Pilot Study of Progvis: A Visualization Tool for Object Graphs and Concurrency via Shared Memory

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

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

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

ABSTRACT

Concurrency and synchronization are two topics that are becoming increasingly important for computer science students due to the high number of cores available in most modern devices. These are topics that many students struggle with at first, perhaps partially due to the inherent nondeterminism and the difficulty to test for absence of race conditions. Furthermore, previous research indicate that some common mistakes when working with concurrency might be due students not connecting the concurrency concepts (such as synchronization) to the data that needs to be protected, especially when pointers and references are involved.

To address these issues, we propose Progvis, which is a visualization tool aimed specifically at concurrency using the shared memory model. It provides a detailed visualization of objects in memory and their relation to the running threads in order to help students connect concurrency issues with the affected data. We have performed an initial, small scale evaluation on whether using the tool helps students solve synchronization problems during voluntary problem-solving sessions. The preliminary results indicate that students who used the tool did indeed perform better.

CCS CONCEPTS

• **Applied computing → Education**; • **Human-centered computing → Visualization systems and tools**; • **General and reference** → *Empirical studies*; • **Theory of computation** → *Concurrency*.

KEYWORDS

concurrency, synchronization, visualization, computer science education

ACM Reference Format:
Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. Pilot Study of Progvis: A Visualization Tool for Object Graphs and Concurrency via Shared Memory. In *Australasian Computing Education Conference (ACE '22), February 14–18, 2022, Virtual Event, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511861.3511885

ACE '22, February 14–18, 2022, Virtual Event, Australia
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9643-1/22/02.
https://doi.org/10.1145/3511861.3511885

1 INTRODUCTION

Concurrency is an important topic for students to learn to be able to fully utilize the many cores available in most modern systems. Concurrency is, however, often experienced as a difficult subject, at least when working with synchronization and shared memory [18]. This might at least partially be due to a lacking understanding of the semantics of the language being used. One such area that is known to be difficult but important in introductory programming courses is pointers and references [3]. These concepts are known to be important in later courses, such as CS2 [20], and in courses covering concurrency [18]. One possible way to aid students in their learning of these concepts is to illustrate the behavior of programs by using suitable visualizations, ideally where students are able to interact with the visualization to a high degree to improve learning [12].

Furthermore, we have informally observed that many students who failed the exam and subsequently seek advice from the instructor struggle with relating synchronization primitives to data in concurrent programs. This is often in cases where multiple instances of some data type are involved. After discussing these matters with the instructor, these students typically score high marks on the next exam, indicating that this is an important aspect in a visualization tool in addition to other synchronization concepts.

In this paper we introduce a visualization tool called Progvis, which aims to help students learn concurrency by addressing these difficult aspects. Thus, Progvis focuses on showing how multiple threads interact through an object graph that represents the shared memory between the threads. We will also present the results of a pilot study in which students are asked to synchronize a piece of C code, both with and without the tool. From the pilot study, we hope to see whether using the tool helps students to arrive at correct solutions or not.

The remainder of this paper is organized as follows: Section 2 introduces concurrency and other related work in program visualization. The tool used in this paper is presented in Section 3. In Section 4, we describe how the evaluation was conducted, after which we present the results in Section 5. Finally, in Section 6, we discuss the results and conclude the paper in Section 7.

2 BACKGROUND

In this section, we first provide a brief introduction to the parts of concurrency that are relevant to this study, as well as research on teaching concurrency. Next, we present related work in program visualization in general and for concurrency in particular.

2.1 Concurrency

In this paper we focus on the concurrency model where multiple *threads* communicate using shared memory. This is the model used in many imperative languages (e.g., C++, Java, Python) and is supported by most major operating systems. In general, few assumptions can be made about the relative execution speed of different threads or the ordering of memory operations in relation to other threads [1]. To communicate safely, threads need to use some synchronization primitive, such as *locks*, *semaphores*, or *condition variables* or use *atomic operations*.

Students' experiences with concurrency has been previously studied by for example Kolikant [4, 5], who studied high-school students' solutions to concurrency problems. Kolikant found that many students failed to identify some synchronization goals, but that solving the problem was generally easy once the synchronization goals were identified. Lawson and Kraemer [7] have also studied students' solutions to similar problems, and found that students sometimes use sleep functions to avoid concurrency issues rather than proper synchronization primitives.

Aside from examining students' solutions, Lönnberg et al. [10, 11] have also investigated students' experiences with developing concurrent programs, and found a number of different approaches to developing concurrent programs, for example *trial and error* and *adapting a known technique* [11]. This was also done for concurrency in general by Strömbäck et al. [19]. The same authors also examined students' mistakes when solving an exercise involving synchronizing a data structure [18], and found a number of different common problems. The authors also hypothesized that a potential issue is lacking knowledge of parts of the semantics of the programming language. This is in line with results from Valstar et al. [20], who found that lacking skills in concepts introduced in CS1 have an impact on students' performance in CS2. As such, it is reasonable to believe that this is true for concurrency courses as well.

2.2 Program Visualization

A number of program visualization tools have been developed and studied with the aim to help students understand what their program is doing, and to help students develop an accurate representation of how their program is executed (i.e., an accurate model of the *notional machine*). Jeliot [8] and UUhistle [16] are two examples of such tools aimed at introductory programming in Java and Python respectively. Both of these tools visualize program execution at a great level of detail, and clearly illustrate how references work, which is a known to be a difficult concept [3]. Other tools include PlanAni [14], which visualizes variables differently according to how they are used, and OpenDSA [15], which visualizes higher level data structures interactively. Contrary to Jeliot and UUhistle, these tools are aimed at showing pre-defined examples rather than arbitrary code supplied or modified by the student (for PlanAni, this was planned but not complete [14]).

There are also a number of visualization tools aimed at concurrency. For example, 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, nor does it support references or pointers. A similar tool, called ConEE, was proposed by Offenwanger and Lucet [13]. This tool does allow loading arbitrary programs and automatically proving whether or not the code is free from race conditions, but does not address the issue of illustrating references. Another approach is explored by Eludicate [2], which provides execution traces that users may use to reason about the program. Lönnberg [9] has also examined a number of tools, for example a train simulation to introduce semaphores, and a tool called Atropos, which visualizes the program flow as a graph and allows executing the program both forwards and backwards. All of these tools, both ones aimed at concurrency and introductory programming, are summarized in Table 1.

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

Table 1: Summary of the capabilities of existing visualization tools and Progvis. These are described in further detail in Section 2.2 and Section 3. Note: Different visualizations in OpenDSA behave differently, which is why *illustrates references* is left blank.

Tool	*Language*	*Concurrency*	*High-level execution*	*Illustrates references*	*Editable code*
Jeliot [8]	Java	✗	✗	✓	✓
UUhistle [16]	Python	✗	✗	✓	✓
PlanAni [14]	Custom	✗	✗	✗	✗
OpenDSA [15]	Multiple	✗	✓	-	✗
The Deadlock Empire	C#	✓	✓	✗	✗
ConEE [13]	Custom	✓	✓	✗	✓
Eludicate [2]	Java	✓	✓	✗	✓
Atropos [9]	Java	✓	✓	✗	✓
Progvis	C/C++	✓	✓	✓	✓

Using visualizations does not necessarily help student learning. The visualization must also make sure to engage the students, as described by the engagement taxonomy proposed by an ITiCSE working group [12]. This taxonomy suggests that just looking at a visualization has little or no benefit for learning, while the ability to interact with and modify the visualization are much more beneficial. One possible way of enabling such interaction is to allow students to modify the source code of the programs that are visualized.

3 OUR TOOL: PROGVIS

In this section we introduce our tool, which is called *Progvis*. In contrast to the other tools summarized in Table 1, the aim of Progvis is not only to help students learn concepts related to concurrency; it also allows illustrating how concurrency interacts with other aspects of a concurrent program. For example, considering whether it is the responsibility of the caller or the callee to ensure that no data races occur when calling a function that accepts a pointer or reference to some data, and take this into account when specifying the interface of the function. In order to be able to illustrate such higher-level concerns, Progvis takes inspiration from the strengths of the tools mentioned in Section 2.2, and aims to provide a visualization that includes concurrency concepts as well as more fundamental concepts like references by providing a clear and correct representation of the underlying notional machine. Progvis does this by visualizing the data in the running program as an object graph, and shows how multiple threads interact with it.

In order to illustrate interactions between concurrency and other aspects of a concurrent program, Progvis currently supports visualizing programs written in a subset of C and C++. The decision to support a subset of C and C++ in favor of some other simpler language is to lower the barrier of entry for students to experience on their own. Since the course examined in this paper involves solving computer labs in C, students do not need to learn another language to use the visualizations. However, since the tool is implemented as a visual debugger in the polyglot Storm platform [17], it can be used to visualize any language supported by the Storm platform. This means that it is also possible to visualize programs written in a language called Basic Storm. Support for Java is also planned, and it is further possible to add support for more languages. Both Progvis and the Storm platform are available for Windows (x86) and Linux (AMD64).[2] For Linux, pre-compiled binaries are available for Debian-based systems, but it is known to work on other distributions as well, such as Arch Linux.

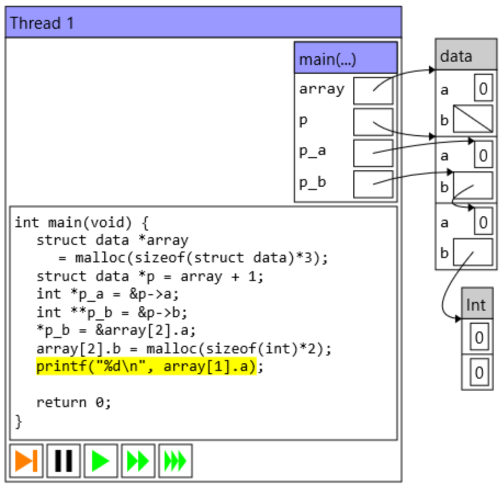
3.1 Object Graphs

As previously mentioned, Progvis aims to provide a clear and correct visualization of the notional machine. Of particular importance is the ability to correctly visualize pointers and references, which is known to be a difficult but important concept [3]. This is especially true when working with concurrent programs. Without a basic understanding of pointers or references, it is difficult to correctly identify shared data, which in turn makes synchronization difficult.

Figure 1 shows how Progvis visualizes the notional machine while running a single threaded program. The large box labeled *Thread 1* contains everything related to a single thread. It consists of two major parts: the current state of the execution stack, and the current location in the source code. The five buttons in the bottom of the box allow controlling the program execution by either proceeding a single step at a time, or by letting Progvis automatically step through the program at a specific rate. Since Progvis is aimed at students who are already familiar with sequential programs, it does not attempt to visualize each step of the computation in detail like UUhistle [16] or Jeliot [8], as can be seen in Table 1. Instead, Progvis focuses on individual statements, as that level of detail is more suitable when focusing on higher-level behaviors of a program, such as how it interacts with objects in memory or other threads.

The second, and perhaps most important, part of the box labeled *Thread 1* is the visualization of the execution stack. This part consists of one box for each stack frame on the call stack. In Fig. 1, only one such frame is active for the `main` function. If more functions would be called, more boxes will be shown on top of the current one, partially covering the variable names (as can be seen to the left in

[2]https://storm-lang.org/progvis.php



```
int main(void) {
   struct data *array
     = malloc(sizeof(struct data)*3);
   struct data *p = array + 1;
   int *p_a = &p->a;
   int **p_b = &p->b;
   *p_b = &array[2].a;
   array[2].b = malloc(sizeof(int)*2);
   printf("%d\n", array[1].a);

   return 0;
}
```

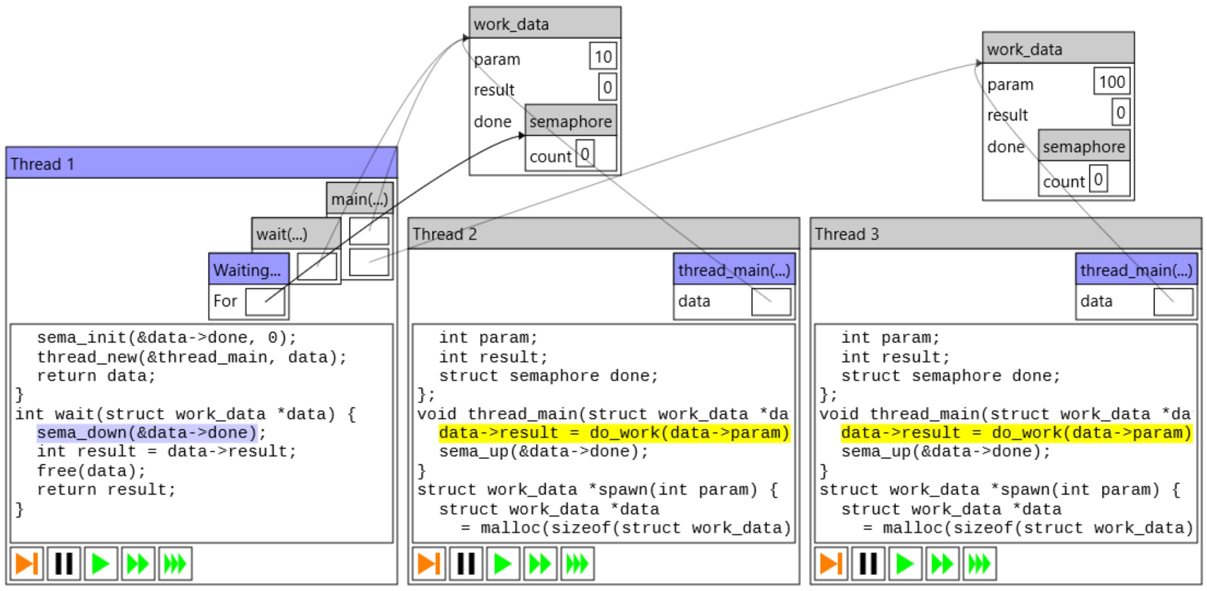
Figure 1: An object diagram generated by Progvis during execution of a single-threaded program.

Fig. 2). Each box contains a list of all local variables in the current function and a box that represents the storage allocated for the variable. In this case, all local variables are pointers, and they are therefore represented by arrows. For non-pointer variables, the box will instead contain a (possibly nested) representation of the data structure. In this way, the difference between values and pointers is clearly visible in the visualization. Furthermore, as can be seen in Fig. 1, Progvis is able to visualize many aspects of C pointers. In particular, the pointer `p` points to element 1 of the array pointed to by `array`, `p_a` and `p_b` point to variables inside an element of the array. It is worth noting that even though `p` and `p_a` technically contain the same address, Progvis visualizes them differently as the pointers have different types, and thereby logically refer to different parts of the data structure.

Even though Progvis was designed to be used in the context of concurrency, the detailed visualizations of pointers and objects allow it to be used in other contexts as well. One such example is to illustrate how pointer arithmetics work in C and C++. Another example is the difference between copy- and move-semantics in C++.

3.2 Concurrency

When concurrent programs are executed in Progvis, one box for each thread is shown, as can be seen in Fig. 2. This allows the user to control the different threads individually, which in turn lets the user explore different interleavings of the concurrent program, much like in The Deadlock Empire. In Fig. 2, the user let the first thread (*Thread 1*) execute until it created two instances of a data structure and spawned two threads (*Thread 2* and *Thread 3*) that were each given a reference to an instance of the data structure. Finally, the thread reached a `sema_down` statement which caused the thread to wait. This is indicated by the color in the source listing, and by a new frame in the call graph labeled *Waiting*, which indicates what the thread is currently waiting for. From the visualization, we can then see that only thread 2 has (indirect) access to the semaphore that



```
  sema_init(&data->done, 0);
  thread_new(&thread_main, data);
  return data;
}
int wait(struct work_data *data) {
  sema_down(&data->done);
  int result = data->result;
  free(data);
  return result;
}
```

```
  int param;
  int result;
  struct semaphore done;
};
void thread_main(struct work_data *da
  data->result = do_work(data->param)
  sema_up(&data->done);
}
struct work_data *spawn(int param) {
  struct work_data *data
    = malloc(sizeof(struct work_data)
```

```
  int param;
  int result;
  struct semaphore done;
};
void thread_main(struct work_data *da
  data->result = do_work(data->param)
  sema_up(&data->done);
}
struct work_data *spawn(int param) {
  struct work_data *data
    = malloc(sizeof(struct work_data)
```

Figure 2: An object diagram generated by Progvis during execution of a multi-threaded program. The leftmost thread is blocked on a call to `sema_down`, and the semaphore waited for is shown in the *Waiting* box.

thread 1 is waiting for. Thread 3 has a pointer to another instance of the same data structure, and is therefore not able to wake thread 1. This is an example of the strengths of Progvis compared to many other tools aimed at visualizing concurrent programs.

As indicated in Fig. 2, the implementation of the C language in Progvis supports a number of concepts related to concurrency. In addition to the function `thread_new` for spawning threads, three synchronization primitives are supported: *locks*, *semaphores* and *condition variables*. To minimize the effort required by students when working with Progvis, these synchronization primitives follow the same interface as the one used by the educational operating system Pintos used in the course. Support for other interfaces, such as *pthreads* can of course be implemented as well, but we have not yet found the need to do so.

This functionality combined with the visualization of object graphs and pointers provides a detailed visualization of concurrent programs. This allows students to explore how different interleavings affect the behavior of the visualized concurrent program. Based on this, students are also able to find interleavings that cause the program to misbehave, and thereby find concurrency errors. Since Progvis allows visualizing (almost) arbitrary C code, students are also able to attempt to solve the problems found by adding synchronization, and then see how the synchronization affects the execution of the concurrent program. This freedom means that students are able to engage to a high degree with the visualizations in Progvis, which as previously noted helps learning [12].

3.3 Limitations of the Tool

While Progvis allows visualizing a large number of programs, there are some limitations to consider. Some of these are inherent to the approach of detailed visualizations, and some are due to current shortcomings in the implementation. We will discuss these limitations in more detail here.

The largest limitation, which is inherent to the approach and thereby also applies for other similar tools, is that a detailed visualization of the object graph of a program quickly gets cluttered for more complex programs. For example, creating a linked list with a couple of hundred elements in, or indeed an array of a similar size, will make the visualization almost unusable. Furthermore, even though Progvis attempts to not visualize each step in the program, it quickly becomes tedious to step through large programs. Progvis addresses this by providing mechanisms to skip entire blocks of code, but more can be done to address this. As such, it is important to keep these limitations in mind when creating examples to visualize in Progvis. Otherwise, the visualization may be too cluttered to clearly illustrate the desired concurrency issues.

The remaining limitations are shortcomings in the current implementation, and can thus be addressed as needed. First and foremost, only a subset of C (and C++) is supported by the tool. Most notably, generic pointers (i.e., `void *`) and arbitrary pointer casts (sometimes referred to as *type punning*) are not supported. There are two reasons for this. First and foremost, the visualization needs to know the types of all data in memory. Allowing arbitrary typecasts (either through `void *` or through pointer casts) would make it impossible to reliably pick a sensible representation of data in memory. Secondly, Progvis executes programs in the same process as the visualization. Since student programs might crash, Progvis need to be able to reliably detect these issues and report them to the user rather than crashing the entire visualization. For this reason, the C implementation uses *fat pointers* (a pointer to the start of the allocation and an offset). This way, the implementation is able to verify that all memory references are safe before they are executed. The stronger type safety imposed by disallowing arbitrary pointer casts help hiding these fat pointers from the user, as well as ensuring memory safety. These limitations has not been a problem so far, as the focus of the tool is visualizing how concurrent programs behave, rather than the intricacies of low level memory manipulation.

The last limitation of the implementation is that it does not currently implement all details of the C memory model [1]. The version of the tool used in this study implements *sequential consistency* [6], meaning that reads and writes are guaranteed to happen in program order. This means that the tool does not report an error when two threads access shared data without proper synchronization. Rather, the tool treats each line (each step) as a unit that is executed atomically, and let students see cases where this lead to incorrect program behavior. We have found this to be enough at an introductory level of concurrent programming, as many programs still need synchronization to work properly. It does, however, give an incorrect picture of the memory model, as both the hardware and the compiler are allowed to reorder reads and writes that are not otherwise synchronized. To address this, we have later implemented support for Progvis to track and show reads and writes performed by the program. This helps to illustrate what the program is doing, and also lets Progvis to detect and report cases where a particular interleaving would cause multiple threads to access shared data concurrently. While this implementation is a step closer to the actual C memory model, it is not yet entirely accurate, as reads and writes are only tracked for individual statements. We plan to further develop this aspect of Progvis in the future.

4 PILOT STUDY

We evaluated Progvis in a pilot study during a course on concurrency and operating systems taught by the first author. The course is given at the end of the second year of a three year bachelor program in computer science. It follows a course on operating systems theory, and is therefore focused on laboratory work in the educational operating system Pintos[3]. The course also introduces concurrency and synchronization (which is not taught in the previous course). An outline of the course is presented in the right side of Fig. 4.

To evaluate the tool, we invited students to (virtual) problem-solving sessions. In the invitation students were informed that their participation is voluntary and that they would remain anonymous. Out of the approximately 80 students taking the course, 12 students signed up for participating in the problem-solving sessions. These 12 students were then randomly assigned to one of two 105 minute sessions. In the end, only 3 students in the first session and 5 in the second session showed up.

Each session started with a 15 minute introduction where the first author briefly introduced the goal of the tasks, and where to hand in their solutions. Students were then given four tasks and 80 minutes to solve them. The problem-solving session ended with a 10 minute discussion of the solutions to each of the tasks. Each of the four tasks presented students with a piece of code containing some concurrency problem. A comment at the top of the file described the intended behavior of the code, and briefly outlined an observable problem with the incorrect implementation. The students were then asked to solve the concurrency issue by adding suitable synchronization. The tasks were designed to be solvable using only semaphores, as that was the only synchronization primitive introduced at the time of the problem-solving sessions. The tasks are presented in further detail below.

As the tasks were designed to be progressively more difficult, students were asked to solve the four problems one by one, in order. Furthermore, students were asked to hand in their solution to the current task before starting the next one. Students were able to ask for help with technical issues (e.g., not able to compile) during the sessions, but not with the tasks themselves. Students in the first session (group A) were asked to compile and run the given code using the supplied makefile in the terminal, similarly to what is done in the labs. Students in the second session (group B) were instead asked to run the code in Progvis. These students did not have the ability to use command-line tools (the makefile and other header files were not given to group B). At this point, both groups had seen Progvis used at the first lecture to visualize pointer arithmetics.

For each session, we recorded when each student handed in each of their solutions. We also graded the solutions. When grading, we first examined if the solution was correct (i.e., behaved according to the specifications and with no possible concurrency issues). For solutions that were not correct, we also recorded why the solutions were not correct.

After the problem-solving sessions, a modified version of Progvis was published on the course webpage. This modified version tracked each time it was started, which allows us to see to what extent the tool was used by students.

```c
int do_work(int param) {
    // Heavy work performed here...
    return param * param;
}
int param;
int result;

void thread_main(int *dummy) {
    result = do_work(param);
}
int main(void) {
    param = 100;
    thread_new(&thread_main, NULL);

    int here = do_work(50);
    printf("From other thread: %d\n", result);
    printf("From here: %d\n", here);
    printf("Sum: %d\n", result + here);
    return 0;
}
```

Listing 1: First task, comments omitted for brevity.

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

4.1 The Four Tasks

The first task, shown in Listing 1, attempts to execute the function `do_work` (which is assumed to take a long time to compute) in two threads concurrently. Due to a lack of synchronization, the result may be printed before it is computed by the spawned thread. The fact that the result was incorrect sometimes was noted in the description. Group A were also given a hint to insert a call to `sleep(param)` in the `do_work` function to better see the issue.

```c
 int do_work(int param) {
    // Heavy work performed here...
    return param * param;
}
struct work_data {
    int param;
    int result;
};
void thread_main(struct work_data *data) {
    data->result = do_work(data->param);
}
struct work_data *spawn(int param) {
    struct work_data *data
        = malloc(sizeof(struct work_data));
    data->param = param;
    thread_new(&thread_main, data);
    return data;
}
int wait(struct work_data *data) {
    int result = data->result;
    free(data);
    return result;
}
```

Listing 2: Second task, comments omitted for brevity.

The second task, shown in Listing 2, is described as a generalization of the first task. It implements the functions `spawn` and `wait`, which have semantics similar to `fork` and `wait` in POSIX. The `spawn` function spawns a thread, returning a data structure that can be passed to `wait` to retrieve the result when it is computed. A `main` function that spawns and waits for two threads was provided as an example along with a comment indicating that the `main` function should not need to be modified. Students in group A were encouraged to insert a sleep call here as well.

The third task, shown in Listing 3, implements a very simple ring buffer (similar to a POSIX pipe) containing only one element. The `buffer_put` function shall wait for the buffer to become empty and then insert an element. The `buffer_get` function shall wait for the buffer to contain an element and then remove and return that element. A `main` program with one producer thread and one consumer thread was given as well, once again that it should not need any modifications.

The final task, shown in Listing 4, implements a simple *future* object, which is essentially a generalization of the data structure in the second task. The difference is that it should be possible to call `future_get` multiple times for each future instance, while `wait` could be assumed to be called only once. A `main` function that created two future objects and used them to wait for two threads to complete was supplied with the customary note that it should not need any modifications.

```c
struct buffer {
    // Single value in the buffer.
    int value;
};
struct buffer *buffer_create(void) {
    struct buffer *buf
        = malloc(sizeof(struct buffer));
    buf->value = 0;
    return buf;
}
void buffer_destroy(struct buffer *buf) {
    free(buf);
}
void buffer_put(struct buffer *buf, int value) {
    buf->value = value;
}
int buffer_get(struct buffer *buf) {
    return buf->value;
}
```

Listing 3: Third task, comments omitted for brevity.

```c
struct future {
    // Result stored in the future.
    int result;
};
struct future *future_create(void) {
    struct future *f
        = malloc(sizeof(struct future));
    f->result = 0;
    return f;
}
void future_destroy(struct future *f) {
    free(f);
}
void future_post(struct future *f, int val) {
    f->result = value;
}
int future_get(struct future *f) {
    return f->result;
}
```

Listing 4: Fourth task, comments omitted for brevity.

5 RESULTS

Figure 3 shows the time taken for each of the students to solve each task. Students from group A are labeled A1, A2, and A3, and students from group B are labeled B1, B2, B3, B4 and B5. From the graph, we can see that none of the students in group A completed task 3, while four out of five in group B completed it. Furthermore, three out of five students in group B solved all tasks within the allotted time.

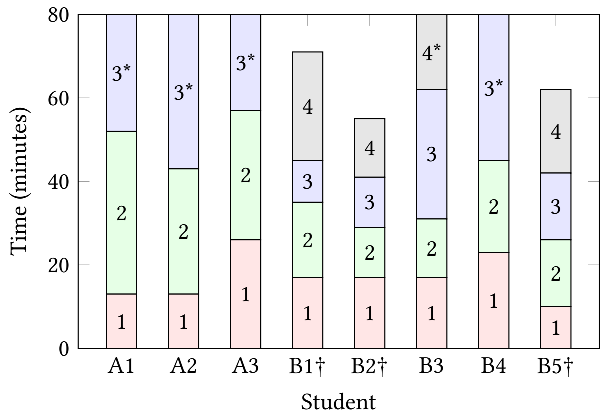


Figure 3: Time taken for each of the students to solve each of the four tasks out of the 80 minutes available. The number inside each bar indicate the task being worked on. Tasks marked with an asterisk were not submitted before the allotted time was up. Students marked with a dagger did not submit their solutions as they were completed, but when all of them were completed.

The times for individual tasks are slightly inaccurate for group B, since three students (B1, B2 and B5) did not submit each solution as soon as it was completed, but at the end when they were done with all tasks. For these students, the time distribution between the four tasks are based on self-assessments by the students. The total time taken is, however, available for these students and is used in Fig. 3.

Table 2 summarizes the correctness of students' solutions to the individual tasks. Each solution was marked as correct if it properly solved all issues in the code. Cases were the solution could be improved, or was incorrect, are noted using the numbers (1) to (9), which refer to the list below. Note that a correct solution in Table 2 might still not work as intended when dealing with multiple instances of the data structure in the task since the semaphore might have been declared globally, as shown in Table 3. Also note that task 1 from student A3 contained almost no modifications, and could therefore not be graded. The following problems were found in the solutions:

(1) Calling `sema_up` before the return statement in `do_work`, rather than when the return value is actually stored at its final location.
(2) Using the semaphore as a lock inside `do_work` (around the `sleep` statement) with no additional synchronization.
(3) Calling `sema_down` before performing computations that could be done in parallel (i.e., either calling `do_work` or starting additional threads). This does eliminate race conditions, but essentially executes the program in serial.
(4) Calling `sema_up` and `sema_down` twice in a row without affecting the behavior of the program in any way.
(5) Only calling `sema_down` when the thread should wait (i.e., when the buffer is known to be full or empty) while always calling `sema_up`. The semaphores will stay above zero and therefore never wait.

Table 2: Summary of correct (✓) and incorrect (✗) solutions. Dashes (-) indicate solutions incomplete enough to tell whether or not they are correct. Numbers in parentheses refer to the list in Section 5.

	A1	A2	A3	B1	B2	B3	B4	B5
Task 1	✗(1)	✗(2)	-	✓	✓	✓(3)	✗(1)	✓
Task 2	✓(4)	✗(2)	✓	✓	✓	✓(3)	✓	✓
Task 3	✗(5)	✗(6,7)	✗(6)	✓	✓	✗(8)	✗(8)	✗(8)
Task 4	-	-	-	✓	✗(9)	-	-	✓

Table 3: Summary of whether the semaphore was placed locally, inside the data structure, (L) or globally (G). Task 1 offered no alternative to global primitives and is therefore omitted. Submissions marked with an asterisk were possibly not complete as they were submitted at the end of the allotted time.

	A1	A2	A3	B1	B2	B3	B4	B5
Task 2	G	G	G	L	L	G	L	L
Task 3	G*	G*	L*	L	L	G	L*	L
Task 4	-	-	-	L	L	G*	-	L

(6) Incorrect initialization of a semaphore (e.g., -1 or a value that does not match how it is used).
(7) Calling `sema_up` before `sema_down` in `buffer_put` rather than the other way around.
(8) Using only one semaphore to wait for an empty buffer to become filled, forgetting to wait for a full buffer to become empty.
(9) Introduces data races in the *future* implementation by adding a variable that is not synchronized (not modeled by the tool).

As previously mentioned, Table 3 shows whether students declared the semaphore as a global variable or inside the data structure. As task 1 did not contain a data structure, it has been omitted from the table. In cases where students declared the semaphore globally, their solution will not work properly if multiple instances of the data structure is used concurrently, even if their solution is marked as correct in Table 2. From this table, a clear difference between the two groups is visible. Almost all of the students declared the semaphore at a global level in group A, while almost all declared it inside the data structure in group B.

Figure 4 contains the data collected from the logging in the modified version of the tool. The tool was published to students after the problem-solving session on day 11. The version containing logging was, however, not finished until day 17. Thus, there may be students who used the tool during these few days that are not in our data set. The unique identifier recorded by the logging allows differentiating between individual students, but does not allow identifying individuals. Furthermore, a new identifier is generated if the student removes the tool and downloads it again, so different users in Fig. 4 could be the same student. From this data, we can see that the tool was indeed used during the course, but not much. Additionally, one of the TAs in the course noted that one student pair had used the tool to solve a part of deadline 3 that involves

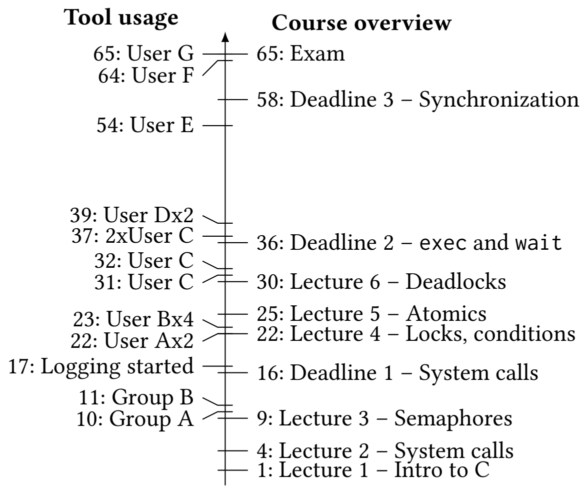

Tool usage
Course overview
65: User G
64: User F
54: User E
39: User Dx2
37: 2xUser C
32: User C
31: User C
23: User Bx4
22: User Ax2
17: Logging started
11: Group B
10: Group A
65: Exam
58: Deadline 3 – Synchronization
36: Deadline 2 – exec and wait
30: Lecture 6 – Deadlocks
25: Lecture 5 – Atomics
22: Lecture 4 – Locks, conditions
16: Deadline 1 – System calls
9: Lecture 3 – Semaphores
4: Lecture 2 – System calls
1: Lecture 1 – Intro to C

Figure 4: Timeline describing lectures and lab deadlines in the course (right), when the study was conducted (left) and when students started the tool (left). The number indicates number of days since the start of the course.

implementing a reader-writer lock. The students had created their own testbed and experimented with different implementations in order to solve the problem.

6 DISCUSSION

Based on the data presented in Section 5 we can see a number of notable differences between the two groups. First and foremost, Fig. 3 shows that three out of the five students in group B managed to solve all four tasks during the allotted time, while none of the three students in group A completed the third task. This is noteworthy on its own since the number of tasks was deliberately high to make it unlikely for students to run out tasks. This also gives a clear indication that the tool might help the students to solve the synchronization issues at hand, perhaps at least by helping students to find the problems in the original code, but maybe also in showing how the synchronization in the students' proposed solution avoid the problems.

Given that three out of the five students using Progvis did not submit their solutions as they were completed, it is difficult to observe trends in the times for individual tasks. By comparing the times for students B3 and B4 (who did submit their solutions as they were completed) to the times for students in group A, a small decrease in the time used for task 2 is observable, while the time used for task 1 might be slightly increased, perhaps since students were initially not familiar with Progvis. We can also see that the time distribution between tasks as reported by students B1, B2 and B5 seem to be in line with students B3 and B4, meaning that the self-reported times are likely not entirely unreliable. However, due to the limited data regarding the times for individual assignments, and the uncertainty of the data, we will not try to draw any definitive conclusions from this part of the data.

The trend that students in group B performed better than group A is also visible in Tables 2 and 3 when comparing solutions to the first two tasks (that were completed by both groups). While many solutions in group A did not solve the synchronization issues correctly, almost all students in group B managed to arrive at correct solutions. One notable synchronization error from group A that would perhaps be illustrated well by the tool is student A2's solution to tasks 1 and 2. The student attempted to use a semaphore as a lock to make sure that only one thread could execute `do_work` at the same time. However, the student failed to realize that the main thread needs to wait for the other threads in order for the program to work correctly. For task 3 (which students in group A did not finish) we can see that students in group A struggled with the semantics of the semaphore (initialization, when to call *up* and *down*), while students in group B supplied correct solutions except for forgetting to wait when the buffer was full. A similar trend is visible in Table 3, where most solutions from group A used a global semaphore while most solutions from group B used a semaphore declared inside the data structure. This gives a fairly clear indication that it is beneficial to clearly visualize instances of data structures when working with concurrency in order to help students realize the connection between synchronization primitives and the data they protect. This is one of the ways in which students experience critical sections found by a Phenomenographic study of concurrency [19].

As all of these results indicate that group B, who used the tool, performed better than group A, we feel positive that using the tool to learn synchronization is beneficial, at least in these early stages of learning concurrency. It is, however, not possible to attempt to quantify the benefits, or to draw definitive conclusions due to the small data set. The results could also be skewed by the fact that the session for group B was held one day after the session for group A. There were, however, no difference in the number of scheduled activities. Both groups had one laboratory session right after the lecture that introduced semaphores and nothing more.

Some students did use the tool outside of the problem-solving sessions, as shown by Fig. 4, but at most seven student pairs (labs are done in pairs). This rather low usage was in spite of the tool being used during lectures to illustrate concepts like pointers, synchronization and deadlocks, and making the source code from the lectures available to students. As the tool only records whenever it is started (in order to not be too invasive on privacy), it is not possible to tell for how long the tool was used for each startup. In the unlikely extreme, this might mean that students left the tool running for days at a time, thus "hiding" their activity. Another possible reason for the low usage is that the logging was not completed when the tool was first published, and thus any initial spikes of interest would have been missed, and some students might have kept using the old version without logging throughout the course. The latter is unlikely as the tool was improved during the course, meaning that some latter examples did not work in the old version of the tool. Based on previous experience with the course and the students who take the course, the likely explanation for the low usage is that students need to spend much of their time working on the lab assignments, and therefore feel like time not spent working on the lab assignments is wasted. Thus, since it is not possible to visualize parts of the large codebase of Pintos directly in Progvis, they feel like the upfront cost of creating a small example that is possible to visualize is not worth the gains. This likely also applies to spending time exploring the concepts by solving one of the numerous examples available on the course webpage. Based on experience from other courses with the same group of students, it is usually necessary to provide some (small) incentive for students to explore material that is perceived as "not required for passing", even if it would help students learn. We plan to further explore this possibility in the future.

As previously mentioned, one of the TAs noted that one pair of students did find the tool useful when solving one of the lab assignments. This indicates that at least some students see the benefit of using the tool, and that one possible way of increasing the usage of the tool is to provide small examples where students are able to prototype their solutions before incorporating them into the Pintos codebase. Since it is not possible to directly run code from the labs in the tool (it is not built to visualize an entire operating system, albeit a small one) the effort required to use the tool might not be considered worth the benefits. The situation could most likely be remedied by providing some pre-made testbeds for testing parts of the labs that might benefit from using the tool.

6.1 Future Work

As previously mentioned, this small pilot study of Progvis is only able to show that using the tool for this kind of exercises is promising, and at the very least does not make students perform worse than without the tool. This is a good starting point for further evaluation of the tool in a more comprehensive study in the future. Ideally, such a study would attempt to find whether the tool help students learn concurrency, or if its value is that it helps students uncover concurrency issues. This could, for example, be done by further incorporating the tool in a course and comparing scores on the final exam.

In order to motivate students to utilize the tool, there are many areas that can be explored. One option is to integrate the tool in the lab assignments, which would show the benefits to students at an early stage and hopefully encourage using the tool later on in the course as well. As previously mentioned, providing skeletons for parts of the lab assignments where students are known to struggle in order to lower the effort required to use the tool is also worth investigating.

Another venue worth exploring to motivate students to explore concurrency using Progvis is to add optional exercises with gamification elements. For example, awarding points to students who solve problems. Perhaps also awarding extra credits for collecting a specified amount of points. This is not entirely trivial depending on how the exercises are designed, however, as automatic grading of concurrent programs is a difficult problem in the general case.

Finally, as previously mentioned, we aim to extend the tool to better illustrate the intricacies of the C memory model. Currently, the tool emulates a strict memory model (sequential consistency), which might lead students to believe that this memory model holds in general. We have already started this work by reporting conflicting reads and writes for single statements that are executed concurrently. While this is closer to the true semantics, it is still not entirely accurate.

7 CONCLUSION

In this paper we have presented the visualization tool Progvis, aimed at helping students learn concurrency and synchronization, as well as results from a pilot study in a course setting. In addition to concurrency concepts, the tool makes an effort to provide a clear model of the data manipulated in the system to help students identify shared data and associate the data with appropriate synchronization primitives. The focus on the data also allows the tool to be used to clarify other known difficult concepts in earlier courses as well. For example, pointers and references, the difference between a class and an instance, and pointer arithmetics. This versatility is further aided by the tool being designed to allow visualizing multiple languages, with support for Java currently planned.

Since only eight students participated in the pilot study, it is difficult to draw definitive conclusions. The results from the limited data is, however, promising. The data shows that students who used the tool did not perform worse than students using only command-line tools in terms of the number of problems solved and the time required to do so. Three out of the five students who used the tool solved all tasks well within the allotted time, while none of the three students without the tool managed to do so. The solutions produced by the students using the tool were also better than the students not using the tool. This is an indication that the tool does indeed aid in some part of the problem-solving process. We are, however, not able to tell if this helps students learn concurrency, as we have not studied whether using the tool while learning improves performance in other settings at later stages.

REFERENCES

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

[2] Chris Exton. 2000. Elucidate: A Tool to Aid Comprehension of Concurrent Object Oriented Execution. *SIGCSE Bull.* 32, 3 (July 2000), 33–36. https://doi.org/10.1145/353519.343066

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

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

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

[6] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[7] Aubrey Lawson and Eileen T. Kraemer. 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* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 82–92. https://doi.org/10.1145/3377814.3381706

[8] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A Uronen. 2003. The Jeliot 2000 program animation system. *Computers & Education* 40, 1 (2003), 1–15. https://doi.org/10.1016/S0360-1315(02)00076-3

[9] Jan Lönnberg. 2012. *Understanding and Debugging Concurrent Programs through Visualisation.* G5 Artikkeliväitöskirja. http://urn.fi/URN:ISBN:978-952-60-4530-6

[10] Jan Lönnberg and Anders Berglund. 2007. Students' Understandings of Concurrent Programming. In *Proceedings of the Seventh Baltic Sea Conference on*

Computing Education Research - Volume 88 (Koli National Park, Finland) *(Koli Calling '07)*. Australian Computer Society, Inc., Darlinghurst, Australia, 77–86. http://dl.acm.org/citation.cfm?id=2449323.2449332

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

[12] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. 2002. Exploring the Role of Visualization and Engagement in Computer Science Education. *SIGCSE Bull.* 35, 2 (June 2002), 131–152. https://doi.org/10.1145/782941.782998

[13] Anna Offenwanger and Yves Lucet. 2014. ConEE: An Exhaustive Testing Tool to Support Learning Concurrent Programming Synchronization Challenges. In *Proceedings of the Western Canadian Conference on Computing Education* (Richmond, BC, Canada). Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/2597959.2597972

[14] Jorma Sajaniemi and Marja Kuittinen. 2003. Program Animation Based on the Roles of Variables. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (San Diego, California) *(SoftVis '03)*. ACM, New York, NY, USA. https://doi.org/10.1145/774833.774835

[15] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. 2011. OpenDSA: Beginning a Community active-eBook Project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '11)*. ACM, New York, NY, USA, 112–117. https://doi.org/10.1145/2094131.2094154

[16] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education.* Ph.D. Dissertation. Aalto University, Helsinki, Finland. http://lib.tkk.fi/Diss/2012/isbn9789526046266/

[17] Filip Strömbäck. 2018. Storm: A Language Platform for Interacting and Extensible Languages (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. Association for Computing Machinery, New York, NY, USA, 60–64. https://doi.org/10.1145/3276604.3276982

[18] Filip Strömbäck, Linda Mannila, Mikael Asplund, and Mariam Kamkar. 2019. A Student's View of Concurrency - A Study of Common Mistakes in Introductory Courses on Concurrency. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. ACM, New York, NY, USA, 229–237. https://doi.org/10.1145/3291279.3339415

[19] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2020. Exploring Students' Understanding of Concurrency - A Phenomenographic Study. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 940–946. https://doi.org/10.1145/3328778.3366856

[20] Sander Valstar, William G. Griswold, and Leo Porter. 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* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 794–800. https://doi.org/10.1145/3287324.3287419