Developing a Pre- and Post-Course Concept Inventory to Gauge Operating Systems Learning

Kevin C. Webb
Computer Science Department
Swarthmore College
Swarthmore, PA
kwebb@cs.swarthmore.edu

Cynthia Taylor
Computer Science Department
Oberlin College
Oberlin, OH
ctaylor@oberlin.edu

ABSTRACT

Operating systems courses often present students with multiple approaches to solve a problem, often with differing trade-offs. While students are more than capable of memorizing the details of these competing approaches, they often struggle to recommend a specific approach and analyze its implications. In particular, we find that students exhibit difficultly in interpreting text-based scenario descriptions in a way that allows them to correctly choose between potential solutions when presented with a high-level, conceptual scenario.

In this paper, we describe the development of a pre- and post-course concept inventory, which we utilize to explore students' misconceptions of operating systems and their associated trade-offs. We compare the results of our assessment with in-class peer instruction questions and exam questions to characterize the areas in which students most commonly struggle with operating systems material.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

Human Factors

Keywords

Concept Inventory, Misconceptions, Operating Systems

1. INTRODUCTION AND BACKGROUND

Students begin courses with intuition and prior knowledge about how to solve tasks. These preconceptions can be leveraged to guide them to a deeper understanding of computer science concepts [5], or it may lead to misconceptions, which if not corrected by the instructor, may persist and hinder their understanding of a subject. In computer science, previous work has shown that students have trouble with even the most basic concepts of programming [4, 18, 22]. In this work, we present the results of a preliminary pre- and post-course inventory of operating systems concepts, designed to examine both students' preconceptions about the material and the misconceptions they hold after completing an upper-division operating systems course.

Most collegiate courses combine a variety of assessments to gauge student learning, with student evaluation primarily centered around midterm and final exams. Such exams often ask students to perform calculations or provide explanations (that may be memorized) in addition to examining student understanding of the course's key high-level conceptual "take-aways." This assessment diversity has led to the development of concept inventories, which exclusively aim to reveal student understanding of critical high-level concepts.

In physics, the Force Concept Inventory (FCI) [15] revealed a dramatic difference between how students and instructors think about mechanics concepts. It demonstrated that final exams were not sufficient for revealing the persistence of students' basic misconceptions, even after they had completed introductory physics coursework. Subsequent work using the FCI to compare new, interactive pedagogical approaches to traditional lecture techniques has transformed the way introductory physics classes are taught, yielding significant student learning gains [11].

Concept inventories are now widely used in a variety of disciplines [9] to assess the impact of pedagogical methods. In the field of computer science, concept inventories have been created for digital logic [12, 13], algorithms and data structures [8], introductory courses [10, 17], and discrete math [2]. However, most pedagogical literature on the subject of operating systems has focused on designing student projects [3, 6, 14]. While some work has been done identifying overall principles of computer systems [16], there has not yet been any work towards characterizing student misconceptions of operating systems principles.

In this work, we discuss our experiences in developing a concept inventory for operating systems courses. This effort is part of a broader initiative to apply the principles of open source software to the development of concept inventories. We believe that by applying rapid, multi-author development models to the construction of computer science concept inventories, the educational community will benefit from high-quality, dynamic concept assessments that will lead to a significant increase in course coverage. Our assessment is publicly available online [1], and we welcome collaboration from interested parties.

We compare student results on our assessment, given as pre- and post-course multiple choice quizzes, to the results of similar questions from both in-class peer instruction questions and traditional exam questions. Our analysis of data collected from four operating systems courses provides insights into student understand-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE'14, March 5–8, 2014, Atlanta, GA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2605-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2538862.2538886 .

Table 1: A brief summary of the courses in which our assessment has been deployed.

Course	Weeks	Students	Methodology	Instructor
A	5	21	Peer Instruction	I_1
B	10	133	Lecture	I_2
C	10	133	Lecture	I_2
D	14	9	Peer Instruction	I_3

ing of the material. In this paper, we specifically examine the lowest-performing assessment questions and discuss what they reveal about student comprehension of operating systems concepts.

2. METHODS

For our inventory, we created questions to be administered before and after collegiate operating systems courses. This section characterizes the courses in which the assessment was administered and describes our question development goals.

2.1 Course Context

To date, our concept inventory has been administered in four courses, at two institutions, by three instructors. We summarize the course summaries in Table 1, referring to the courses as A through D and instructors as I_1, I_2, and I_3. Courses A through C were taught at a public, research-intensive university in the western United States. Approximately 23,000 undergraduate students attend the university, with about 1,200 students majoring in computer science. Course D was taught at a small, private liberal arts college in the midwestern United States. Approximately 2,800 undergraduate students attend this college, with about 60 students majoring in computer science.

Courses A through C are required for majors in the computer science department of the respective institution and are typically taken by students in their junior or senior year. Prerequisites for the course include computer organization, algorithms, and software engineering. Course D is an elective, which was taken exclusively by seniors. Its prerequisites include computer organization, algorithms, and systems programming. Students are expected to be comfortable with programming in C and working in a Unix-like environment prior to enrolling in all courses.

Course A was a five week, intensive summer course, with 21 students enrolled at the time of the final exam. Courses B and C were ten-week quarters, each with 133 students enrolled at the time of the final exam. Course D was a fourteen-week semester long course, with 9 students enrolled at the time of the final exam.

In courses A and D, students were given the pre-test as an in-class assessment prior to the first lecture, and the post-test on the final day of instruction. In courses B and C, the pre and post tests were given as part of an optional discussion section. The pre-test was given in the first discussion section, and the post-test was given as part of a final review section.

Courses B and C were taught by an experienced instructor who has taught the course many times. They were taught in a traditional lecture format. Courses A and D were taught by relatively inexperienced instructors using peer instruction (PI) pedagogy [7]. PI utilizes frequent student discussion to engage students and provide a more interactive learning environment. Students were presented with four to six multiple-choice PI questions per class period. Students responded twice for each question, and we collected data at both points. The first time, students responded individually to the presented questions. Afterwards, they discussed the question in small groups of three or four and then responded again, as a group. Student response data was captured using hand-held "clickers," and we refer to individual and group response data in our results. After the group response phase completed, the instructor guided the class though a class-wide discussion. In this paper, we incorporate the data collected through the peer instruction process to give us more insight into how and why students performed as they did on specific questions in the concept inventory.

Table 2: A brief summary of our assessment's questions.

Question	Concept
1	Timing of system calls and context switching
2	How kernel code is executed
3	Choosing a scheduling algorithm
4	The bounded buffer problem (discussed in 4.3)
5	Segments versus pages in memory management
6	Indirection in file pointers (discussed in 4.1)
7	Polling versus interrupts (discussed in 4.2)
8	Access Control Lists versus Capabilities
9	Multiprocess execution on a single CPU
10	Least recently used eviction (Courses B-D only)

2.2 Goals and Question Development

Operating systems courses typically focus on analyzing design trade-offs and selecting the best solution for a specific set of circumstances. Rather than identifying a single right answer, students in an operating systems course must often choose from among a set of viable answers, each of which is ideal for a specific situation. Unlike exam questions, which may involve students reciting memorized pros and cons for different approaches, we aim for our assessment's questions to force students to examine the *implications* of choosing between opposing trade-offs. Many of our questions present a scenario and ask students to apply the course's principles in choosing the most appropriate option. We feel that this form of question better allows us to assess their understanding of the subject's core concepts.

At the same time, we intend for our inventory to be used as both a pre- and post-test, and thus we would like our questions to be reasonably accessible to students who do not have prior knowledge of the subject. To this end, we minimized our use of specialized vocabulary where possible and defined terms when we felt it might improve the accessibility of a question. By doing this, we hope to capture how students conceptualize important material prior to any formal instruction. To reduce the effects of students randomly guessing, we provide an option of "I am not familiar with this terminology / I don't know" as the final choice (E) for every question.

With these goals in mind, we developed questions based on our past experiences teaching operating systems courses and working with students who had difficulties with the material. We identified areas that students frequently have trouble understanding and developed multiple choice questions with distractor answers based on our experiences with common student misconceptions. We created a ten-question, multiple choice inventory that poses high-level, conceptual questions about key concepts in operating systems. Table 2 outlines the topic of each question.

Our inventory is available online in its entirety [1], and we encourage collaborative development and refinement from additional authors. Moving forward, we intend to rigorously validate our assessment by consulting with other experts in teaching operating systems, interviewing students, and collecting additional data by giving our assessment to more classes of students. Our goal is to construct a validated and widely-adopted operating systems concept inventory in collaboration with the operating systems academic community.

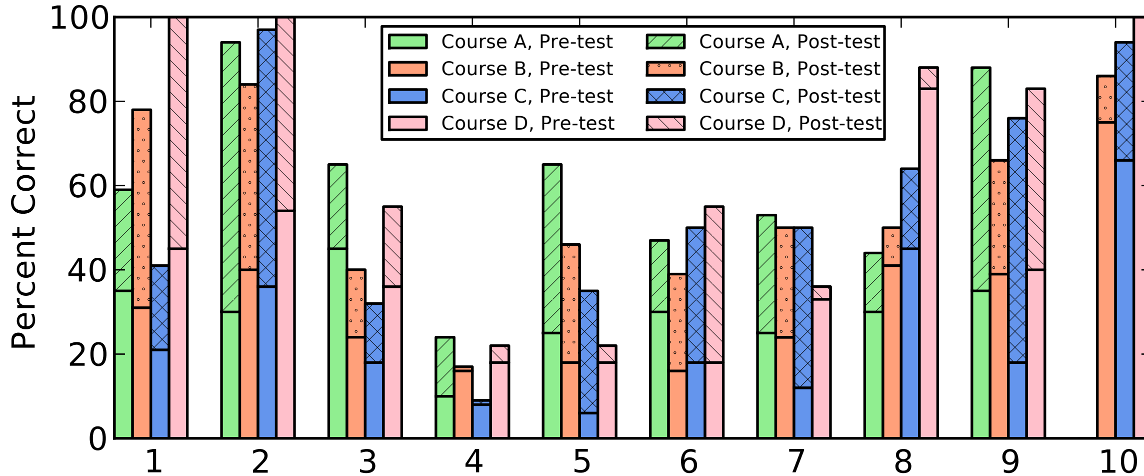


Figure 1: An overview of concept question scores, grouped by question number as identified in Table 2. Each group of bars depicts student performance across the four courses (A–D) in which the inventory was deployed. We focus our analysis of selected results (Section 4) on questions 4, 6, and 7, which showed relatively poor performance.

3. RESULTS OVERVIEW

All courses showed improvement across each question between the pre- and post-test. In courses B, C, and D, students averaged 3.3 correct responses (median of four) on the pre-test, and 5.5 correct responses (median of five) on the post-test. In course A, where students were only given questions 1 through 9, students averaged 2.7 correct responses (median of three) on the pre-test and 5.0 correct responses (median of six) on the post-test.

Figure 1 illustrates the pre-test scores and post-test improvement, grouped by question and class. While variation is to be expected between multiple courses taught by different instructors, there are clear trends across classes, particularly for questions on which students performed poorly. Students uniformly performed the worst on question 4, a fairly straightforward question on the bounded buffer problem. They uniformly did well, even in the pre-test, on question 10, which uses an analogy about bookshelves to discuss the least recently used eviction policy. They also all show a great deal of post-course improvement on question 2, which covers how kernel code is executed, and question 9, on the relative execution rates of processes.

This uniformity in student answers across classes allows us to generalize about the content students struggle with in operating systems courses. They generally have trouble with synchronization, indirection, and I/O, all of which we discuss at length in Section 4. High student performance on the pre-test for question 10 shows that many students already have an intuitive understanding of the concept of least recently used eviction, which can be leveraged when teaching that concept.

Figure 2 depicts the cumulative percentage of responses, across all four courses, that selected “I don’t know.” or chose the most popular distractor. While every question showed improvement on the post-test, the extent of the improvement varied significantly across questions. Interestingly, students chose a much wider selection of answer choices on the pre-test than the post-test, where their incorrect responses tended to strongly favor one distractor. This implies that even for questions that students did not perform well on, their answer selection was influenced by the material in the course. In other words, while they may not have always selected the correct choice, students narrowed their selections, and when incorrect, they tended to focus on the “most attractive” distractor choice. Furthermore, the sharp decrease in the rate of *E* responses indicates that students felt more confident answering the questions, even if their selections were incorrect.

3.1 Changes to the Concept Inventory

As we continue to refine the inventory, we have modified the content several times, mainly to improve student comprehension of the questions. The most significant change was the addition of question 10, on LRU replacement. This question was given to courses B through D. Additionally, the wording of question 8, on access control lists (ACLs) versus capabilities, was changed after course A. This question originally used the analogy of a theater, where patrons would either be let in by a bouncer checking names on a list (representing ACLs), or a ticket (representing capabilities), along with two additional distractor answers. In course A, 44% of students chose the correct answer of ACLs, but an equal number chose the distractor answer of tickets. We suspected that the analogy to the theater was leading them to choose the wrong answer, so the question was rephrased to use the analogy of a party instead. Question 7, about polling versus interrupts, was also changed after course A, in order to better characterize the device. However, this clarification did not have a discernible effect on students answering this question correctly.

4. SELECTED RESULTS

In this section, we present a selected subset of our OS concept assessment results. We discuss individual questions and examine the questions’ core concepts in the context of student responses. Due to space constraints, we cannot cover every question in detail. Instead, we choose to focus on the three of the lowest-performing questions. We feel that these questions best illustrate common student misconceptions and that their analysis provides the most potential to positively affect student learning.

These three questions covered important operating systems topics like indirection (44% answered correctly), I/O (50%), and synchronization (16%). We emphasize these questions because, even after completing an operating systems course, our results indicate that students still harbor misconceptions on these topics. For each question, we present the question along with the cumulative percentage of answer choices annotated by a (pre-test %, post-test %) tuple to indicate how frequently students selected each answer on the pre- and post-test, respectively, from all four courses. We mark

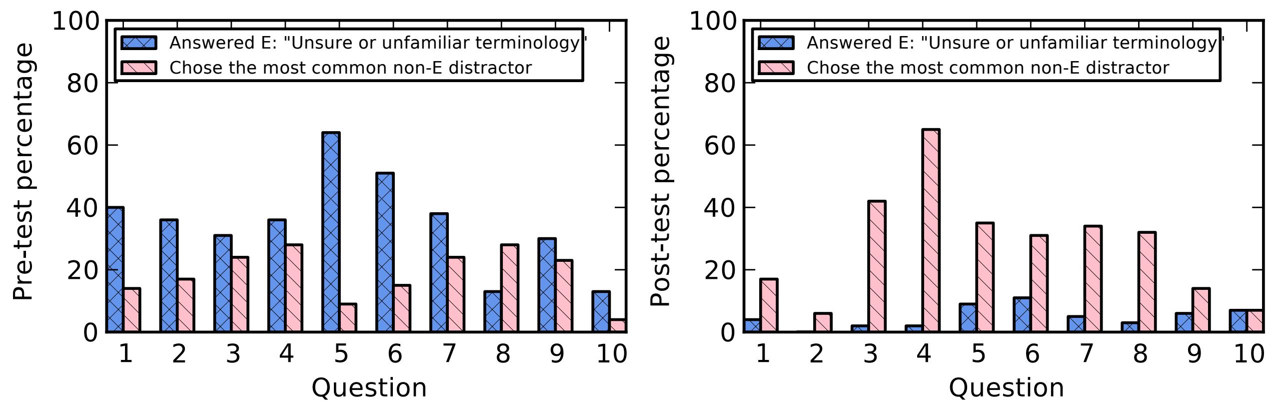
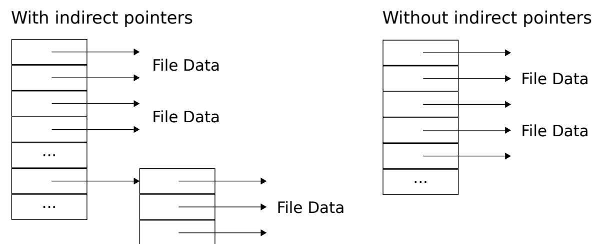


Figure 2: The cumulative percentage of *E* and popular distractor responses, per question, on the pre-test (left) and post-test (right).

the correct answer with italics. All comparisons with exam and clicker questions are taken from course A.

4.1 Indirection: File System Block Map

Suppose we have a file system that uses two forms of pointers to find file data: direct and indirect pointers. We refer to this set of pointers as a "block map." Direct pointers point directly at file data, and indirect pointers point at a separate disk block that contains multiple pointers to file data. Every file in the system uses the same block map structure, with unused pointers directed at null. Eliminating the indirect pointers while supporting the same maximum file size would be...



A) (7%, 7%) Beneficial - The new block map would save space storing small files.

B) (8%, 7%) Beneficial - The new block map would save space storing large files.

C) (19%, 44%) *Detrimental - The new block map would waste space storing small files.*

D) (15%, 31%) Detrimental - The new block map would waste space storing large files.

E) (51%, 11%) I am not familiar with this terminology / I don't know.

Concept Question 6: File System Block Map Indirection

Indirection is a recurring concept in many areas of computer science. Students typically face it first in the form of pointers, where indirection often proves to be difficult for novice programmers [19]. In the context of operating systems, indirection is often used to make the metadata associated with storing a variably-sized entity (like a file or process memory) proportional the entity's size.

Question 6 examines this concept of indirection, specifically for storing file metadata (the same concept applies to using multiple levels of page tables to reduce page table sizes). Our pre-test results show that this question had the second-highest rate of *E* answer selections at 51% (the highest was 64% for a segmentation vs. paging question). We suspect that this concept is intimidating to students who have not previously encountered the terminology and constructed a mental model of indirection. In our experience, students are typically much more comfortable with direct pointers, likely because their behavior resembles that of an array, which is a familiar construct.

We presented two in-class peer instruction questions to students on this topic, and the resulting data confirms that students more easily reason about direct pointers. The first question introduced the idea of using only direct pointers to store file metadata. Our data shows that 78% of the class responded correctly to the individual response, and 100% responded correctly after discussing the question with their group. The second PI question asked students to compute the amount of metadata that would be needed to store a particular file when given a block map structure with both direct and indirect pointers. The correct response rate was 44% individually and 39% after the group discussion. This was one of a small minority of questions for the entire course in which the correct response rate *decreased* after group discussion, which indicates that students were uncomfortable with the concept of indirection.

Adding indirect pointers introduces a challenging concept, since systems often use indirection as an optimization as opposed to it being necessary for correctness. We included question 6 on the assessment due to our observation that students often ask why we bother with this added complexity. We believe that the challenge of indirection may stem from the fact that a solution using only direct pointers *seems* to be attractive to students because it would be relatively straightforward.

Final exam results further corroborate this observation with two questions related to the concept of indirection. The first question presented a scenario in which we asked students to determine the numerical properties of a hypothetical page table and then explain (as free-response text) whether or not they would recommend using multiple levels of page tables (indirection) in this scenario. The second question presented a similar hypothetical scenario of a file system block map and eventually asked students how we justify the complexity of using indirect pointers. Unsurprisingly, students performed better on second question, where they were implicitly told that the complexity was useful, as opposed to the former, in which students had to make that decision themselves.

Given these types of responses to exam questions and the results of question 6, indirection is clearly an unintuitive concept. Even after demonstrating indirection for multiple class topics, many students still struggle with indirection misconceptions. We believe that students who are learning these concepts may benefit from instructors initially introducing an abstract form of indirection, perhaps by relating it to C memory pointers, which students are likely to find familiar. Discussing indirection abstractly by mapping it to a more comfortable topic [20] may enable students to construct a mental model of indirection prior to adding the confounding details of memory management or file systems.

4.2 I/O: Polling vs. Interrupts

The core concept for Question 7 is the use of polling vs. interrupts to retrieve data from an I/O device. To force students to consider the device's characteristics without relying on memorization, we invented a new device with no obviously recognizable "type." Ideally, students answering this question would recognize answer *B* as polling and *D* as interrupts. Options *A* and *C* were meant as plausible distractors, despite being impractical for this device.

> Your company has developed a new I/O device, and you've been tasked with writing its driver. The device produces a byte of data at frequent, regular times. The device does not save data after it has been produced. Which of the following options for accessing the device would best suit your driver?
>
> A) (0%, 1%) Have the kernel check the device for new data when it is idle.
>
> B) (23%, 50%) *Have the kernel check the device for new data at set time intervals.*
>
> C) (15%, 10%) Have the kernel check the device for data when applications request it.
>
> D) (24%, 34%) Have the device alert the kernel when it has new data so that the kernel can collect the data as soon as it is available.
>
> E) (38%, 5%) I am not familiar with this terminology / I don't know.

Concept Question 7: Polling vs. Interrupts

Students taking the post-test predominantly chose between *B* and *D*, indicating that they recognized these two options. In another assessment question, 90% of students correctly identified the role of interrupts in transferring execution control to the kernel. We therefore believe that students are comfortable with these terms and their definitions.

However, despite recognizing polling and interrupts, applying them to a scenario remains non-intuitive to students. Having been given an explanation of polling and interrupts in class, we presented the students with a peer instruction question in which we asked whether they would use polling or interrupts for keyboard and disk devices. Like the indirection concept described in section 4.1, this was another of the few PI questions whose correct response rate *dropped* after group discussion (41% individually to 29% after discussion), with most students opting to use polling for disk devices. Such a drop indicates that students found the concept difficult and that many students lacked confidence in their original answer.

We asked a similar free-response question on the final exam regarding the use of polling or interrupts for a disk device. Students performed better on the exam, with 81% correctly identifying interrupts as the correct choice for disks. However, only 62% correctly articulated *why* they made that decision, despite the class having directly discussed this topic after the PI question. Our conclusion is that, like the other common misconceptions we identify in this paper, students most frequently struggle when asked to analyze the trade-offs of a scenario when described primarily using text (as opposed to code).

4.3 Synchronization: Bounded Buffer

Proper use of synchronization and concurrency is a difficult subject for students to understand [21, 23]. To examine this concept, question 4 addresses synchronization using a producer–consumer (bounded buffer) scenario. We chose this scenario because it benefits from applying the less-common approach of using semaphores purely for synchronization, rather than mutual exclusion.

In our experience teaching operating systems, students commonly believe that mutual exclusion is required to solve this problem correctly. The results here show this to be the case, despite lecture slides and peer instruction questions indicating the contrary to students. While not incorrect, enforcing mutual exclusion is inefficient and unnecessary for this scenario.

> Suppose you have a fixed-size queue shared between two processes on a system that has one CPU. One process produces data and puts it into the queue, and the other process reads data from the queue and removes it. For correctness, you want to ensure that the producing process only writes when there is space available, and the consuming process only reads when there is data available. Your solution needs to be as efficient as possible. You should:
>
> A) (4%, 2%) Only let one process access the queue at a time.
>
> B) (17%, 15%) Have the producing process signal to the consuming process when it fills a slot - the consuming process will only consume as many slots as received signals.
>
> C) (14%, 16%) *Have the producing process signal when it fills a slot, and the consuming process signal when it empties a slot.*
>
> D) (29%, 65%) Have the producing process signal when it fills a slot, the consuming process signal when it empties a slot, and let only one of them access the queue at a time.
>
> E) (36%, 2%) I am not familiar with this terminology / I don't know.

Concept Question 4: Bounded Buffer

Initially, students performed well on related peer instruction questions. When shown unsynchronized bounded buffer code and asked why synchronization was required, 81% of students correctly identified (after group discussion) that synchronization is necessary to prevent overflowing or underflowing the buffer rather than to provide mutual exclusion. Likewise, when asked to add semaphores to this unsynchronized example, 81% of students responded correctly, and no students voted for a mutual exclusion distractor. This indicates that students are capable of understanding the basics of using semaphores for synchronization, but they struggle applying this understanding to a less familiar "word problem" scenario that does not involve code.

We compared this to a midterm question, which had a component that required a bounded buffer style synchronization solution and a different component that required a mutex exclusion solution. The question displayed unsynchronized code for two processes: one

representing a refinement department, refining raw materials into a product, and another representing a shipping/receiving department, providing the refinement department with raw materials, and then shipping the finished product. Students were asked to identify why the code needed semaphores and to correctly insert them to eliminate any problems they identified. To categorize whether students use synchronization versus mutual exclusion, we count as synchronization any call to wait(a) followed by signal(b) in one process, and wait(b) followed by signal(a) in the other process, regardless of whether the student put the calls in the correct place.

Looking at just the synchronization component of the question, only one student (5% of the class), attempted to solve the problem using both synchronization and mutual exclusion, a solution which would correspond with question 4's *D* choice. Forty-five percent of students used mutual exclusion with no attempt at synchronization, which would correspond to answer *A*, and fifty percent of students made some attempt at synchronization with no additional mutual exclusion, an answer that corresponds to the correct option, *C*.

Of the four students who answered C on the post-test in course A, three of them also used synchronization on the midterm, consistently indicating that they understood using synchronization without mutual exclusion. Of the ten students who answered D, five had used mutexes, and five had used synchronization (including the student who used both). Since very few students chose A in either the pre- or post-test, this could indicate that students simply didn't recognize it as the bounded buffer problem. However, when asked to describe in words why synchronization was needed, almost all of them correctly identified a buffering problem, although none used the phrases "Bounded Buffer" or "Producer–Consumer."

Eighty-six percent of the students could accurately describe why synchronization was needed. One student wrote, "One problem that could occur is that, if T2 tries to unload and there is no product produced from T1, there is nothing to tell T2 to wait if there is nothing available from T1" while describing the problem on the midterm. Despite his accurate description of the bounded buffer problem, he both used mutual exclusion on the midterm and answered D to question 4, indicating that while he conceptually understood the need for synchronization in this scenario, he was unable to connect it to the correct solution.

Despite what we might tell them, most in-class and textbook examples (besides the bounded buffer) *do* use synchronization to enforce mutual exclusion, and many operating systems courses often strongly emphasize identifying critical sections and resolving race conditions with mutual exclusion. Our results indicate that when teaching operating systems, instructors should cautiously emphasize the different uses of synchronization, rather than focusing primarily on mutual exclusion.

5. CONCLUSION

This paper looks at preliminary results of a concept inventory for operating systems material. We use these results to explore areas in which students harbor misconceptions regarding indirection, I/O, and synchronization in an operating systems context. The misconceptions we identified point to student difficulty in analyzing trade-off scenarios and interpreting the implications of competing solutions. Our concept assessment and preliminary results provide valuable insights to help operating systems instructors understand student misconceptions. These results will be expanded as we continue to collect additional student responses, and we encourage other instructors to collaborate in constructing an operating systems concept inventory.

6. ACKNOWLEDGEMENTS

We are indebted to Beth Simon, Leo Porter, Cynthia Lee, and Sat Garcia for providing valuable feedback on earlier drafts of our concept inventory, and to Joe Pasquale for allowing us to adapt his course materials for peer instruction, and all his other help.

7. REFERENCES

[1] http://git.io/ZRngng.

[2] V. Almstrum, P. Henderson, V. Harvey, C. Heeren, W. Marion, C. Riedesel, L. Soh, and A. Tew. Concept Inventories in Computer Science for the Topic Discrete Mathematics. *ACM SIGCSE Bulletin*, 38(4):132–145, 2006.

[3] J. Andrus and J. Nieh. Teaching Operating Systems Using Android. In *SIGCSE*, 2012.

[4] P. Bayman and R. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Communications of the ACM*, 26(9), 1983.

[5] T.-Y. Chen, G. Lewandowski, R. McCartney, K. Sanders, and B. Simon. Commonsense Computing: using student sorting abilities to improve instruction. In *SIGCSE*, 2007.

[6] W. Christopher, S. Procter, and T. Anderson. The Nachos Instructional Operating System. In *USENIX Winter*, 1993.

[7] C. H. Crouch and E. Mazur. Peer Instruction: Ten Years of Experience and Results. *American Journal of Physics*, 69(9), September 2001.

[8] H. Danielsiek, W. Paul, and J. Vahrenhold. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *SIGCSE*, 2012.

[9] D. Evans, G. Gray, S. Krause, J. Martin, C. Midkiff, B. Notaros, M. Pavelich, et al. Progress on Concept Inventory Assessment Tools. In *IEEE Frontiers in Education*, 2003.

[10] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Identifying Important and Difficult Concepts in Introductory Computing Courses using a Delphi Process. In *SIGCSE*, 2008.

[11] R. Hake. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66:64, 1998.

[12] G. Herman and J. Handzik. A preliminary pedagogical comparison study using the digital logic concept inventory. In *IEEE Frontiers in Education*, 2010.

[13] G. Herman, M. Loui, and C. Zilles. Creating the Digital Logic Concept Inventory. In *SIGCSE*, 2010.

[14] R. Hess and P. Paulson. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *SIGCSE*, 2010.

[15] D. Hestenes, M. Wells, and G. Swackhamer. Force Concept Inventory. *The Physics Teacher*, 30:144–158, March 1992.

[16] M. Holliday. Teaching computer systems through common principles. In *IEEE Frontiers in Education*, 2011.

[17] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying Student Misconceptions of Programming. In *SIGCSE*, 2010.

[18] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating the Viability of Mental Models Held by Novice Programmers. *ACM SIGCSE Bulletin*, 39(1):499–503, 2007.

[19] I. Milne and G. Rowe. Difficulties in Learning and Teaching Programming - Views of Students and Tutors. *Education and Information Technologies*, 7(1), 2002.

[20] J. D. Novak. Concept Mapping: A Useful Tool for Science Education. *Journal of Research in Science Teaching*, 27(10), 1990.

[21] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). In *Invited presentation at USENIX ATC*, 1996.

[22] R. Pea. Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, 2(1):25–36, 1986.

[23] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.