Visualizing Classic Synchronization Problems

Dining Philosophers, Producers-Consumers, and Readers-Writers

Joel C. Adams
Department of Computer Science
Calvin College
Grand Rapids, MI USA 49546
adams@calvin.edu

Elizabeth R. Koning
Department of Computer Science
Calvin College
Grand Rapids, MI USA 49546
erk24@students.calvin.edu

Christiaan D. Hazlett
Department of Computer Science
University of Illinois, Urbana-Champaign
Champaign, IL USA
christiaanhazlett@gmail.com

ABSTRACT

Classic synchronization problems are often used to introduce students to the subtleties of concurrency and synchronization mechanisms, such as semaphores, monitors, locks, and condition variables. The Dining Philosophers, Producers-Consumers, and Readers-Writers are all classic problems in which a correct solution requires the actions of multiple processes or threads to be synchronized. In this paper, we present visualizations for these three problems and describe their use as pedagogical tools to help students build accurate mental models of concurrency abstractions such as starvation, deadlock, livelock, and correct execution. We also present the results of an experiment that indicate students find using these visualizations to be significantly more engaging than reading a textbook, with no significant difference in learning. We do not claim that our visualizations should replace a course text; rather we present them as engaging pedagogical tools to complement the textbook in courses on *Operating Systems*, *Programming Languages*, and other courses where concurrency and synchronization are covered.

CCS CONCEPTS

• Computing methodologies~Shared memory algorithms • Social and professional topics~Computer science education • Software and its engineering~Synchronization

KEYWORDS

Dining Philosophers; Graphics; Parallel; Producers; Consumers; Readers; Writers; Synchronization; Threads; Visualization

ACM Reference format:

Joel Adams, Elizabeth Koning and Christiaan Hazlett. 2018. Visualizing Classic Synchronization Problems: Dining Philosophers, Producers-Consumers, and Readers-Writers. In *Proceedings of the 2019 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'19). ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3287324.3287467

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287467

1 INTRODUCTION

Classic synchronization problems, such as the Cigarette Smokers, the Dining Philosophers, the Producers-Consumers (also known as the Bounded Buffer problem), the Readers-Writers, and the Sleeping Barber Problems are commonly used to introduce CS students to the subtleties of concurrency.

Operating Systems textbooks (e.g., [14-16]) commonly cover a subset of these problems and use solutions to these problems to illustrate the use of semaphores, locks, condition variables, monitors, and related mutual exclusion and synchronization mechanisms. For example, [14] uses the Dining Philosophers, Producers-Consumers, and Readers-Writers Problems to illustrate the use of semaphores; and revisits the Dining Philosophers Problem to illustrate the use of a monitor.

Programming Languages textbooks may also use these problems to illustrate programming languages' concurrency features. For example, [9] uses the Producers-Consumers Problem to illustrate semaphores and monitors; [13] uses the same problem to explain Ada's Task-rendezvous mechanism, Java's semaphores, and Java's monitor mechanism.

Some of these problems involve real-world entities (cigarette smokers, philosophers, barbers) and have a concrete, non-abstract scenario that may be relatively easy for students to understand. Other problems deal with more abstract entities (producers, consumers, readers, writers) that may make it more challenging for students to build accurate mental models. To help students understand these problems, traditional textbooks often include figures, but static figures and diagrams are necessarily limited in their abilities to depict the dynamic interactions of the entities. These limitations stem from publisher economics and the static nature of the paper on which such books are printed. By contrast, software is a more flexible medium that can be used to create dynamic, interactive visualizations of these problems.

In this paper, we present software visualizations of the Dining Philosophers, Producers-Consumers, and Readers-Writers problems and explore their use. We also present the results of an assessment activity whose results indicate (i) that students learn as much from interacting with these visualizations as they do from reading a textbook, and (ii) that students find the visualizations to be significantly more engaging than a textbook.

2 RELATED WORK

Our project adds to an extensive literature on visualization. In this section, we present a very brief overview of that literature.

In recognition of the limitations of paper-technology textbooks at communicating how an algorithm solves its problem, computer science (CS) educators have created a wide variety of algorithmic visualizations over the years. For example, [4, 7, 11, 12] present dynamic visualizations of various sequential algorithms and report that such visualizations help students understand those algorithms. [16] reports that students who explore a visualization *interactively* learn significantly more than those who view it passively. Our project differs from these in its focus on concurrent algorithms.

CS educators also have a long history of looking for ways to help students visualize concurrent and/or parallel algorithms. For example, [17] presents a visualization of the Readers-Writers Problem, using specialized hardware and a vintage-1988 Modula-like language called Portal. [8] provides an overview of efforts in this area, circa 1993. [6] presents a visualization of the Dining Philosophers Problem using a 1994 sequential animation library called XTANGO and the SR programming language, with each philosopher represented as an SR process. Likewise, [10] presents three parallel algorithm visualizations using a 1999 parallel simulator called MultiPascal and a visualization tool called JSAMBA. These projects generally use a "post-mortem" approach: a concurrent program's behavior is traced during its execution (e.g. saved to a file), that trace is then used to drive the visualization after the program has terminated. Our project differs from these in (i) providing a visualization of the concurrent program's behavior in real-time, during program execution; (ii) using multithreading instead of multiprocessing; and (iii) using an object-oriented language (C++) and approach.

A project whose aims align with ours is ThreadMentor [2, 3], which provides a C++/GTK class library of GUI widgets for real-time visualization of specific multithreading synchronization constructs—threads, semaphores, monitors, locks, etc.—and demonstrates their use using the Cigarette Smokers, Dining Philosophers, and Readers-Writers Problems. However, the current version of GTK is not thread-safe [5]; neither [2] nor [3] describes the architecture in sufficient detail to indicate how ThreadMentor resolves this issue. Our project differs from ThreadMentor by (i) using the Thread Safe Graphics Library (TSGL) [1] instead of GTK; (ii) using TSGL to create real-time graphical visualizations of the actual concurrent entities—philosophers, producers, consumers, readers, writers, etc. (vs. ThreadMentor's lower-level synchronization primitives)—and how they interact to solve a given problem; and (iii) including the results of an assessment exercise that indicates students find interacting with these visualizations to be more engaging than reading a textbook. We chose TSGL because it is a native C++ (in which parallel applications are commonly written), thread-safe, general library for real-time graphical drawing. It also provides platform independence by being built atop OpenGL. [1] presents results of an experiment in which the use of a TSGL visualization improved student understanding of the parallel loop construct.

3 THE VISUALIZATIONS

The Dining Philosophers, Producers-Consumers, and Readers-Writers Problems seem to be the "classic" problems that are most commonly-used by textbook authors. In this section, we present our visualizations for these three problems.

3.1 The Dining Philosophers Problem

In the Dining Philosophers Problem, $N > 1$ philosophers sit around a circular table. Each philosopher goes through an endless cycle of thinking, getting hungry, eating, thinking, getting hungry, eating, and so on. Between each pair of philosophers is a single fork, and a philosopher must acquire both forks in order to eat. Since each philosopher is a perfect logician, the problem is to devise a single protocol for each philosopher to follow in sharing the forks, that will ensure no one starves.

Our visualization of this problem can be run from the command-line as follows:

```
$ ./DiningPhilosophers [N] [speed] [protocol]
```

If no command-line arguments are specified, parameter *N* defaults to 5 philosophers, the *speed* defaults to 5 (on a scale of 1-60), and the *protocol* defaults to odd-even check (see Section 3.1.3 below). Figure 1 shows our visualization at the outset:

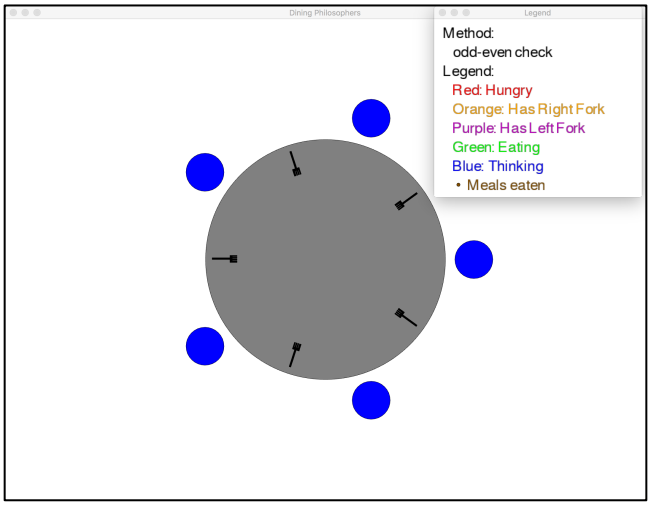

Figure 1: The Dining Philosophers, Initial State

As can be seen, our visualization draws a circular gray table with *N* equidistant forks around it. A philosopher, represented by a circle that is color-coded to indicate her current state, sits between each pair of forks. A separate legend-window indicates the state corresponding to each color; each philosopher begins in the blue 'thinking' state.

3.1.1 Deadlock. A deadlock occurs if each process or thread in a group is holding a resource that is needed by another member of the group, in a circular-waiting pattern. Operating Systems textbooks frequently use the Dining Philosophers problem to illustrate the topic of deadlock. For example, [14-16] all illustrate deadlock by presenting a "first attempt" Dining Philosophers protocol like the following for each philosopher to follow:

```
Semaphore fork[N];
right = getID();
left = (right+1) % N;
while (true) {
  /* think for a while */
  fork[right].acquire();
  fork[left].acquire();
  /* eat for a while */
  fork[right].release();
  fork[left].release();
}
```

This protocol has a philosopher pick up her right fork when it is available and then pick up her left fork when it becomes available. If all philosophers happen to get hungry at the same time, a deadlock ensues. Our visualization uses this protocol if 'w' (for wait-when-blocked) is given as the command-line *protocol*:

```
$ ./DiningPhilosophers 5 5 w
```

Since each philosopher starts out by thinking for a random length of time, the protocol will seem to work correctly for a while, but eventually the philosophers deadlock, as shown in Figure 2:

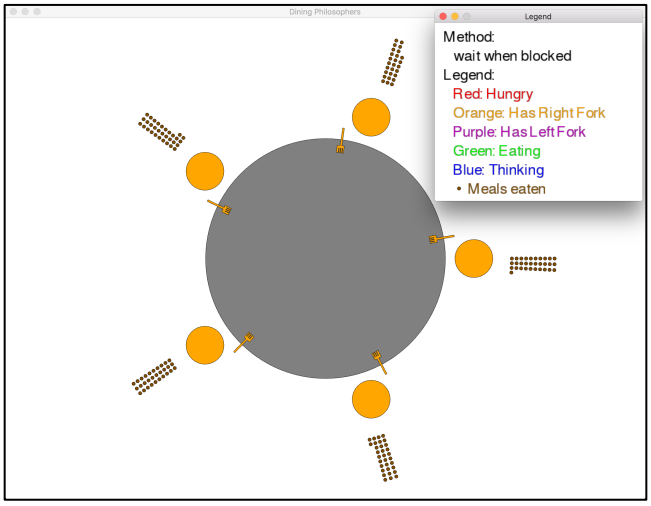

Figure 2: The Dining Philosophers, Deadlocked

Each brown dot "behind" a philosopher represents a meal she has eaten. The number of these dots varies because a philosopher thinks and eats for random lengths of time. When these dots stop appearing behind a hungry philosopher, she begins to starve.

3.1.2 Livelock. In an attempt to fix the deadlock problem, a protocol like the one below might be presented to students:

```
/* same variables as before */
while (true) {
  /* think for a while */
  fork[right].acquire();
  do {
    if (fork[left] is in use) fork[right].release();
    else fork[left].acquire();
  } while (I have less then 2 forks);
  /* eat for a while */
  fork[right].release();
  fork[left].release();
}
```

Seeking to avoid a deadlock, this protocol has a philosopher pick up her right fork when it is available, but if her left fork is not available, put down her right fork. While this does avoid a deadlock, it can unfortunately lead to *livelock*: if all of the philosophers become hungry at the same time, they will all pick up their left forks, all put down their left forks, all pick up their left forks, all put down their left forks, ... forever. Since no philosopher gets to eat, a livelock also leads to starvation. Our visualization uses this protocol if 'f' (for forfeit when blocked) is given as the command-line *protocol*:

```
$ ./DiningPhilosophers 5 5 f
```

Using this protocol, our philosophers will endlessly oscillate between the "hungry" and "has right fork" states. Figure 3 shows the two views this protocol will alternately produce:

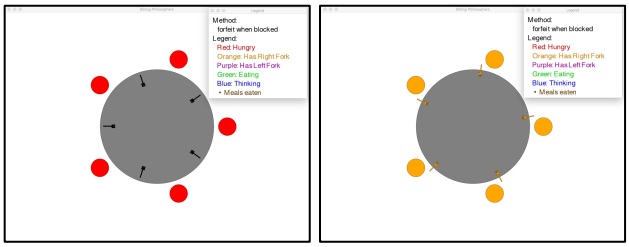

Figure 3: The Dining Philosophers, Livelocked

In Figure 3, the philosophers following this protocol endlessly cycle between the two states. Our visualization thus lets students see and experience the difference between livelock and deadlock.

3.1.3 A Correct Solution. One way to solve the Dining Philosophers Problem is to use a protocol like the one below:

```
/* same as before */
id = getID();
while (true) {
  /* think for a while */
  if ( odd(id) ) {
       fork[right].acquire();
       fork[left].acquire();
  } else {
       fork[left].acquire();
       fork[right].acquire();
  }
  /* eat for a while */
  fork[right].release();
  fork[left].release();
}
```

By having odd-numbered philosophers pick up their forks in a right-then-left ordering and even-numbered philosophers pick them up in a left-then-right ordering, this protocol prevents any circular-wait from developing, and thus avoids the deadlock issue. Since a philosopher only releases her fork after she is finished eating, it also avoids the livelock issue. Our visualization uses this protocol by default, if no command-line protocol is given:

```
$ ./DiningPhilosophers 7
```

Figure 4 shows our visualization using 7 philosophers after it has run for a few minutes:

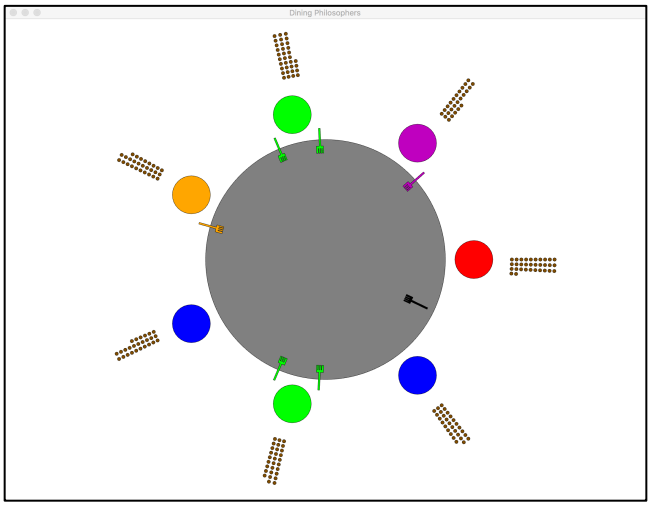

Figure 4: The Dining Philosophers, Working Correctly

At the moment captured in Figure 4, philosopher 0 is at the "one o-clock position", is even-numbered, is hungry, and has acquired her left fork but cannot yet acquire her right fork; her left neighbor (1) is odd-numbered, hungry, and cannot yet acquire her right fork because 0 has it; 1's left neighbor (2) is thinking; 2's left neighbor (3) has acquired both forks and is eating; 3's left neighbor (4) is thinking; 4's left neighbor (5) is odd-numbered, has acquired her right fork, but cannot yet acquire her left fork because it is in use; 5's left-neighbor (6) is even-numbered, has acquired both forks, and is eating. The brown "meals eaten" dots behind each philosopher provide a visual indicator that no one is starving. As before, different philosophers have eaten different numbers of meals because each thinks and eats for slightly different random lengths of time.

Our Dining Philosophers visualization thus allows a student to specify the number of philosophers, the speed of execution, the philosophers' protocol, and to view the resulting behavior. Students can also view/modify the program's source code, letting them explore the protocol-code that produces a given behavior.

3.2 The Producers-Consumers Problem

In the Producers-Consumers Problem, $M > 0$ producer entities produce items and $N > 0$ consumer entities consume the items produced by the producers, ideally in the order they were produced. The problem is to come up with a scheme ensuring that:

- Each item produced by a producer is consumed by some consumer (i.e., no item gets lost).
- No item produced by a producer is consumed by more than one consumer (i.e., no item gets duplicated).
- No producer or consumer waits forever (i.e., no starvation).

The usual approach to solving this problem is to introduce an intermediate FIFO data structure called a *buffer*, into which the producers deposit their items, and from which the consumers retrieve the items. When this buffer has a fixed capacity, the problem is often called the *Bounded Buffer Problem*.

Textbooks often use this problem to introduce the topic of monitors, since building the buffer as a monitor provides the conditions needed to solve the problem. (A monitor has self-synchronizing methods that use a lock to ensure mutually exclusive access and condition variables for synchronization.) Given a shared buffer implemented as a monitor, the producer and consumer entities can be easily coded as follows:

```
/* declaration of shared buffer */

void producer() {
  while (true) {
    /* produce item it */
    buffer.put(it);
  }
}

void consumer() {
  while (true) {
    it = buffer.get();
    /* consume it */
  }
}
```

Our visualization can be run from the command-line as follows:

```
$ ./ProducerConsumer [M] [N]
```

where M and N are the numbers of producers and consumers, respectively. M and N both default to 5, as shown in Figure 5:

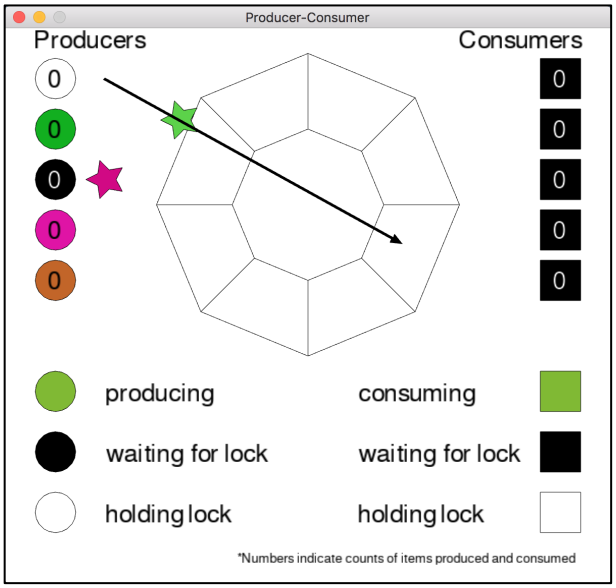

Figure 5: Producers-Consumers, Initial View

On the left is a column of producers, represented by circles; on the right, a column of consumers, shown as squares. The numbers inside a producer or consumer indicates the number of items it has produced and deposited, or has retrieved and consumed, respectively. In the middle is a circular buffer with room for 8 items. An item is represented by a star that appears next to its producer. In Figure 5, the topmost producer (0) has produced an item, has acquired the buffer's lock, and is depositing its item into the buffer; producers 1, 3, and 4 are in the process of producing items, producer 2 has produced an item and is waiting to acquire the buffer's lock. Since the buffer is empty, all of the consumers are also waiting to acquire the buffer's lock.

Note this visualization's color-coding: an entity waiting for the lock is colored black, one that has acquired the lock is colored white, and an item is colored with the color of its producer.

Figure 6 shows the same visualization later in the run:

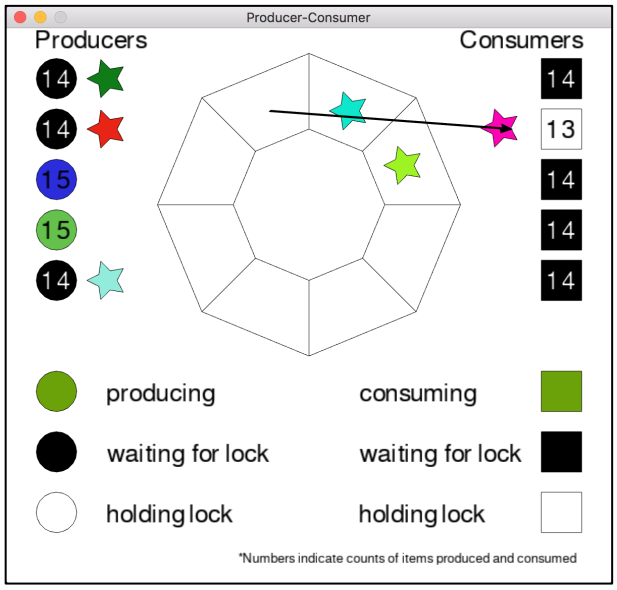

Figure 6: Producers-Consumers, Later View

In Figure 6, producers 0, 1, and 4 have all produced items and are awaiting their chance to deposit them into the buffer; producers 2 and 3 are in the process of producing items. Consumer 1 is retrieving an item from the buffer, which contains two other items; consumers 0, 2, 3, and 4 are all waiting to get items from the buffer but must wait (along with the producers) until consumer 1 releases the buffer's lock. Note that the production of items is balanced among the producers and the consumption of items is balanced among the consumers. As before, students can access the source to see the code that is driving the visualization.

By seeing that only one entity can ever access the buffer at a time, a visualization can help students build accurate mental models of a monitor and its mutually exclusive behavior.

3.3 The Readers-Writers Problem

In the Readers-Writers Problem, $M > 0$ writer entities write items to a shared database, and $N > 0$ reader entities read items from that database. While writers must write in a mutually exclusive fashion, all readers can read simultaneously because reading does not alter the database. The problem is to devise such a system and ensure that no reader or writer starves. The Producers-Consumers Problem would be similar to this problem, if multiple consumers could retrieve items from the buffer simultaneously.

Since a write must be performed mutually exclusively but a read need not, a key question is: When a writer is finished writing and both readers and writers are waiting, who gets to proceed? There are at least two ways to answer this question:

- In the *writer-priority protocol*, a waiting writer gets to proceed before any waiting readers.
- In the *reader-priority protocol*, waiting readers get to proceed before any waiting writers.

Textbooks [14, 16] present solutions using the reader-priority protocol; [15] presents solutions using both protocols.

Our visualization for this problem supports both protocols; it can be invoked as follows:

```
$ ./ReadersWriters [M] [N] [protocol]
```

where M is the number of writers, N is the number of readers, and *protocol* can be 'r' for reader-priority, or 'w' for writer-priority (which is the default). If omitted, both M and N default to 6. Figure 7 shows this visualization using all default values:

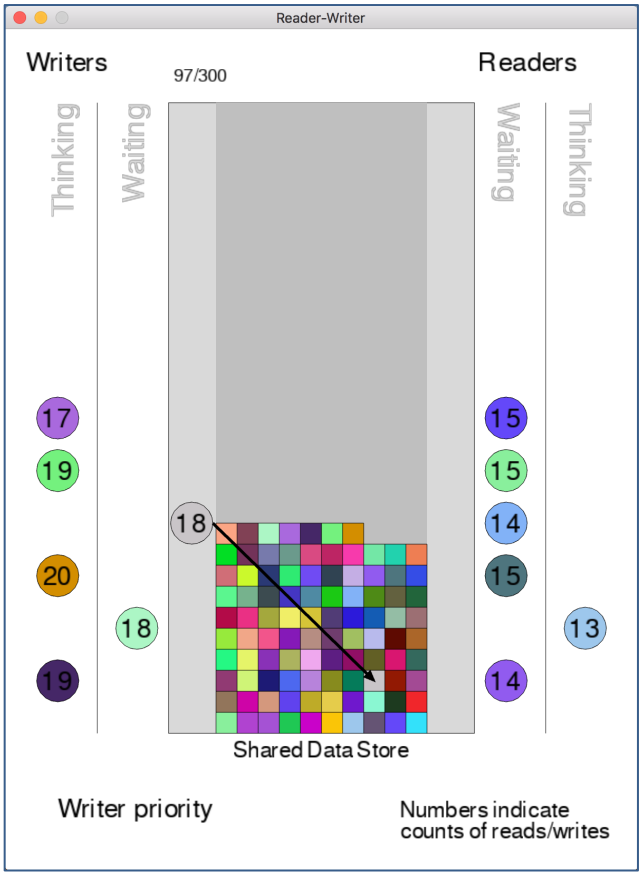

Figure 7: Readers-Writers, Early View

Readers and writers may be in any of three states: "thinking" about their next action, "waiting" to perform their next action, or performing their action. Our visualization provides six columns—one for each kind of entity's three states—plus a seventh central column for the database. This visualization thus uses spatial-positioning to show an entity's state, rather than color-coding.

When it writes, a writer may either create a new database entry or update an existing entry. In Figure 7, the writers have filled roughly a third of the database with new entries. Writer 2 has entered the database and is updating an entry; writers 0, 1, 3, and 5 are all "thinking" about their next write; writer 4 is ready and waiting to write; readers 0-3 and reader 5 are all waiting to read; and reader 4 is "thinking" about the last item it read. Since this is the writer-priority version, writer 4 will get to go next.

Figure 8 shows this same run a bit later in the execution:

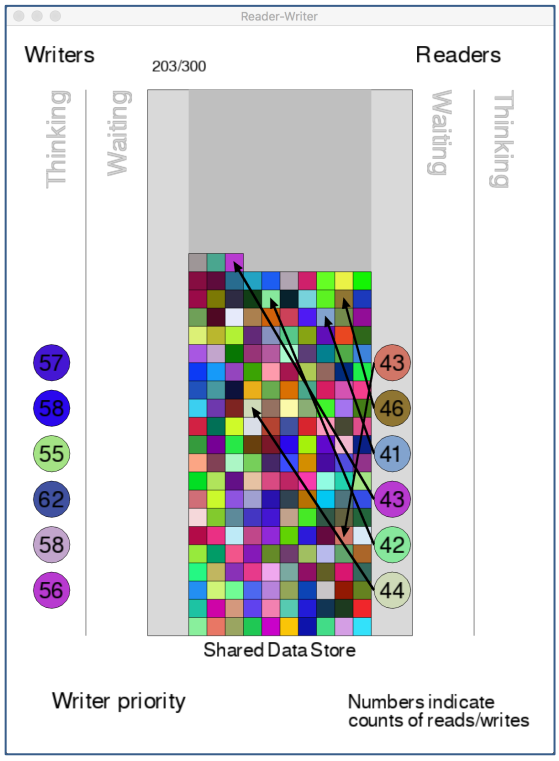

Figure 8: Readers-Writers, Later View

Unlike a writer, all of the readers can access the database and read different items simultaneously, as Figure 8 shows. By seeing the behaviors and the source code (which they may modify), students can see how these behaviors are produced. Space limitations prevent us from showing the reader-priority protocol, but it lets students see and experience how that protocol's behavior differs.

4 ASSESSMENT

To evaluate this work, we identified two research questions:

RQ1: Does interacting with our visualizations improve long-term recall of these problems, compared to reading a textbook?

RQ2: Do students find our visualizations to be a more engaging way to explore these problems, compared to reading a textbook?

To answer these questions, we devised the following experiment:

1. 34 volunteer subjects were recruited from our CS2 course (*Data Structures*, to ensure students had not seen these topics before), using extra course-credit as an incentive.
2. The subjects were randomly divided between a control group and a treatment group.
3. Simultaneously, and in identical computer labs:
   a. Both groups watched a short (~15 minute) video that introduced multithreading, concurrency, synchronization, and the Dining Philosophers Problem.
   b. For 10 minutes, the control group studied a textbook's discussion of that problem and its solutions, while the treatment group interacted with our visualization for it.
   c. Both groups watched a short (~10 minute) video that introduced the Producers-Consumers Problem.
   d. For 10 minutes, the control group studied a textbook's discussion of that problem and its solutions, while the treatment group interacted with our visualization for it.
   e. Both groups watched a short (~10 minute) video that introduced the Readers-Writers Problem.
   f. For 10 minutes, the control group studied a textbook's discussion of that problem and its solutions, while the treatment group interacted with our visualization for it.
4. Two weeks later, all subjects were emailed a link to a 14-question online quiz. Twelve questions covered details of the three classic problems; one question covered race conditions; the final question had subjects rate how engaging they found the post-video activities on a 1-5 scale.

22 of our volunteers showed up for the experiment and completed the quiz, so our sample size was limited. Our null hypothesis for RQ1 was that there would be no significant difference in quiz performance between the control and treatment groups. The treatment group's mean score was 9.9 compared to 10.5 for the control group; this difference was not significant ($p=0.3387$), so we were unable to reject the null hypothesis for RQ1. The control group outperforming the treatment group stems from our use of randomized groups: our control group's median CS2 final exam score was 84.5 compared to 80.5 for our treatment group.

For RQ2, our null hypothesis was that there would be no significant difference in the two groups' engagement-ratings of the post-video activities. The treatment group's mean rating was 4.2 compared to 2.8 for the control group; this difference was significant ($p=0.0090$), so we reject the null hypothesis for RQ2.

Thus, we found no significant difference in learning between our textbook-readers and our visualization-users, but the latter group measured their experience as significantly more engaging, despite our small sample size. The similarity of the two groups in learning is interesting, given that the students in our treatment group were measurably weaker than those in our control group.

Our experimental materials (videos, quizzes, visualizations, etc.) are all available upon request.

5 CONCLUSIONS

We have presented interactive visualizations for three "classic" synchronization problems: The Dining Philosophers, Producers-Consumers, and Readers-Writers Problems. By controlling the number of concurrent entities and the protocol, students can *see* abstractions like deadlock, livelock, and starvation in real-time, as well as the behaviors of correct solutions to these problems.

We have also presented experimental results that indicate: (i) students using our visualizations achieved levels of learning that were statistically similar to students reading textbook materials, and (ii) students found our visualizations to be significantly more engaging than traditional textbook treatments of the same topics. By letting students see and dynamically experience concurrent protocols in action, our visualization tools can supplement a course textbook by "bringing to life" the text's static figures and descriptions, and thus help students build more accurate mental models of concurrency abstractions.

Our thanks to NSF (DUE#1225739) and to the Calvin College Science Division, whose support made this work possible.

REFERENCES

[1] J. Adams, P. Crain, C. Dilley, S. Nelesen, J. Unger, M. Vander Stel, Seeing Is Believing: Helping Students Visualize Multithreaded Behavior, *Proc. of 47th ACM Technical Symposium on Computer Science Education* (SIGCSE'16), March 2016, 473-478. DOI= 10.1145/2839509.2844557.

[2] M. Beady, S. Carr, X. Huang, C.K. Shene, A Visualization System for Multithreaded Programming, *Proc. of 31st SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'00), March 2000, 1-5.

[3] S. Carr, J. Mayo, and C.K. Shene, ThreadMentor: a Pedagogical Tool for Multithreaded Programming, Journal on Educational Resources in Computing (JERIC), 3(1), March 2003, Article 1. DOI= 10.1145/958795.958796.

[4] E. Fouh, M. Akbar, C. Shaffer. The Role of Visualization in Computer Science Education. *Computers in the Schools: Interdisciplinary Journal of Practice, Theory, and Applied Research*, Vol. 29, 2012, 95-117. DOI=10.1080/07380569.2012.651422.

[5] Gnome Developer, GNOME Threads, Online, Accessed 2018-08-04: https://developer.gnome.org/gdk3/stable/gdk3-Threads.html.

[6] S. Hartley, Animating operating systems algorithms with XTANGO, *Proc. of 25th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'94), March 1994, 344-348.

[7] C. Kehoe, J. Stasko, A. Taylor. Rethinking the evaluation of algorithm animations as learning aids. *International Journal of Human Computer Studies*, (54)2, Feb 2001, p. 265-284. DOI=10.1006/ijhc.2000.0409.

[8] E. Kraemer and J. Stasko, The Visualization of Parallel Systems: an overview, *Journal of Parallel and Distributed Computing*, 18(2), June 1993, p. 105-117.

[9] K. Louden and K. Lambert, *Programming Languages: Principles and Practice* (3e), Cengage, 2011.

[10] T. Naps and E. Chan, Using Visualization to Teach Parallel Algorithms, *Proc. of 30th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'99), March 1999, 232-236.

[11] S. Roger, Integrating Animations into Courses, *Proc. of 1st Conference on Integrating Technology into Computer Science Education* (ITiCSE'96), June 1996, 72-74.

[12] C. Shaffer, M. Cooper, A. Alon, M. Akbar, M. Stewart, S. Ponce, S. Edwards. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3), August 2010. Article no. 9. DOI=10.1145/1821996.1821997.

[13] R. Sebasta, *Concepts of Programming Languages* (10e), Pearson, 2012.

[14] A. Silberschatz, P. Galvin, and G. Gagne, *Operating Systems Concepts* (9e), Wylie, 2012.

[15] W. Stallings, *Operating Systems: Internals and Design Principles* (8e), Pearson, 2014.

[16] A. Tanenbaum, H. Bos, *Modern Operating Systems* (4e), Pearson, 2015.

[17] M. Zimmerman, F. Perrenoud, and A. Schipper, Understanding Concurrent Programming Through Program Animation, *Proc. of 19th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE'88), Feb 1988, 27-31.