariants. These solutions are similar to those in the first question since they often lock the parts where an issue was found in part (a), but failed to see other potential locations for the same issue.

```
bool idea_add(struct idea_buffer *buffer, const char *idea) {
    int found = BUFFER_SIZE;
    for (int i = 0; i < BUFFER_SIZE; i++) {
        if (buffer->ideas[i] == NULL) {
A           found = i;
            break;
        }
    }
    if (found >= BUFFER_SIZE)
        return false;
B   buffer->ideas[found] = idea;
    buffer->count++;
    return true;
}
const char *idea_get(struct idea_buffer *buffer) {
    // ...
    int pos = rand() % BUFFER_SIZE;
    while (buffer->ideas[pos] == NULL) {
C       pos = (pos + 1) % BUFFER_SIZE;
    }
D   const char *result = buffer->ideas[pos];
    buffer->ideas[pos] = NULL;
    return result;
}
```

Listing 3. Incorrect critical sections found in the first question.

```
void make_drink(const char *name) {
    // ...
    if (ing1->amount < recipe->amount1)
A       return false;
    if (ing2->amount < recipe->amount2)
        return false;
    // Everything seems to be good. We can make the drink!
B   ing2->amount -= recipe->amount2;
    ing1->amount -= recipe->amount1;
    return true;
}
```

Listing 4. Incorrect critical sections found in the second question.

5.1.2. *Split Critical Section*

A related problem to the above-mentioned *too small critical section* is the one of correctly identifying the problematic parts of the functions, but failing to lock them as one consecutive region, essentially splitting the critical section into two pieces. This is incorrect as it allows other threads to modify any invariants that were established in the first part of the critical section but are not re-established in the second critical section.

In the first question, we found solutions that identified both of the parts highlighted in Section 5.1.1 for both `idea_add` and `idea_get`. All of these solutions would once again be correct if *using a lock inside a function prevents the entire function from being executed concurrently (4)*. Another possibility is if *a function is only called concurrently with other functions, not with itself (7)* (either `idea_add`, `idea_get`, or both, depending on which critical sections were split). Since all shared data is protected by locks, these solutions do not require a stronger memory model. In this case, students have likely found a number of locations where preemption would be an issue and have attempted to avoid those issues with locks, but failed to see that their solution has the same issue when preemption occurs between the locked regions.

Another type of solution with split critical sections was visible in solutions to the first question, namely solutions that only locked around writes to shared data. This is essentially an extreme case of the solutions mentioned above. Again, a model where this solution would be correct is one where locks inside a function prevent the functions from executing concurrently. Likely, these particular solutions come from an over-reliance on pattern matching – the students have learned that writes need to be protected with locks and do so without understanding why.

In the second question, we saw solutions similar to the first question, where the check for sufficient ingredients (A) was locked separately (but with the same lock) as the subtractions (B). Once again, this is correct if *using a lock inside a function prevents the entire function from being executed concurrently (4)* or if the ingredients in stock are allowed to become (a bit) negative, which, again, is essentially a variant of *a function is allowed to fail sporadically (9)*, but applied to invariants. This does not require a stronger memory model.

5.2. *Unique to the First Question*

In this section we present categories that were only visible in solutions to the first question. This does not necessarily mean that they are specific to the first problem itself, but rather that these problems did not surface in the second question (e.g., since only one function had to be locked there).

5.2.1. *Different Locks for the Same Data*

A number of solutions used different sets of locks to protect the data in the `ideas` array. Some of these solutions used two locks (either inside the data structure or global), one for `idea_add` and one for `idea_get`. Others used an array of locks (each associated with an element in `ideas`) for `idea_add` and a single lock for `idea_get`. Regard-

less, since these solutions do not ensure mutual exclusion when accessing shared data, they exhibit undefined behavior according to the C memory model. However, if *sequential consistency can be assumed (1)* these solutions would be correct. These solutions would also be correct if either *acquiring a lock disallows all other threads from executing (6)*, or if the semantics of the code were relaxed, namely assuming that *a function is only called concurrently with itself, not with other functions (8)*, in this case for both `idea_add` and `idea_get`.

Another interesting solution used one global lock to protect the `while`-loop in `idea_get`, but rather than holding the global lock until removal was complete, acquired the lock associated with the found element and released the global lock before removal. Even though this solution might at first look correct, it does not actually ensure that two threads do not attempt to remove the same element twice. It is, however, correct if *a function is allowed to fail sporadically (9)*, in this case if `idea_get` is allowed to return `NULL` even if the buffer contains an element. This also requires assuming that *acquiring a lock disallows all other threads from executing (6)*.

5.2.2. *Multiple Calls to Acquire*

One interesting, but incorrect, solution was to acquire a global lock as the first statement of the loop body in `idea_add` (A in Listing 3). It is indeed a correct observation that the loop header does not need to be protected by the lock, but acquiring the lock multiple times in the loop without also releasing it (the release was after the line marked B) is problematic as the number of calls to acquire and release do not necessarily match. As such, this solution would be correct if *the acquire and release operations define a lexical region that is protected by the lock (5)*. This could indeed be achieved in a language which has first-class support for synchronization primitives, but this is not the case for C.

5.3. *Unique to the Second Question*

In this section we present categories which were only visible in solutions to the second question. Once again, this does not necessarily mean that they are due to the second problem itself, but rather that they did not surface in the first question (e.g., since there was no possibility of deadlock there).

5.3.1. *Semantics*

Solutions in this category alter the semantics of the `make_drink` function in some way. One set of these solutions modify the code to check and subtract each ingredient individually. This is possible to do correctly with some extra care: if the code finds an insufficient amount of the second ingredient, it needs to restore the amount subtracted from the first ingredient to preserve proper semantics. This was not done by a large number of these solutions, and they do thereby not preserve the semantics of the original code as they may decrease one ingredient but not the other. Even with proper care, this solution means that *a function is allowed to fail sporadically (9)*, in this case that one call

`make_drink` may cause other concurrent calls to fail sporadically if supplies are low. The question does not specify whether this is a problem or not.

A number of solutions also attempted to replace or augment the variable `amount` in `ingredient` with a semaphore or a condition variable in order to wait for a sufficient amount of ingredients to become available, rather than returning `false` from `make_drink`. As such, this is correct if the function should be *waiting for a condition to be fulfilled rather than failing (10)*, even though the question explicitly states that the supply is filled before `make_drink` is called, and not concurrently with it.

Another semantic issue was that a number of solutions attempted to synchronize other functions, such as `add_supply`, even though the question and the comments in the code state that they are not executed concurrently with `make_drink`. While too much synchronization typically does not hurt the correctness of the program, it is considered incorrect in the context of the questions since students are asked to strive to minimize their synchronization. Thus, these solutions would be correct if *make_drink could be executed concurrently with the add_* functions (11)*.

5.3.2. *Protecting Local Data*

Solutions in this category are correct in the sense that they do not exhibit concurrency issues, but incorrect in the sense that they needlessly limit the ability to execute code concurrently by protecting local variables with locks.

First and foremost, two solutions locked the assignment to `recipe` and the call to `find_recipe` (marked A in Listing 5). Since none of them attempted to protect any of the `add_*` functions, this is most likely not due to the realization that calling `find_recipe` while adding a recipe is problematic. Rather, since both solutions noted that they protected the variable `name`, and one also noted the variable `recipe`, this protection is sensible if we assume that *local variables are shared between concurrent calls to the same function (2)*. This issue may also be due to excessive use of pattern matching:

```
bool make_drink(const char *name) {
  A{struct recipe *recipe = find_recipe(name);
    // Does the recipe exist?
  B{if (recipe == NULL)
       return false;
    // Do the ingredients exist?
  C{struct ingredient *ing1
       = find_ingredient(recipe->ingredient1);
    struct ingredient *ing2
       = find_ingredient(recipe->ingredient2);
D{   if (ing1 == NULL || ing2 == NULL)
       return false;
    // ...
}
```

Listing 5. Local data protected in the second question.

These students may have learned that pointers are often problematic, and therefore opt to lock code dealing with pointers. Some solutions also protected the check if `rec-ipe` is `NULL` (marked B in Listing 5) in isolation. These solutions would be sensible if *local variables are shared between concurrent calls to the same function (2)*, but would also require that *using a lock inside a function prevents the entire function from being executed concurrently (4)*.

Secondly, we found a number of solutions that lock the assignments to `ing1` and `ing2`, as well as the two calls to `find_ingredient` (C), sometimes including the checks for `NULL` (D). Many of them explicitly point out that the variables `ing1` and `ing2` need to be protected, and thus these solutions would also be correct if *local variables are shared between concurrent calls to the same function (2)*. Some of these solutions also suffer from the problem that they release the lock and re-acquire the lock in the function, which could cause other threads to alter the shared locals, thus also requiring that *using a lock inside a function prevents the entire function from being executed concurrently (4)*.

5.3.3. *Lock Placement*

Solutions in this category are incorrect since the locks are not properly associated with the data they protect. One example of this is solutions that used a lock in the `recipe` struct to protect the ingredients in there. This means that the lock will ensure that only one thread uses a particular recipe, rather than the ingredients required by the recipe. Therefore, race conditions are not eliminated if two recipes share some ingredient. This kind of solution would be correct if *different recipes are not allowed to share ingredients (12)*. This is, however, shown not to be true by the main function. As such, another possibility is that *pointers work like values (3)*, which in this case means that the ingredients are copied as they are returned from `find_ingredient`. Yet another alternative abstraction that would make these solutions correct is if *acquiring a lock disallows all other threads from executing (6)*, since that means it is not important *which* lock is used to protect data.

Another type of solution was to use two global locks to protect `ing1` and `ing2` independently. Due to the nature of these solutions, they are most likely related to the belief that *local variables are shared between concurrent calls to the same function (2)*, but to make these solutions correct it is also necessary to assume that *pointers work like values (3)*. Another possibility would once again be that *acquiring a lock disallows all other threads from executing (6)*.

A final type of solution found in this category was solutions that used locks declared as local variables inside the `make_drink` function. This is incorrect since each invocation of the function will use a separate instance of the lock, and thus each invocation is guaranteed immediate access to the critical section. Therefore, this solution would indeed be correct if *local variables are shared between concurrent calls to the same function (2)*, or if *acquiring a lock disallows all other threads from executing (6)*, since then it would not matter which lock was acquired. It is worth noting that according to Strömbäck *et al.* (2019) this was visible in the first question as well. It was, however, not in the dataset examined in this paper.

5.3.4. *Possible Deadlock*

Solutions in this category used multiple locks (one for each ingredient), but failed to ensure that deadlocks were not possible. As previously mentioned, this could be done by making sure to always acquire locks in a particular order. By examining the answer to (d), we found that most students were aware of the issue, at least when it was pointed out to them. Some students even suggested a solution to the problem. The most common solution was to ensure that locks were always acquired in some global order. Other solutions, such as separating the critical sections for the two ingredients (thus altering semantics as previously mentioned) were also suggested.

  Six of these solutions argue that there is no risk for deadlock. This was either due to failing to understand some of the semantics of their solution (e.g., arguing that there is no mutual exclusion), or by arguing that the example recipes provided in the `main` function have no risk of causing a deadlock when only two robots were used. While this observation is true (the example recipes need three robots to exhibit a deadlock), the code inside the `main` function was labeled as a conservative example.

5.4. *Summary of Incorrect Abstractions*

To conclude the results section, we list all alternative abstractions proposed previously. To aid readability, we order these assumptions according to the abstraction they modify:

The Memory Model:
  1. Sequential consistency can be assumed.
  2. Local variables are shared between concurrent calls to the same function.

Abstractions in C:
  3. Pointers work like values (i.e., pointer assignments copy the data pointed to).

Abstractions for Synchronization:
  4. Using a lock inside a function prevents the entire function from Being executed concurrently.
  5. The acquire and release operations define a lexical region that is protected by the lock.
  6. Acquiring a lock disallows all other threads from executing.

Concurrent Aspects of Abstractions:
  7. A function is only called concurrently with other functions, not with itself.
  8. A function is only called concurrently with itself, not with other functions.
  9. A function is allowed to fail sporadically.
  10. Waiting for a condition to be fulfilled rather than failing.

Specific to the Questions:
  11. `make_drink` could be executed concurrently with the `add_*` functions.
  12. Different recipes are not allowed to share ingredients.

6. Discussion

In this section, we will first discuss the validity and limitations of the method and the results. After that, we will further examine the incorrect abstractions found in the results and discuss how they can be applied to teaching in the future.

6.1. *Validity and Limitations*

First and foremost, it is worth reiterating that the method used in this paper relies on the assumption that students reason logically in the realm of their understanding of the abstractions that the problem relies on. This is the same assumption that is used by Marton (2014, ch. 4) when arguing for the validity of Phenomenographic studies of how a cohort of students understand a particular object of learning. While we believe this assumption to be true in general, as noted by Albrecht and Grabowski (2020), it is necessary to take into account the fact that it is easy to make mistakes in one's reasoning when working with complex problems, especially if one's view of the underlying abstractions is incomplete or contains contradictions. Because of this, we believe that our approach of finding alternative abstractions for individual categories in isolation, rather than for the entire solution as a whole, works well. Not only is it much more feasible to conduct for larger data sets, but it is also less affected by students possibly failing to consider all aspects of a relatively large and complex problem, and is therefore less affected by mistakes or slips made by the students. This could, however, mean that we fail to see the implications of how certain combinations of errors interact with each other. Furthermore, it is worth pointing out that while we believe that this approach gives a good approximation of students' reasoning while solving a problem, it is an approximation based only on their answers and is not yet validated.

Even though it might be difficult to find all possible ways in which the underlying abstractions can be modified in order to make a solution correct, our data set of 137 solutions makes it likely that we have found most of them regardless. Using two separate problems rather than a single one is also beneficial since it reduces the risk of one problem hiding certain types of errors. This benefit is clearly visible in Fig. 3, which shows that many categories only contained errors from one question. Thus, excluding one of the problems would have removed many of the categories, and thereby also many alternative abstractions. It is, however, worth noting that the types of errors visible in this paper are most likely influenced by how the subject is taught. For example, relaxation number 1 (sequential consistency can be assumed) might be prominent since concurrency is introduced in terms of preemption on a single-threaded system, and might not be as clearly visible in other settings.

It is also worth pointing out that even though we provide an overview of how common certain types of errors are in Fig. 3, similarly to Strömbäck *et al.* (2019), it is not possible to draw definitive conclusions regarding how common it is for students to use a particular incorrect abstraction. This is since some errors may be caused by more than one combination of such abstractions, and in these situations it is not possible to deter-

mine which of the alternatives were actually used by the student. This kind of estimations are further complicated by the fact that there are likely scenarios where approximating the reason for errors in terms of alternative abstractions is not ideal. One good example of this is solutions with too small critical sections. Here, students might simply have failed to realize which overall constraints were important to protect rather than believing that locks automatically protect the entire function they are used in. Students with particularly short critical sections may also have resorted to pattern matching (e.g., believing that it is enough to protect all writes to shared data individually).

Taking these issues into account, we still believe that this method provides interesting and valuable results that reveal how students experience concurrency and the abstractions provided by a concurrent execution environment.

6.2. *Interpretation of the Results*

Based on the previous discussion of limitations, we will now discuss how the results may be interpreted and used. While most of them have a clear connection to incorrect beliefs that a student might have, some do not. As previously noted, this might be due to failure to reason about the abstractions in a larger problem rather than understanding the abstractions themselves. Therefore, we will discuss how the alternative abstractions can be interpreted, with the goal of providing important aspects of concurrency to help students discern when teaching concurrency. We will start by discussing the incorrect abstractions related to the memory model, abstractions in C, and abstractions for synchronization (number 1–6). This discussion constitutes our answers to RQ1 and RQ2.

1. Since concurrency was introduced in terms of preemption in a system with a single CPU, assuming sequential consistency is not far-fetched since sequential consistency actually holds in such a system (ignoring compiler optimizations). Some combinations of assumptions only require total store ordering, but not in isolation.

2. Since both recursion and scope are pointed out as difficult but important concepts for beginners by Goldman *et al.* (2008), it is not unlikely that some students assume that *local variables are shared between concurrent calls to the same function*. This could, for example, be a direct consequence of believing that *all* variables are global. One consequence of this belief would be difficulties in tracing recursion, since that also requires keeping different instances of local variables apart.

   Even though this relaxation was only seen in the solutions to question B, it is probably present in the solutions to question A as well but not clearly visible. For example, the synchronization of `idea_add` would look almost the same if locals were shared, only that the line before the loop would have to be protected as well. Since this is a small difference, these solutions were still considered to be correct without alternative abstractions and did not show up in our analysis. Some examples for part (a) of the question also pointed in this direction, further suggesting that this belief is indeed present in solutions to question A as well.

3. The belief that *pointers work like values*, or some variation thereof, is also common in introductory programming courses (Goldman *et al.*, 2008). Thus, the same thing being a problem when working with concurrency is no surprise.

4. The assumption that *using a lock inside a function prevents the entire function from being executed concurrently* is one of the alternative abstractions that is less likely to correspond to the belief of the student. Most of the solutions that were correct under this modified abstraction had too small or split critical sections, which could be caused by the student failing to realize what invariants need to be maintained by the data structure, for example.

5. Assuming that *the acquire and release operations define a lexical region that is protected by the lock* is a valid interpretation of how critical sections work. In fact, there are languages that take this approach to synchronization (e.g., `synchronized` blocks in Java). This is, however, not the case in C, as locks are acquired and release through regular function calls.

6. Assuming that *acquiring a lock disallows all other threads from executing* is also a natural first step when concurrency is introduced as preemption in a system with a single CPU. In such a system, a lock could simply prevent preemption from occurring, thus solving all concurrency issues.

The following four categories highlight incorrect ways in which students understood the abstractions described by the two questions. As such, these categories cover the additional aspects one needs to consider when implementing or using an abstraction (e.g., an abstract data structure) in a concurrent environment, and thus our answer to RQ3. These four categories are described below:

7. Assuming that *a function is only called concurrently with other functions, not with itself* likely arises from a student who fails to realize that concurrent calls to the same function are allowed. This could be simply because it is easier to trace concurrent execution in two different functions, or due to some other reason (e.g., believing that locals are shared between threads) that makes it infeasible to allow concurrent calls to a function.

8. In contrast to the previous assumption, assuming that *a function is only called concurrently with itself, not with other functions* likely arises from failure to see how multiple functions interact with each other. This could, in turn, either be due to failure to realize how data is shared through pointers or references, or failure to understand the desired behavior of the abstraction that is described in the question.

9. Assuming that *a function is allowed to fail sporadically* is likely either due to students failing to account for some possible interaction between threads in the program (the issue in question B is indeed quite difficult to spot), or due to students believing that some (unlikely) failures are acceptable. Kolikant (2005) found the latter to be a reason for some portion of student errors in concurrent programs, and used the term *alternative correctness* to describe the phenomenon.

10. Finally, assuming that a function shall *wait for a condition to be fulfilled rather than failing* illustrates one important but problematic aspect of concurrent abstractions. Namely, *when* something is to happen. This could be because students failed

to distinguish between different types of waiting. In this particular case, the two options are: 1) acquire a lock and check if a condition is true, which may require waiting if another thread is holding the lock, and 2) wait for the condition to become true using a semaphore. Both options may require waiting, but option 1) is guaranteed to not keep threads waiting forever (i.e., *progress*), while option 2) may require other parts of the program to call some function to avoid unbounded waiting. This key difference is likely not trivial to see, as it often requires a higher level view of the system.

Finally, we found two categories specific to the second question. First, some student assumed that *all* functions may be called concurrently, even though the problem states that this is not the case. This is essentially the opposite of categories 7 and 8. It is, however, less problematic as too much synchronization rarely makes a solution incorrect, it only degrades performance. Finally, the assumption that recipes do not share ingredients (number 12) is most likely related to issues with differentiating between pointers and values, but might also be due to failing to understand the problem statement.

6.3. *Applications of the Results*

There are two immediate applications of these results. First and foremost, the alternative abstractions, as previously discussed, provide an insight into areas that benefit from additional care when teaching concurrency. In particular, many of these alternative abstractions have implications that are not always visible in smaller examples that may be suitable to teach during class. They only surface in slightly larger and more complex examples that involve multiple functions and multiple instances of data structures. Additionally, some of them also highlight the importance of connecting the new concurrency concepts to more fundamental language concepts that are typically taught in introductory courses (e.g., pointers and scope).

These insights are also valuable when designing other kinds of teaching aids, such as visualization tools. Once again, being aware of these incorrect abstractions makes it possible to design a visualization that highlights why they are incorrect, and thereby allow students to acquire a correct understanding of these important abstractions. In order to address some of the incorrect abstractions, it is necessary to provide a visualization that embeds concurrency into its context of more fundamental concepts, rather than focusing entirely on concurrency. It is also worthwhile to illustrate how concurrency interacts with other abstractions, such as data structures implemented in the language, by highlighting what operations are allowed to be called concurrently.

6.4. *Future Work*

There are many ways in which the work presented in this paper can be extended in the future. One path of future work is to collect data from different problems and/or from

different institutions in order to find any abstractions that are missing from the list presented previously, and/or to validate the results in this paper. As noted previously, different types of problems and perhaps also different approaches of teaching concurrency could make other incorrect abstractions surface. This could then, in turn, be used to build a codebook of skills (perhaps with associated incorrect abstractions) that are important to assess in concurrency courses, similarly to what Nelson *et al.* (2020) did for prerequisite skills. Such a codebook is useful for creating differentiated assessments (i.e., assessments that can tell students what part of a subject they need to practice further), and also concept inventories for the subject in the future.

The list of incorrect abstractions could also be used to create assignments that highlight these issues, use these assignments while teaching concurrency, and evaluate whether such interventions are effective in helping students build the correct abstractions. These incorrect abstractions could also be used to design visual representations that can be used in visualization tools to illustrate how the correct abstractions work, and why the ones found here are incorrect. In particular, none of the visualization tools presented previously allow exploring how concurrency abstractions interact with pointers vs. values. It is also a challenge to create a useful and intuitive visualization of the weak C/C++ memory model. Such tools could then be used, perhaps in conjunction with the previously mentioned assignments, when teaching concurrency to further aid students.

7. Conclusion

In this paper we collected and analyzed 137 solutions to two concurrency questions. To estimate what incorrect abstractions students use when reasoning about concurrency, we proposed one or more incorrect abstractions under which each of the 61 incorrect solutions would be correct. In this analysis, we considered incorrect abstractions both related to the concurrent execution environment, and regarding the abstract data types covered by the two questions. The analysis resulted in 10 incorrect abstractions that are presented in Section 5.4. These abstractions describe problematic assumptions made by the solutions, such as assuming a stronger memory model than what is actually provided, assuming that certain functions are not called concurrently, or allowing functions to fail sporadically.

Since these modifications were derived from incorrect solutions, they indicate some aspect of the abstraction that students find difficult or failed to understand entirely. The analysis in this paper does not, however, give any exact measure of ho*w* difficult each of these aspects are. Regardless, being aware of these alternative abstractions makes it possible to design relevant problems and examples so that the sometimes hidden difficulties with these abstractions can be properly addressed when teaching. They can also be used to design visualizations to highlight these concepts, so that students can explore these areas for themselves.

References

Adams, J.C., Koning, E.R., Hazlett, C.D. (2019). Visualizing classic synchronization problems: Dining Philosophers, Producers-Consumers, and Readers-Writers. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Association for Computing Machinery, New York, NY, USA, pp. 934–940. 9781450358903. `https://doi.org/10.1145/3287324.3287467`

Albrecht, E., Grabowski, J. (2020). Sometimes it's just sloppiness -studying students' programming errors and misconceptions. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Association for Computing Machinery, New York, NY, USA, pp. 340–345. 9781450367936. `https://doi.org/10.1145/3328778.3366862`

Alexandron, G., Armoni, M., Gordon, M., Harel, D. (2016). Teaching Nondeterminism through programming. *Informatics in Education*, 15(1), 1–23. `https://doi.org/10.15388/infedu.2016.01`

Alglave, J., Maranget, L., Sarkar, S., Sewell, P. (2010). Fences in weak memory models. In: Touili, T., Cook, B., Jackson, P. (Eds.), *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 258–272. 978-3-642-14295-6.

Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T. (2011). Mathematizing C++ concurrency. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. ACM, New York, NY, USA, pp. 55–66. 978-1-4503-0490-0. `https://doi.org/10.1145/1926385.1926394`

Bijlsma, A., Huizing, C., Kuiper, R., Passier, H.J.M., Pootjes, H.J., Smetsers, J.E.W. (2017). A structured design methodology for concurrent programming. In: *Proceedings of the 6th Computer Science Education Research Conference*. CSERC '17. Association for Computing Machinery, New York, NY, USA, pp. 1–9. 9781450363389. `https://doi.org/10.1145/3162087.3162088`

Bijlsma, L., Huizing, K., Kuiper, R., Passier, H., Pootjes, H., Smetsers, S. (2019). Evaluation of a structured design methodology for concurrent programming. In: *Proceedings of the 8th Computer Science Education Research Conference*. CSERC '19. Association for Computing Machinery, New York, NY, USA, pp. 58–65. 9781450377171. `https://doi.org/10.1145/3375258.3375266`

Bogaerts, S.A. (2017). One step at a time: Parallelism in an introductory programming course. *Journal of Parallel and Distributed Computing*, 105, 4–17. Keeping up with Technology: Teaching Parallel, Distributed and High-Performance Computing. `https://doi.org/10.1016/j.jpdc.2016.12.024`

Burtscher, M., Peng, W., Qasem, A., Shi, H., Tamir, D., Thiry, H. (2015). A module-based approach to adopting the 2013 ACM curricular recommendations on parallel computing. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Association for Computing Machinery, New York, NY, USA, pp. 36–41. 9781450329668. `https://doi.org/10.1145/2676723.2677270`

Choi, S.-E., Lewis, E.C. (2000). A study of common pitfalls in simple multi-threaded programs. *SIGCSE Bull.*, 32(1), 325–329. `https://doi.org/10.1145/331795.331879`

Desell, T. (2013). Using actors and the SALSA programming language to introduce concurrency in computer science II. In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 1257–1262. `https://doi.org/10.1109/IPDPSW.2013.153`

Ernst, D.J., Stevenson, D.E. (2008). Concurrent CS: Preparing students for a multicore world. In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '08. Association for Computing Machinery, New York, NY, USA, pp. 230–234. 9781605580784. `https://doi.org/10.1145/1384271.1384333`

Exton, C. (2000). Elucidate: A tool to aid comprehension of concurrent object oriented execution. *SIGCSE Bull.*, 32(3), 33–36. `https://doi.org/10.1145/353519.343066`

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M.C., Zilles, C. (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, New York, NY, USA, pp. 256–260. 978-1-59593-799-5. `https://doi.org/10.1145/1352135.1352226`

Kolikant, Y.B.-D. (2001). Gardeners and cinema tickets: High school students' preconceptions of concurrency. *Computer Science Education*, 11(3), 221–245.

Kolikant, Y.B.-D. (2004). Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies*, 60(2), 243–268. `https://doi.org/10.1016/j.ijhcs.2003.10.005`

Kolikant, Y.B.-D. (2005). Students' alternative standards for correctness. In: *Proceedings of the First International Workshop on Computing Education Research*. ICER '05. ACM, New York, NY, USA, pp. 37–43. 1-59593-043-4. `https://doi.org/10.1145/1089786.1089790`

Lamport (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), 690–691. `https://doi.org/10.1109/TC.1979.1675439`

Lawson, A., Kraemer, E.T. (2020). Sidekicks and superheroes: A look into student reasoning about concurrency with threads versus actors. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET ’20. Association for Computing Machinery, New York, NY, USA, pp. 82–92. 9781450371247. `https://doi.org/10.1145/3377814.3381706`

Lawson, A., Kraemer, E.T., Che, S.M., Kennedy, C. (2019). A multi-level study of undergraduate computer Science reasoning about concurrency. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’19. Association for Computing Machinery, New York, NY, USA, pp. 210–216. 9781450368957. `https://doi.org/10.1145/3304221.3319763`

Lewandowski, G., Bouvier, D.J., McCartney, R., Sanders, K., Simon, B. (2007). commonsense computing (episode 3): Concurrency and concert tickets. In: *Proceedings of the Third International Workshop on Computing Education Research*. ICER ’07. Association for Computing Machinery, New York, NY, USA, pp. 133–144. 9781595938411. `https://doi.org/10.1145/1288580.1288598`

Lin, S., Tatar, D. (2011). Encouraging parallel thinking through explicit coordination modeling. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*. SIGCSE ’11. ACM, New York, NY, USA, pp. 441–446. 978-1-4503-0500-6. `https://doi.org/10.1145/1953163.1953292`

Lönnberg, J. (2012). *Understanding and Debugging Concurrent Programs through Visualisation*. G5 artikkeliväitöskirja. 978-952-60-4530-6. `http://urn.fi/URN:ISBN:978-952-60-4530-6`

Lönnberg, J., Berglund, A. (2007). Students’ understandings of concurrent programming. In: *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research – Volume 88*. Koli Calling’07. Australian Computer Society, Inc., Darlinghurst, Australia, pp. 77–86. 978-1-920682-69-9. `http://dl.acm.org/citation.cfm?id=2449323.2449332`

Lönnberg, J., Berglund, A., Malmi, L. (2009). how students develop concurrent programs. In: *Proceedings of the Eleventh Australasian Conference on Computing Education -Volume 95*. ACE ’09. Australian Computer Society, Inc., Darlinghurst, Australia, pp. 129–138. 978-1-920682-76-7. `http://dl.acm.org/citation.cfm?id=1862712.1862732`

Lönnberg, J., Malmi, L., Ben-Ari, M. (2011a). Evaluating a visualisation of the execution of a concurrent program. In: *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. Koli Calling ’11. ACM, New York, NY, USA, pp. 39–48. 978-1-4503-1052-9. `https://doi.org/10.1145/2094131.2094139`

Lönnberg, J., Malmi, L., Berglund, A. (2008). Helping students debug concurrent programs. In: *Proceedings of the 8th International Conference on Computing Education Research*. Koli ’08. ACM, New York, NY, USA, pp. 76–79. 978-1-60558-385-3. `https://doi.org/10.1145/1595356.1595369`

Lönnberg, J., Mordechai, B.-A., Malmi, L. (2011b). Visualising concurrent programs with dynamic dependence graphs. In: *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pp. 1–4. `https://doi.org/10.1109/VISSOF.2011.6069456`

Ma, L., Ferguson, J., Roper, M., Wood, M. (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, New York, NY, USA, pp. 499–503. 1-59593-361-1. `https://doi.org/10.1145/1227310.1227481`

Marton, F. (2014). *Necessary Conditions of Learning*. Routledge, New York, NY, USA. 9780415739146.

Nelson, G.L., Strömbäck, F., Korhonen, A., Begum, M., Blamey, B., Jin, K.H., Lonati, V., MacKellar, B., Monga, M. (2020). Differentiated assessments for advanced courses that reveal issues with prerequisite skills: A Design Investigation. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR ’20. Association for Computing Machinery, New York, NY, USA, pp. 75–129. 9781450382939. `https://doi.org/10.1145/3437800.3439204`

Offenwanger, A., Lucet, Y. (2014). ConEE: An exhaustive testing tool to support learning concurrent programming synchronization challenges. In: *Proceedings of the Western Canadian Conference on Computing Education*. Association for Computing Machinery, New York, NY, USA. 9781450328999. `https://doi.org/10.1145/2597959.2597972`

Prasad, S., Gupta, A., Rosenberg, A., Sussman, A., Weems, C.J. (2015). *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses* (1st ed.). Elsevier, Amsterdam, Netherlands. 9780128038994.

Prasad, S., Gupta, A., Rosenberg, A., Sussman, A., Weems, C.J. (2018). *Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms*. Springer, Berlin, Heidelberg. 978-3-319-93108-1.

Qasem, A., Bunde, D.P., Schielke, P. (2021). A module-based introduction to heterogeneous computing in core courses. *Journal of Parallel and Distributed Computing*, 158, 56–66. https://doi.org/10.1016/j.jpdc.2021.07.011

Raj, R.K., Romanowski, C.J., Impagliazzo, J., Aly, S.G., Becker, B.A., Chen, J., Ghafoor, S., Giacaman, N., Gordon, S.I., Izu, C., Rahimi, S., Robson, M.P., Thota, N. (2020). High performance computing education: Current challenges and future directions. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '20. Association for Computing Machinery, New York, NY, USA, pp. 51–74. 9781450382939. https://doi.org/10.1145/3437800.3439203

Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O. (2010). X86-TSO: A rigorous and usable programmer's model for X86 multiprocessors. *Commun. ACM*, 53(7), 89–97. https://doi.org/10.1145/1785414.1785443

Silberschatz, A., Galvin, P.B., Gagne, G. (2010). *Operating System Concepts* (8th ed.). Wiley Publishing, New Jersey, USA. 978-0-470-23399-3.

Strömbäck, F., Mannila, L., Kamkar, M. (2020). Exploring students' understanding of concurrency – A phenomenographic study. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Association for Computing Machinery, New York, NY, USA, pp. 940–946. 9781450367936. https://doi.org/10.1145/3328778.3366856

Strömbäck, F., Mannila, L., Asplund, M., Kamkar, M. (2019). A student's view of concurrency – A study of common mistakes in introductory courses on concurrency. In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER '19. ACM, New York, NY, USA, pp. 229–237. 9781-4503-6185-9. https://doi.org/10.1145/3291279.3339415

The ACM/IEEE Computer Science Curriculum Joint Task Force (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, United States. 978-1-4503-2309-3.

Valstar, S., Griswold, W.G., Porter, L. (2019). The relationship between prerequisite proficiency and student performance in an upper-division computing course. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Association for Computing Machinery, New York, NY, USA, pp. 794–800. 9781450358903. https://doi.org/10.1145/3287324.3287419

Xie, S., Kraemer, E., Stirewalt, R.E.K. (2007). Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In: *29th International Conference on Software Engineering (ICSE'07)*, pp. 727–731. https://doi.org/10.1109/ICSE.2007.31

F. Strömbäck is a PhD student and a lecturer at the Department of Computer and Information Science at Linköping University, Sweden. He has been teaching computer science since 2012, primarily teaching courses on concurrency, operating systems, as well as data structures and algorithms. He started his PhD studies in 2018, and his main research interest is teaching and learning concurrency. As a part of this work, he has also studied prerequisites to learning concurrency, for example by co-chairing an ITiCSE working group on prerequisite skills.

L. Mannila is a researcher in computer science education at Linköping University, Sweden. Her research interests include questions related to computational thinking, digital competence and programming at K-9 level, both from a student, teacher and organizational perspective.

M. Kamkar is a professor in Software Engineering, specializing in large-scale industrial software engineering at Linköping University, Sweden. Her research interests include software testing and debugging.

Appendix A. Example Program for the First Question

Below is the example code provided alongside the code for the first question (see Section 4.3).

```
1  /**
2   * Main program. This is only to illustrate how we can use the
3   * code above. All changes below will be ignored during grading.
4   */
5
6
7  const char *sample_ideas = {
8      "Bakeries", "Card games", "Dice", "Cookies", NULL
9  };
10
11 void worker(void *param) {
12     struct idea_buffer *buffer = param;
13
14     for (int i = 0; sample_ideas[i]; i++) {
15         idea_add(buffer, sample_ideas[i]);
16     }
17 }
18
19 int main(void) {
20     srand(time(NULL));
21
22     struct idea_buffer buffer;
23     idea_init(&buffer);
24
25     thread_create("worker", 0, &worker, &buffer);
26
27     for (int i = 0; sample_ideas[i]; i++) {
28         printf("Idea: %s\n", idea_get(&buffer));
29     }
30
31     return 0;
32 }
```

Appendix B. Example Program for the Second Question

Provided below are the operations that were left out of the Listing in Section 4.4.

```
1  #define MAX_INGREDIENTS 20
2  #define MAX_RECIPES 20
3
4  // Current supply of ingredients in the bar.
5  struct ingredient supply[MAX_INGREDIENTS];
6  int supply_count = 0;
7  // All recipes the bar knows how to make.
8  struct recipe recipes[MAX_RECIPES];
9  int recipe_count = 0;
```

```
10
11 // Initialize the bar. Called once at the start of the program.
12 void init_bar() {
13     supply_count = 0;
14     recipe_count = 0;
15 }
16 // Add an ingredient to the bar. If we call "add_supply" for an
17 // ingredient that already exists, the amount available is
18 // increased. Otherwise, a new one is added.
19 // We assume that this function is not called concurrently to
20 // other functions in the program.
21 void add_supply(const char *name, int amount) {
22     // Already available?
23     struct ingredient *found = find_ingredient(name);
24     if (found) {
25         found->amount += amount;
26         return;
27     }
28
29     // Not found... add a new element!
30     int id = supply_count;
31     supply[id].name = name;
32     supply[id].amount = amount;
33     supply[id].index = id;
34     supply_count++;
35 }
36 // Add a recipe.
37 // We assume that this function is not called concurrently to
38 // other functions in the program.
39 void add_recipe(const char *name, const char *ing1, int amount1,
40                 const char *ing2, int amount2) {
41     struct recipe *recipe = &recipes[recipe_count];
42     recipe->name = name;
43     recipe->ingredient1 = ing1;
44     recipe->amount1 = amount1;
45     recipe->ingredient2 = ing2;
46     recipe->amount2 = amount2;
47
48     recipe_count++;
49 }
50 // Find an ingredient. Returns NULL if none was found.
51 struct ingredient *find_ingredient(const char *name) {
52     for (int i = 0; i < supply_count; i++) {
53         if (strcmp(name, supply[i].name) == 0) {
54             return &supply[i];
55         }
56     }
57     return NULL;
58 }
59 // Find a recipe. Returns NULL if none was found.
60 struct recipe *find_recipe(const char *name) {
61     for (int i = 0; i < recipe_count; i++) {
62         if (strcmp(name, recipes[i].name) == 0) {
63             return &recipes[i];
64         }
65     }
```

```
66      return NULL;
67 }
```

Below is the example code provided alongside the code for the second question (see Section 4.4).

```
1  /**
2   * Main program. This is only to illustrate how we can use the
3   * code above. All changes below will be ignored during grading.
4   */
5
6  void check(const char *drink) {
7      printf("Making %s... ", drink);
8      if (make_drink(drink))
9          printf("OK!\n");
10     else
11         printf("Failed...\n");
12 }
13
14 int main() {
15     init_bar();
16     add_supply("milk", 100);
17     add_supply("blueberry", 50);
18     add_supply("raspberry", 50);
19     add_supply("cacao", 10);
20
21     add_recipe("Blueberry milk", "blueberry", 5, "milk", 10);
22     add_recipe("Hot cocoa", "milk", 10, "cocoa", 1);
23     add_recipe("Chocolate berries", "cocoa", 1, "blueberry", 5);
24
25     // Make some drinks.
26     check("Blueberry milk");
27     check("Hot cocoa");
28     check("Chocolate berries");
29     check("Blueberry milk");
30     check("Hot cocoa");
31     check("Blueberry milk");
32     check("Hot cocoa");
33     check("Chocolate berries");
34     check("Blueberry milk");
35     check("Hot cocoa");
36     check("Blueberry milk");
37     check("Blueberry milk");
38     check("Blueberry milk");
39     check("Blueberry milk");
40
41     return 0;
42 }
```