Synchron-ITS: An Interactive Tutoring System to Teach Process Synchronization and Shared Memory Concepts in an Operating Systems Course

Manoj Kumar Putchala* and Adam R. Bryant†
Department of Computer Science and Engineering
Wright State University
Dayton, Ohio USA 45435
Email: {*putchala.3 †adam.bryant}@wright.edu

***Abstract*—Operating Systems is a course in undergraduate computer science curricula to teach students concepts relating to the environment on which their applications run. In practice, operating systems software is very complicated, and the internal processes and mechanisms are often difficult for students to grasp, particularly those that still struggle with programming. Many operating systems courses are taught by describing high-level abstractions of structures and algorithms from a textbook, and then providing homework or project assignments that, in the interest of being tractable for the student, may be disconnected from the way an operating system actually performs its tasks.**

These methods only present a theoretical display of essential concepts which lack concrete examples to anchor the concepts. What many students need is a way to connect the low-level details of an operating system's implementation with the high-level abstractions provided in the class, all while being accessible to people who are still improving newly-acquired programming skills. To bridge the gap between operating systems theory and implementation, we propose an interactive approach to present the concepts involved with process synchronization and shared memory management.

Keywords–Operating systems, interactive tutoring systems, process synchronization, shared memory algorithms

I. Introduction

Process synchronization and algorithms for synchronizing shared memory are concepts students must learn in an undergraduate Operating System (OS) curriculum. Typically, this is the first encounter students will have with concurrency and parallelism. In process synchronization, when two or more processes share a memory region, they should access it in an orderly manner to avoid erroneous outputs. Various multi-threading techniques and locking mechanisms are already used in practice to permit concurrency of processes using shared memory.

The field of synchronization has developed rapidly, resulting in synchronization for high end processors to perform at maximum speed. A person who learns synchronization concepts gets information from sources like textbooks, online tutorials, video lectures, or from direct instruction in a university OS course. But, considering the complexity and depth of concepts in this area, these sources may not provide the intuition or in-depth knowledge an average student would need to code or simulate the concurrency of programs in practice. The diversity of solutions provided for synchronization and concurrency problems can be confusing to undergraduate students.

Hence, there is a need to teach these concepts in a more user-friendly manner that allows a student to start from high-level concepts, and work down to implementation-specific details. We propose a two-step process of learning memory synchronization that involves first visualizing the step-wise data flow among processes (in *autonomous mode*), and later allowing a user to validate his or her level of understanding of the concepts with a guess-and-test process, stepping through the simulation and verifying the guess with output from the tool (in *manual mode*).

The concepts presented in understanding shared memory synchronization involve solutions to the "producer / consumer problem," the "dining philosophers problem" (using semaphores), and mutex locks. We designed a tool to demonstrate each of these concepts at a high level, and are working toward integrating the high-level models with low-level information collected from a running operating system and real-world operating system source code. In the current version of the tool, we have created visual representations of data flow between producer and consumer processes, the modifications of lock values, changes to semaphore values, and data in buffer queues along with pop-up explanations at each point of execution.

The rest of the paper describes the use cases, architecture, implementation, and research focus of this system along with our rationale for various design decisions. Section II reviews related research on educational simulations for OS concepts and compares features of those simulations with features of our own interactive tutoring tool. Section III discusses the use cases of the system and demonstrates the activities and components of those use cases using the Unified Modeling Language (UML). Section III-C explains the distinction between the two learning modes in the tool: *autonomous mode* and *manual mode*. Section III-D describes the architecture of the system and the design and features of the system's user interface (UI).

It also discusses how the various components are used in the context of classical synchronization problems from operating systems theory. Section IV describes our plan for performing user testing and evaluation for the software in various stages of its life cycle. Finally, Section V presents the conclusions from our work and discusses planned future improvements of the tool such as 3D visualizations, data model development, database integration and integration with information from a running operating system.

II. Instructional Tools for OS Concepts

Concurrency and synchronization are part of the core of the computer science curricula recommended by the IEEE and Association for Computing Machinery (ACM). Students are expected to learn the need for concurrency, understand potential run-time problems that can occur with multitasking, understand the mechanisms invented to handle concurrency and why they are used, and understand how to implement these different techniques [1]. Unfortunately, in the way these subjects are often delivered, there is a potential for leaving a knowledge gap between students' theoretical knowledge of these concepts and the practical implementation of them in real-world operating systems.

At many educational institutions such as ours (Wright State University), the approach for teaching these concepts involves a combination of classroom lectures, quizzes, programming labs, and homework assignments. Since operating systems code can be very complex, there is a potential for students to successfully complete such a course but still have a large knowledge gap between high-level theory and low-level implementation. Depending on the structure of each individual course and its computing and lab resources, there may or may not be course time devoted to practical guided programming exercises in the classroom in a single semester, so students are either forced to understand complex systems code themselves, or kept away from that level of detail entirely.

Students may also have difficult programming for a real operating system. Operating systems run in supervisor mode which makes the standard debugging methods students are used to using much more difficult. These problems can be exacerbated in schools that teach computer science courses using only high-level, "non-systems" languages, as it requires further abstraction of the concepts from the implementation details. Even source code is abstracted away from how synchronization algorithms run on the hardware, and it may be difficult for students to relate source code samples to the effects of on a running OS.

A software tool to connect high-level theory with low-level implementation details must thus meet the following requirements:

1) Be easy to use, easy to understand, and accessible for students who have limited knowledge of systems-level programming in C or assembly language,
2) Present theoretical concepts, data structures, and algorithms involved with synchronization and shared memory, similar to the box-and-line diagrams found in Operating Systems textbooks,
3) Allow a student to interact with the tool to explore those concepts, data structures, and algorithms work,
4) Present realistic source code examples that can help students connect theory to implementation details, and
5) Collect, integrate, and visualize data from a running operating system so students can see the implementation details in motion.

A. Instructional Operating Systems

There are many instructional operating systems that were developed to support Operating Systems education at many educational institutions. Ambitious projects such as MINIX [2] and the GeekOS systems [3] follow "bare-hardware" approach enabling the users to modify the code directly on the computer hardware or in a virtual machine monitor. However, these systems contain a substantial amount of source code (4,202 lines for Geek-OS and 14,866 lines of MINIX), making them very complex for under-graduate students that are mostly unfamiliar with systems-level programming in C or C++.

The Nachos Instructional Operating System [4] is implemented in C++ and MIPS assembly code (and was later re-implemented in Java [5]). It runs as a set of processes on a simulated MIPS architecture. Dex-OS [6] runs only on the Intel 386 platform in 32-bit protected mode, but it is written in FASM assembly code and C and covers mainly memory management and process management concepts. ICS-OS [7] is a similar instructional operating system based on the DEX-OS kernel. It has poor graphical visualizations, which makes it mostly unfit for tutoring purposes.

Simple-OS [8] is a component-based operating system which provides the user with a graphical interface and access to low-level code modifications, but it does not cover shared memory concepts. While each of these systems is useful for exposing students to running code, they do so at the expense of exposing students to the full complexity of an operating system, and requiring the student to be able to read and understand large programs written in C, C++, Java, and in many cases assembly language; understand many systems programming concepts which they may have not yet learned. Pintos [9] and RCOS [10], these systems also do not directly provide mental connections for students between the implementation details of the particular system and the theoretical concepts covered in the classroom.

B. Tutoring Environments

Others have created software-based tutors to present students interactive environments for learning OS concepts. Proc_OS [11] and Alg_OS [12] are two such software tools. While these models allow users to interact with particular OS concepts, they are limited in the other way: they present theoretical concepts but do not connect those concepts to running code. Additionally, these systems do not cover process or thread synchronization and their associated mechanisms. Another unnamed learning tool created by Besim Mustafa

[13], provides two simulators for the OS and for the CPU which showed useful in helping students learn Operating Systems course material. Unfortunately, the simulators only cover basic visualizations of the producer / consumer problem.

There are various other high-level simulations such as like XINU [14], CPU Sim [15], BACI [16], SOSim [17], and TOST [18] which provide students interactive tutorials on various operating systems concepts. Specifically, the MINT simulator, a front end tool for efficient simulation of shared-memory multiprocessors, is intricate and only focuses on memory hierarchy simulations that are event-driven [19]. SMART, an extension to the MINT simulator, provides a user-friendly interface but lacks the interactivity to be used for tutoring purposes [20]. The high performance simulator TROJAN [21] is used for studying the network traffic patterns, cache behavior, and multiprocessor architectures. SIME [22], is a memory simulator that specifically visualizes the memory management structures of an operating system, it does not include the shared memory concepts. These simulators are either overly complex for our purposes, not user-friendly, or are do not provide the interaction with synchronization and shared memory concepts.

An interactive tutoring tool that meets all of our requirements does not currently exist. Existing tools tend to provide either high-fidelity OS implementation details or low-fidelity interactive visualizations, but not both.

III. Design and Implementation

We designed Synchron-ITS as an interactive visualized tutoring tool to meet the above stated requirements and solve this problem. In our design, we have tried to overcome many of the problems mentioned in Section II and also developed a number of features to help instruct students in synchronization and shared memory concepts.

The design architecture of this interactive tutoring tool is established based on the principle - "All computer students must learn to integrate theory and practice" [23]. Synchron-ITS is different from previous simulators in that we are integrating visualizations of real-world data from a running Linux operating system with high-level diagram models and source code examples. The rationale is that this will enable students to see connections between abstract models of synchronization (such as the dining philosophers problem) and how those models manifest themselves, both in source code and system behavior.

A. System Concept and Design

A use case diagram to represent interaction between users and the Synchron-ITS system is shown in Figure 1. The top-level use cases for a user are:

UC1. Select log in type (guest user or registered user).
UC2. Create user account (if registered user).
UC3. Select a concept from concept list.
UC4. Choose a mode (*Simulator*, *Manual*, or *Self-check*).
UC5. Estimate and validate self-check results.
UC6. Logout.

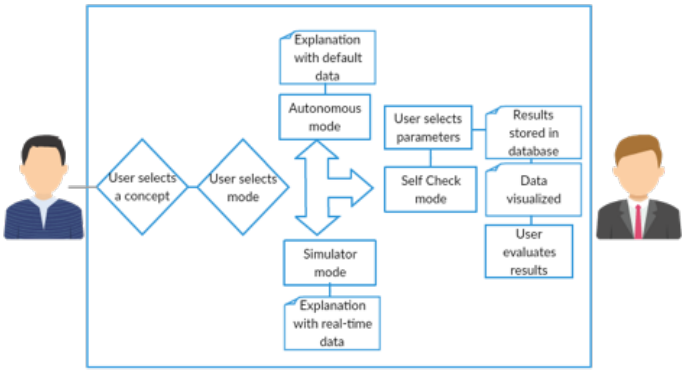

Fig. 1. Use Case Diagram.

Most of the interaction of Synchron-ITS falls under UC5. The sub-use cases of UC5 are:

1) UC5.1: View synchronization problems: We designed visual 2D displays and representations for the producer / consumer problem, the dining philosopher problem, the bounded buffer problem, and their corresponding synchronized solutions. Synchron-ITS also displays monitors, semaphores, and mutex locks visually in 2D. This helps the user to compare and contrast the different synchronization problems and their solutions with each other.

2) UC5.2: Select parameters: When the user clicks on self-check mode, he or she will be given with a chance to choose the parameters used in the given concurrency algorithm and then evaluate how they affect the system's behavior.

3) UC5.3: Perform self-check: To increase the students confidence in *Self-Check* Mode, he or she can verify the implementation by guessing the expected results, and then comparing those to the actual results of running the given algorithm. This is to aid students in learning from their mistakes and to help them improve their performance.

B. Other Requirements

1) Real-time mode: In *Real-Time Mode*, Synchron-ITS displays visualizations for the real-time values of the memory segments and shared variables drawn from a running Linux operating system. Integrating real-time mode requires selecting appropriate processes from the Linux kernel from which to sample (that feature the synchronization issues in question), developing software probes to gather information, and developing visual representations for this data.

2) Database integration: Student data will be stored in a database to track performance over time with respect to each of the content areas used in *Self-Check* Mode. This database contains the student ID, name, synchronization algorithm ID and evaluation marks as columns in the main table.

3) Monitoring and control: A key feature for being able to interact with the simulation is to allow the users to control parameters and monitor the execution of processes. Synchron-ITS allows the user to start, stop, suspend, and re-start processes during simulated execution. Users can also change the speed of execution of the algorithms.

4) Programming assignments: The tool provides a set of small and focused programming assignments to implement concurrency in C and Java. The list of concurrency assignments are:

1) Implement a synchronization primitive to block multiple process that are depending on an event to get triggered;
2) Implement a mutex mechanism for multiple processes by a user-space daemon;
3) Implement a monitor solution for the dining philosophers problem; and
4) Select and implement an appropriate concurrency primitive which is efficient and correct for a various synchronization problems.

C. Modes of Operation

The educational OS simulator we developed contains three different modes of operation: *Autonomous* mode, *Real-time* mode, and *Self-check* mode (Figure 2).

1) Autonomous mode: In autonomous mode, the algorithm executes with no interruption until it has completed. The algorithm runs with default parameters from data in the database.

2) Self-check mode: It is important for students to be able to validate their level of understanding after being presented with the concepts and how they are implemented. Self-check mode mode provides an interactive approach, in which the program steps into a crucial step in the algorithm and prompts the user to guess the values that will result upon completion of that step. If the user estimates the output correctly, the execution carries on to the next step. Otherwise, the system presents a message containing the actual correct values and an explanation as to possible misunderstandings that might have caused the error.

3) Real-time mode: This feature is still under development and is planned to be implemented in the next version of the software. The system will draw data from a running Linux system so users inspect the behavior of the operating system. Synchron-ITS is written in the Java programming language, and modules for accessing data from the operating system are written as callable C programs in Linux. Synchron-ITS accesses them using interprocess communication (IPC) through method calls to the Java Native Interface (JNI). In this way, the data collected from the operating system will be fetched and visualized in the Synchron-ITS JFrame window. A few examples of IPC commands are shown in Figure 3.

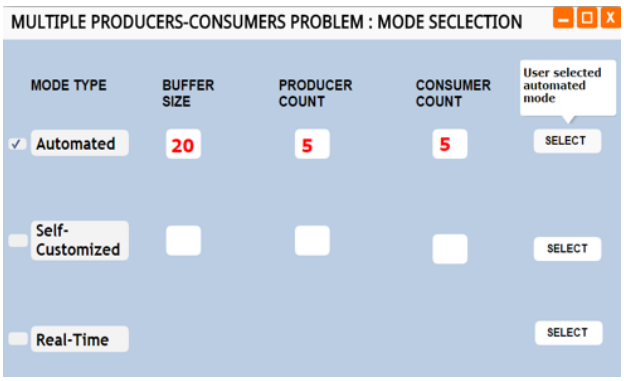

Fig. 2. Mode Selection Window.

```
# ipcs -m -c

------ Shared Memory Segment Creators/Owners --------
shmid      perms      cuid       cgid       uid        gid
1056800768 660        oracle     oinstall   oracle     oinstall
323158020  664        root       root       root       root
325713925  666        root       root       root       root
# ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0103eefd 0          root       664        1
0x0103eefe 32769      root       664        1
0x4b0d4514 1094844418 oracle     660        204
# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0xc616cc44 1056800768 oracle     660        4096       0
0x0103f577 323158020  root       664        966        1
0x0000270f 325713925  root       666        1          2
```

Fig. 3. IPC Commands in Linux.

D. Implementation

The system is designed and developed using Java. Each window is implemented as an instance of the JFrame class. Considering the planned improvements, we have followed an object-oriented paradigm to help modularize and expand the system in the future. The views in the system are composed of the main window, and both problem and solution windows for each of the synchronization problems.

1) Main Window: The main window opens up when the user successfully logs into the application. A user account is required to access the system, so the first time a user access it, he or she creates a new user account and logs into the application with valid credentials. An administrator account is used for teachers to evaluate students' performance.

The main window contains the list of synchronization concepts displayed as buttons in the UI. The user can explore each concept by clicking on the corresponding button (Figure 4).

2) Producer / Consumer Problem Window: The producer / consumer problem is a basic synchronization problem found in most Operating Systems textbooks. The problem demonstrates two CPU threads, one that produces data and one that consumes it, and both share a data variable (in this case a counter), which may cause inconsistency or errors due to improper signaling or synchronization. The reason for improper synchronization can be due to a lack of proper locking or signaling for the shared data which may results in incorrect outputs of shared variables or deadlock situations.

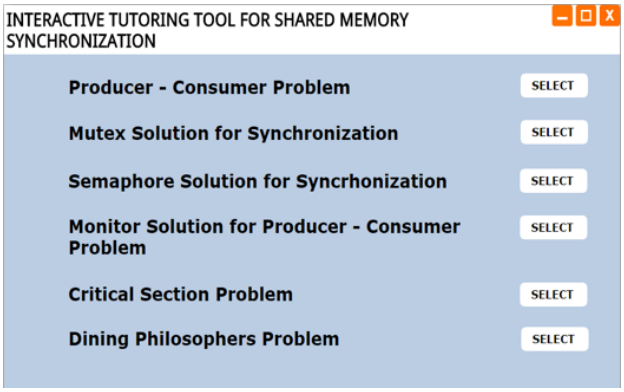

Fig. 4. Concept Selection Window.

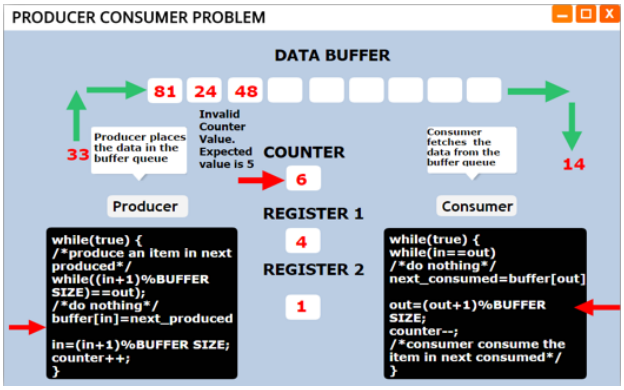

Fig. 5. Producer / Consumer Window.

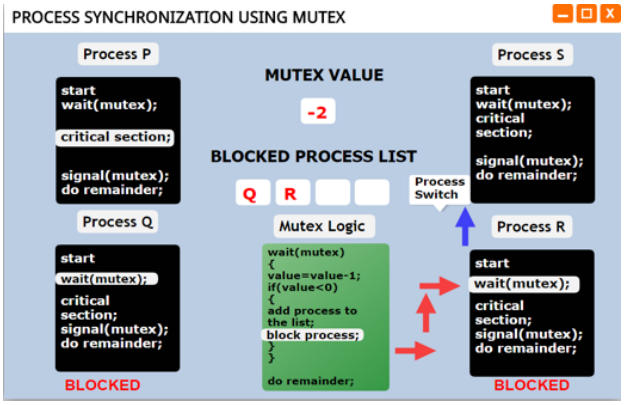

Fig. 6. Critical Selection Problem Window.

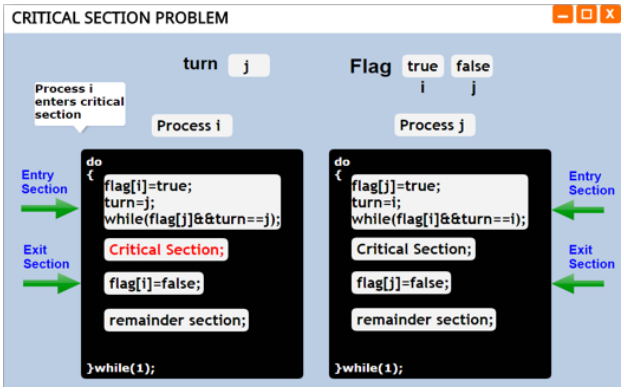

Fig. 7. Semaphores and Mutex Locks Window.

When both the producer and consumer threads try to increment the counter variable at the same time, the value is increased twice resulting in invalid counter value. This problem can also appear with multiple producers and consumers sharing a common data queue. Synchron-ITS implements multithreading techniques using POSIX threads in Java to demonstrate this problem. The counter value is incremented whenever data are produced or consumed. In this scenario, the user can choose the producer count, consumer count, and the buffer size in the UI an can define each of the parameters in real-time mode (Figure 5).

3) Critical Section Problem Window: Figure 6 shows the window demonstrating a visualization of the critical section problem for multiple processes. Each process contains a section of program code known as the "critical section," which contains the crucial operations that must be synchronized such as writing a file, updating the data tables, or modifying the shared variables. The key principle of the critical section problem is that no two processes should be allowed to execute in their critical sections at the same time. The solution is derived by designing a protocol which satisfies the requirements-mutual exclusion, progress, and bounded waiting. Synchron-ITS shows five different solutions for this problem, each with its own disadvantage, followed by an improved solution.

As shown in Figure 6, the user can compare and analyze the different solutions for this problem.

4) Synchronization with Semaphores and Mutex Locks: The critical section problem can be handled with Semaphores, a synchronized tool that can be accessed through wait () and signal () operations. The value of the counting semaphore object can be unrestricted over a domain whereas the binary semaphore ranges between 0 and 1. The critical section problem among multiple processes can be solved with shared binary semaphores, also known as mutex locks. The operations `wait()` and `signal()` decrement and increment the count of a semaphore, respectively. The value of the mutex lock is initialized to 1 and whenever the value of the binary semaphore becomes less than 1, the process gets blocked. The system traverses across different processes with the help of context switch mechanism by the CPU scheduler. However, this implementation may result in busy waiting problem, which wastes the CPU cycles. To rectify this, when the value of a semaphore is not positive, the corresponding process is blocked and placed into a waiting queue and the CPU scheduler is switched to a new process. This process is displayed in the window shown in Figure 7.

5) Producer / Consumer Monitor Solution: The main disadvantages of using semaphores are 1) they can be accessed from anywhere in the program, 2) it can be difficult to see the connection between the data values and the semaphores

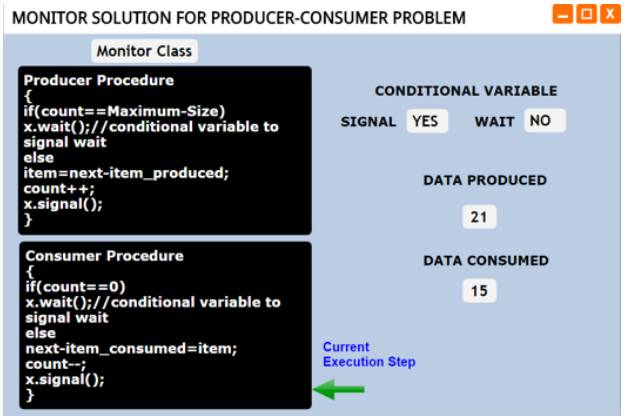

Fig. 8. Solution with Monitors Window.

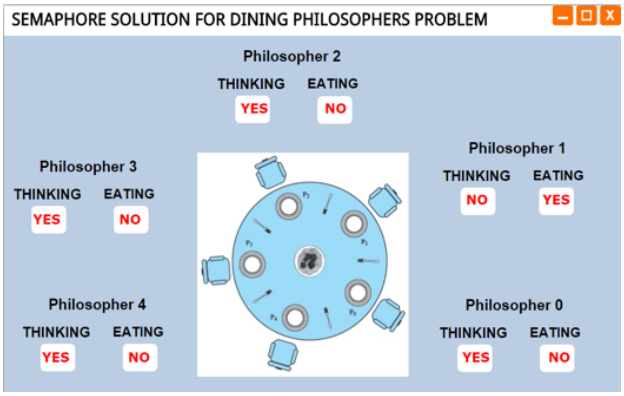

Fig. 9. Dining Philosophers Problem Window.

that controls their access, and 3) they are essentially shared variables themselves. Monitors are higher-level primitives which are introduced to handle these problems. The monitor is an abstract data type (ADT) in which multiple processes defined in the monitor are executed one at a time. However, to provide sufficient synchronization support, the implementation makes use of condition variables x and y, on which `wait()` and `signal()` operations can be invoked. When a process invokes the `wait()` operation on x, it is suspended until another process invokes a a corresponding `signal()` operation on x. This concurrency methodology is displayed in the UI screen as below Figure 8.

6) Dining Philosophers Problem Window: The dining philosophers problem is another classical synchronization problem in which limited resources are allocated among multiple processes, and these processes must have their access to the resources synchronized so they can execute without experiencing starvation or deadlock. In the problem, n philosophers (representing threads of execution) sit at a circular table with a single bowl of rice (representing a shared resource) and are equipped with n chopsticks (representing access to the resource) placed in between them. Each philosopher requires two chop sticks to eat from the bowl, they are either in the state *thinking* or *eating*, are not allowed to interact, and each must share one of the chopsticks with the adjacent philosopher.

The essence of the problem is in synchronizing access to the shared resource through the access mechanisms in such a way that none of the processes experience starvation and none end up in a deadlock state. Different solutions exist, such as the resource hierarchy solution, or the Chandy / Misra solution, but the solution for this problem can be easily implemented by using an array of shared semaphores in which each element is initialized to 1, and using the `wait()` and `signal()` operations to moderate access to the chopsticks in an orderly way. This is visualized in the Dining Philosophers Solution window as shown in Figure 9.

IV. Test Plan

The testing of the software tool is a crucial part of the Software Development Life Cycle (SDLC) where all the initial requirements have to be verified and validated. The current status of the testing activity covers only preparation and execution of the the black box test cases. The test cases solidify requirements for regression testing, integration testing, and stress testing (Table I).

TABLE I
Black-Box Test Cases

Functionality	**Expected Result**
Select concept	The corresponding window opens.
Select mode	The mode is selected.
Start algorithm	The algorithm runs with no errors.
Suspend algorithm	The algorithm pauses until the user clicks *Resume*.
Terminate algorithm	The algorithm terminates irrespective of its state of execution.
Run with valid data	The algorithm runs successfully using the values provided.
Run with invalid data	An error message appears and the algorithm does not start.

A. Testing

Various types of testing are involved in every phase of a system's development. Synchron-ITS is being developed using the following testing and evaluation procedures:

1) *Unit testing*: Validation of the proper operation of each new unit of functionality;
2) *Regression testing*: Automated tests to ensure recent developments do not affect previously implemented features;
3) *Integration testing*: Verification of proper functionality between the *UI* window, database, operating system IPC calls, etc.; and
4) *Stress testing*: Manual and automated verification that the system can handle unusual conditions or loads (such as when a user clicks on multiple concepts or operates multiple algorithms in parallel).

B. User Experiments

We plan to evaluate the quality and usefulness of the tool through user testing / human participant experiments. These

experiments are currently being designed with the help of a model called The Technology Acceptance Model (TAM) developed by Davis, Bagozzi & Warshaw [24]. This model outlines "determinants" of user acceptance of a computing technology that are "general, capable of explaining user behavior across a broad range of the end-user computing technologies and user populations, ... parsimonious, and theoretically justified" [25].

In the model, user acceptance of computer software is decided by behavioral intention, attitude toward its use, and user attitude toward the behavior (Figure 10). We used the TAM model as a software quality reference and developed a plan to test the tool using the following standard variables from the TAM model:

- Selection of participants,
- Testing procedure,
- Questionnaire,
- Perceived usefulness,
- Intention,
- Attitude toward its use,
- Subjective norm, and
- Perceived ease of use [24].

The methodology for each of the measures are under development. When completed, we plan to test the tool using undergraduate and graduate student participants from Wright State University.

C. Planned Improvements

The current state of the tool includes the basic functionality and interactive visualizations for the synchronization problems and their solutions, but there are a number of feature upgrades, such as 3D visualizations, database integration, OS integration, and user experiments, which will be developed in the development spiral. Other applications considered for Synchron-ITS are to provide it as a mobile or web-based application.

1) Database Integration: The database integration is used in two different ways as part of this project. Initially, the default values which are hard-coded in the autonomous mode may make the code cumbersome for a user who analyzes it. Instead, we planned to store the default values in a database table and have the system fetch the corresponding values through database calls during execution. Secondly, in *Self-Check* mode, the students performance can be stored in and retrieved from the database to help in evaluation of user learning over time. These are two different tables to hold this information–one containing the synchronization concept descriptions and IDs, and the other containing student data.

2) Real-Time Mode: In real-time mode, a student can drill down into the code and see actual memory changes and other behaviors on a running CPU. However, this implementation will be restricted to a suitable Linux operating system platform, either through accessing information from the host machine, or by connecting to an emulated virtual machine using virtual machine introspection or remote procedure calls.

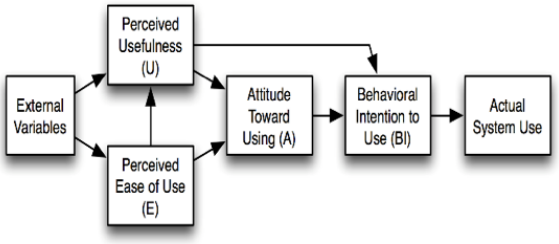

Fig. 10. The Technology Acceptance Model (TAM) from [24].

V. Conclusions and Future Research

We presented details of our working prototype of an interactive tutoring tool that visually and interactively demonstrates the main concepts of shared memory and synchronization. While developing the tool, we considered 1) the problems faced by students in learning the concepts of shared memory synchronization, 2) the variations of teaching and learning methods for presenting this material across institutions world wide, and 3) the efforts made by others in analyzing and developing OS simulators and other instructional tools. We addressed the background literature involved with presenting OS concepts to students, and discussed the design and architecture of the Synchron-ITS system along with some details of its implementation and our testing plan to verify its usefulness.

Future research that is needed (beyond the next spiral) is to determine the usefulness of using 3D representations and whether porting the system to a mobile platform can help students understand mobile and embedded operating systems.

A. 3D Visualizations

We are researching ways to expand visualizations of the synchronization data structures and algorithms into 3-dimensional representations using software frameworks such as Java3D or Unity3D to see how that can improve student learning or engagement. We also plan to compare user testing results between the 2D and 3D visual representations to evaluate their usefulness.

B. Integration in a Mobile Platform

In recent years, end-user computing has shifted from traditional desktop and server software to mobile device and embedded platforms. With the massive increase in the number of mobile device users, this tool can be used to contrast the similarities and differences of synchronization between desktop, server, mobile device, and real-time operating systems. We plan to port the shared memory synchronization tutors (particularly those in *Real-Time* mode to the Android OS as it is built on top of the open-source Linux kernel. This feature is in requirements gathering phase and will be developed in a future development spirals to be determined.

ACKNOWLEDGMENT

P. Manoj Kumar would like to thank Dr. Adam Bryant for the guidance provided in the research activities.

REFERENCES

[1] C. Clifton, "Concurrency in the curriculum: demands and challenges," in *Workshop on Curricula for Concurrency*, 2009.

[2] W. Du, M. Shang, and H. Xu, "A novel approach for computer security education using Minix instructional operating system," *Computers & Security*, vol. 25, no. 3, pp. 190–200, 2006.

[3] D. Hovemeyer, J. K. Hollingsworth, and B. Bhattacharjee, "Running on the bare metal with GeekOS," in *ACM SIGCSE Bulletin*, vol. 36, no. 1. ACM, 2004, pp. 315–319.

[4] W. A. Christopher, S. J. Procter, and T. E. Anderson, "The nachos instructional operating system," in *USENIX Winter*, 1993, pp. 481–488.

[5] D. Hettena and R. Cox, "A Guide to Nachos 5.0j."

[6] J. Dayo and C. L. Khan, "Dex-os: An aspect-oriented approach in developing an educational extensible operating system for the ibm pc and compatibles," in *Proceedings of the 4th Philippine Computing Science Congress*, 2004.

[7] J. A. C. Hermocilla, "Ics-os: A kernel programming approach to teaching operating system concepts," *Philippine Information Technology Journal*, vol. 2, no. 2, pp. 25–30, 2009.

[8] A. Hoskey, "Simple os: A component-based operating system simulator in the spirit of the little man," *Journal of Computing Sciences in Colleges*, vol. 28, no. 6, pp. 117–124, 2013.

[9] B. Pfaff, A. Romano, and G. Back, "The pintos instructional operating system kernel," in *ACM SIGCSE Bulletin*, vol. 41, no. 1. ACM, 2009, pp. 453–457.

[10] D. Jones and A. Newman, "RCOS.java: A simulated operating system with animations," in *Proceedings of the Conference on Computer-Based Learning in Science (CBLIS'01)*. Pedagogical Faculty of University of Ostrava, 2001.

[11] P. Gayet and B. Bradu, "Procos: A real-time process simulator coupled to the control system," in *Proceedings of ICALEPCS, Kobe, Japan*, 2009, pp. 794–796.

[12] A. Garmpis, "Design and development of a web-based interactive software tool for teaching operating systems," *Journal of Information Technology Education*, vol. 10, no. 10, pp. 1–17, 2011.

[13] B. Mustafa, "Visualizing the modern operating system: simulation experiments supporting enhanced learning," in *Proceedings of the 2011 Conference on Information Technology Education*. ACM, 2011, pp. 209–214.

[14] J. W. Carissimo, "XINU–An easy to use prototype for an operating system course," *ACM SIGCSE Bulletin*, vol. 27, no. 4, pp. 54–56, 1995.

[15] D. Skrien, "CPU Sim 3.1: A tool for simulating computer architectures for computer organization classes," *Journal on Educational Resources in Computing (JERIC)*, vol. 1, no. 4, pp. 46–59, 2001.

[16] M. Ben-Ari, "A suite of tools for teaching concurrency," in *ACM SIGCSE Bulletin*, vol. 36, no. 3. ACM, 2004, pp. 251–251.

[17] L. P. Maia, F. B. Machado, and A. C. Pacheco Jr, "A constructivist framework for operating systems education: A pedagogic proposal using the sosim," *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 218–222, 2005.

[18] T. Golemanov and E. Golemanova, "A teaching in operating systems tool," in *Proceedings of the International Conference on Computer Systems and Technologies*, pp. 4–5.

[19] J. E. Veenstra and R. J. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS94)*. IEEE, 1994, pp. 201–207.

[20] F. Gabbay and A. Mendelson, "Smart: An advanced shared-memory simulator–towards a system-level simulation environment," in *Proceedings of the 5th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS97)*. IEEE, 1997, pp. 131–138.

[21] D. Park and R. H. Saavedra, "Trojan: A high-performance simulator for shared memory architectures," in *Proceedings of the 29th Annual Simulation Symposium*. IEEE, 1996, pp. 44–53.

[22] Á. R. Lopes, D. A. de Souza, J. R. B. de Carvalho, W. O. Silva, and V. L. P. de Sousa, "SIME: Memory simulator for the teaching of operating systems," in *Computers in Education (SIIE), 2012 International Symposium on*. IEEE, 2012, pp. 1–5.

[23] T. J. T. F. on Computing Curricula, "Computer science, final report," 2013.

[24] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, pp. 319–340, 1989.

[25] F. D. Davis, R. P. Bagozzi, and P. R. Warshaw, "User acceptance of computer technology: A comparison of two theoretical models," *Management Science*, vol. 35, no. 8, pp. 982–1003, 1989.