If a Picture is worth a thousand words, what would an animation be worth?

Viswanathan Manickam and Alex Aravind
Department of Computer Science
University of Northern British Columbia, Prince George, BC, Canada.
E-mail: (manickam,csalex)@unbc.ca

ABSTRACT

This is an exciting time for computer scientists as we are witnessing a paradigm shift in programming. The concept of computing has long passed the definition of simple input-output mapping and evolved to include models of even intricate real-life interactive systems. From the technological front, multicore processors are about to revolutionize the way we design software systems. Therefore, for the design and implementation of software systems, we believe concurrent programming is the natural programming paradigm to adopt.

Concurrent programming is hard compared to sequential programming, as the former involves multiple threads of execution within a program. These threads often interact asynchronously with the possibility of unpredictable program behavior. However, as we increasingly witness the need for concurrent programming from both application and technological fronts, we feel the time has arrived to teach concurrent programming as part of our regular computer science and computer engineering curriculum. As concurrent programming is intrinsically hard, we need to find effective ways of teaching it to our students. The contribution in this paper is a step in that direction.

We feel that a simulation software with an animation capability would be an attractive tool to teach some fundamental algorithms and concepts in concurrent programming. We have designed such a simulator that can be used as a teaching tool to animate some simple and elegant algorithms, and get deeper into their underlying logic and behavior under various settings.

Categories and Subject Descriptors

K.3.2 [**Computers and Information Science Education**]: Computer Science Education; D.1.3 [**Concurrent Programming**]: Distributed Programming; D.4.1 [**Process Management**]: Concurrency, Mutual Exclusion, Synchronization

WCCCE'11, May 6-7, 2011, Prince George, British Columbia, Canada.

Keywords

Mutual exclusion, concurrency, visualization, simulation

1. INTRODUCTION

Program design is the core activity to be taught in computer science. Concurrent Programming (CCP) is becoming one of the important subjects in the computer science and computer engineering curriculum, because concurrency is ubiquitous in almost all the systems that we implement in software. Although concurrent programming is currently introduced as a part of operating systems courses[8, 15], we feel the time has arrived to study concurrent programming as a separate subject[4, 9, 19].

In the sequential programming model, a program is a sequence of instructions to be executed one after the other in their order. In the concurrent programming model, a program consists of one or more sequential threads of execution and they are allowed to progress independently of each other. Each of these threads is essentially a sequential program.

When two or more threads are executed on separate processors, they can progress truly in parallel. Otherwise, their execution can interleave and create the illusion of parallel execution. Since concurrent programs have sequential code as their integral components, the knowledge of designing a sequential code is generally assumed in concurrent programming courses.

Each thread in a concurrent program can be simply abstracted as an alternating sequence of local computation and interaction with other threads in the system. The interaction could be either through reading or writing of shared variables or by sending and receiving of messages. Since these interactions may alter the values in the shared resources nondeterministically, their executions need careful attention. That is, these interaction components are the critical sections in the concurrent programs and their nondeterministic executions have the potential to cause a range of issues, from simply corrupting the values of the shared resources to catastrophic system failure due to incorrect actions.

The problem of coordinating the accesses to such critical sections is fundamental in concurrent programming. The simplest version of this problem is called the critical section problem or the mutual exclusion problem and it is defined as follows. Assume a system of n independent cyclic threads competing for a shared resource R. The problem is, at any time, at most one thread is allowed to access R. The interacting code in each thread which accesses R is called the critical

section (CS). In terms of CS, the critical section problem is that, at any time, at most one thread is allowed execute the CS.

Most problems in concurrent programming have their components as the critical section problem or its variation. Therefore, concurrent programming courses typically start with the study of the critical section problem. In this paper, we propose a simulation tool that can be used to gain deeper understanding of this fundamental problem, some generalizations, and some of their interesting solutions.

1.1 How hard is CCP?

The answer to this question is obvious if the reader is familiar or has experience with concurrent algorithms. Designing concurrent algorithms, although small in code, is hard. We quote from the opinions expressed by some leading experts in the field based on their experience with concurrent programming. These quotes nicely capture the state of the complexity involved in concurrent programming.

"The program pieces for "enter" and "exit" are quite small, but they are by far the most difficult pieces of program I ever made." - E.W. Dijkstra, 1971. "I tried over a dozen ways to solve this problem before I found what I believe is a correct solution." - D. E. Knuth, 1966. "I learned that concurrent algorithms were hard to understand and still harder to get it right!" - M.J. Fischer, 2008.

CCP is hard for several reasons. First of all, as we so accustomed with sequential programming, for most of us, concurrent programming is new and anything new will appear hard in the first encounter due to lack of knowledge. Second, concurrent programs often involve complex phenomena that can arise from simple interactions among simpler parts. Third, non-determinism is inherent in the execution of a concurrent program that involves multiple threads. Finally, lack of tools and knowledge to handle concurrency adds to the complexity of understanding the threads behavior properly. Also, CCP requires to think in a way that we find it difficult. However, these difficulties do not anyway diminish the usefulness and inevitability of CCP, and therefore it is time to develop tools and knowledge so that concurrency can be understood properly and handled effectively.

1.2 Motivation and Objectives

Recent trends indicate that there is not much argument these days about the importance of teaching CCP to our future generation software designers and developers. The question is how to introduce and teach this difficult subject in an attractive and effective way so that it will not drive the students away from the computer science and computer engineering programs. We are already suffering from the problem of students recruitment and retention, and we rather need to encourage and engage them in the learning process of computer science.

Visualization is an effective technique that can make even complicated matters easy to understand. If we adopt visualization to teach some of the most fundamental algorithms in CCP, then it has the potential to attract the students easily into the subject. The clarity acquired through visualization will not only remove the initial barrier in getting into the subject but also engage them further with the interest to investigate the questions and challenges, and understanding the subject matter deeper. This motivated us to design a visual illustration tool for understanding threads interactions and executions. We refer our tool as VITTI - the acronym for Visual Illustration Tool for Thread Interactions.

VITTI has been designed with the following main objectives in mind. First, it should provide an easily understandable representation of the logic of the algorithm intended to be taught. Second, the threads and the logical objects of the representation must be animated during the execution to illustrate the behavior of the threads. Third, it should succinctly display information required to stop and analyze a particular execution state. Finally, it should illustrate some important scenarios or extreme cases of the chosen algorithm.

1.3 Related Works

Surprisingly, not many tools are available in the literature to simulate and study concurrent programs. Most popular concurrency simulator tool is BACI, developed by Bill Bynum and Tracy Camp[6]. It has a graphical user interface that displays the run time data. Its latest version called JBACI is written in Java for portability and also extended to include some more features[5]. Another tool called Convit, developed at the Tampere Univerisity of Technology, Finland, is a debugger and simulator written in Java[10]. These tools allow programs written in dialects of Pascal and C and support some synchronization primitives. They are mainly intended for analyzing the CCPs written in the language supported by the tool. These tools allow compilation, execution, and debugging. The other popular tools are DAJ designed for simulation of distributed algorithms and CPV, PVS, SPIN, JSPIN, etc., designed for verification of concurrent programs written in the language supported by the tools[5].

Our simulator is different from the above discussed simulators in a fundamental way that VITTI is not intended for concurrent program development, debugging, or verification. It is primarily designed to offer visual illustration of some interesting algorithms, and through which the students can develop further insights and ideas to solve concurrency related problems by designing their own code. To the best of our knowledge, VITTI is the first of its kind. VITTI can be used in conjunction with BACI or JBACI as tools for teaching concurrency in the shared memory systems. In the first part of the course, VITTI can be used to learn about some interesting concurrent programs. Then, in the second part of the course, BACI or JBACI can be used to develop and debug their own concurrent programs.

2. DESIGN CHOICES

When we decided to design VITTI, we had to face several basic questions and that we discuss next before presenting the simulator.

2.1 Topics Selection

First question we had to address was what to illustrate through our simulator. We needed to illustrate the most fundamental problem in concurrent programming so that its deeper understanding would motivate the students to go on to advanced topics. It was immediately evident from the literature and the popular textbooks on CCP that mutual exclusion is that fundamental and influential problem in CCP. The mutual exclusion problem appears in almost all other concurrent programming problems in various forms[4, 19, 9].

The mutual exclusion problem deals with threads accessing a shared resource in a mutually exclusive way. Teaching synchronization of concurrent threads typically starts with the mutual exclusion problem for the simplest case of 2. The solutions to this problem are generally small pieces of code and understanding them as the first concurrent program is not easy. Therefore, as it stands, understanding mutual exclusion algorithms seem to act as an initial barrier to CCP. To attract students interest into CCP, the mutual exclusion problem and its solutions must be introduced in an illustrative and enjoyable way. The approach should offer greater insight into the behavior of the threads during their coordination to access the shared resource.

After we have chosen the topic of mutual exclusion, we decided to expand the simulator by simulating some of its important generalizations. We chose to simulate l-exclusion and room synchronization algorithms. The l-exclusion problem was chosen because it is the simple generalization of mutual exclusion. The problem is that, at any time, at most l threads are allowed to access R. The room synchronization was selected because it aims to achieve seemingly conflicting two important aspects of CCP at the same time. Mutual exclusion and concurrency are two important aspects in CCP, and room synchronization basically aims at achieving mutually exclusive access to the room R while facilitating suitable concurrency among the threads of same interest.

The room synchronization problem is defined as follows. There are several forums threads can attend. Threads may be interested in attending different forums, but only one forum can be held at R at a time. However, in any forum, any number of threads interested in that forum can participate simultaneously. The problem is to design an algorithm to ensure that (i) at any time only one forum is active, (ii) any threads attempting to attend a forum will eventually succeed, and (iii) if several threads are interested in the same forum then they may be able to attend the forum simultaneously.

One of the interesting aspects of room synchronization problem is that it can be reduced to other interesting synchronization problems such as mutual exclusion, l-exclusion, and reader-writer problem as its particular cases.

2.2 Algorithms Selection

We used the following metrics to choose the algorithms to be simulated. First, the algorithm must be simple and elegant. This is very important to create interest. More complex or with boring logic may not be interesting to many students. Second, it must be generic that it does not require any special assumptions about the system. Special assumptions might force the requirement of knowledge about the specialized system, which we feel could be a distraction. Third, the algorithm must have interesting behavior. That is, it should have some behavior that must surprise or challenge the students. Finally, the set of algorithms must be comprehensive. That is, it must cover the major ideas used to design such generic mutual exclusion algorithms.

Based on the above metrics, we chose to simulate seven mutual exclusion algorithms (Dijkstra's Algorithm[7], Knuth's Algorithm[11], Bakery Algorithm[12], Symmetric Token Algorithm[2], Peterson's Algorithm[14] Queue Algorithm[3], and LRU Algorithm:[1]) and one l-exclusion algorithm and one room synchronization algorithm.

Dijkstra's algorithm was chosen for the historical reason that it is the first correct solution for n threads. It is a turn based algorithm. The basic idea is that when the turn is free, threads compete for it. The thread which captures the turn successfully is allowed to access the shared resource next. Unfortunately, due to its random nature, this algorithm is susceptible to starvation. Also, this is a good example to illustrate a starvation scenario. Knuth's algorithm was the first starvation free algorithm. This algorithm removes the random competition by setting the term deterministically, but still allows $2^{n-1} - 1$ overtakes on a thread.

Bakery algorithm, by Lamport, is the most popular algorithm for several attractive properties. It is based on a simple service strategy commonly used in bakeries (hence the name Bakery algorithm). In this algorithm, each thread chooses a token number upon entering the competition for shared resource. The holder of the lowest token number is the next one to be served. If two threads choose the same token number due to concurrency, then the thread with lowest id goes first. Bakery algorithm has a limitation that the token numbers can grow unboundedly. With a simple twist, symmetric token algorithm solves the issue of unbounded token growth.

Peterson's algorithm is a very elegant algorithm based on the idea that each thread has to pass through $n - 1$ stages to access the shared resource. These stages are designed to block one thread per stage so that after $n - 1$ stages only one thread will be eligible to access the shared resource. A thread can move to next stage only if it is either pushed by some other thread or all other threads are in stages below its own. Queue algorithm is simple and elegant algorithm that emulates the queue system used commonly in practice. It has several interesting properties. Finally, LRU algorithm assures Least-Recently-Used (LRU) Next fairness, while all popular mutual exclusion algorithms aim to assure First-Come-First-Served (FCFS) fairness property or none.

The queue based algorithm is interesting that it is easily modified to solve l-exclusion and room synchronization problems.

2.3 Basic properties

Concurrent programs have several properties beyond sequential programs. Through simulation of the above mentioned algorithms, the students must understand and able to verify the following basic properties of concurrent programs. The first and foremost property is that nothing bad should happen due to concurrency. This property is called safety property. Well, the immediate solution to satisfy this property would be doing nothing. That is, if we stop concurrency, nothing bad will happen. That is not good either. This extreme measure is rectified by the next property and that is something good should happen. This property is called freedom from lockout. This is good for the system, but need not be good for every individual thread in the system. To guarantee fairness for each individual thread, a sequence of fairness properties is specified in an incremental fashion.

The first fairness property is called freedom from starvation, which states that no thread should be denied access to R forever. The next one is putting some bound on the number overtakes on a thread, called k-bounded wait, that no thread can be bypassed more than k times to access R. The ultimate requirement is FCFS or LRU assuring LRU fairness.

Another important property is not assuming atomicity on

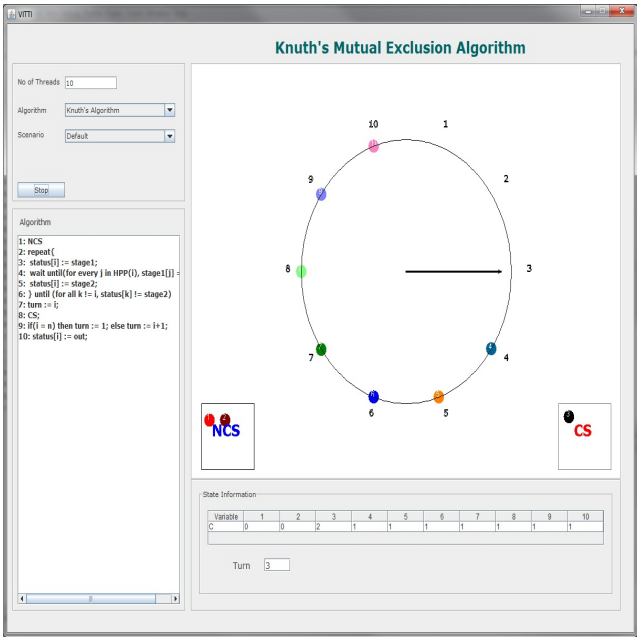

Figure 1: Animation Panel

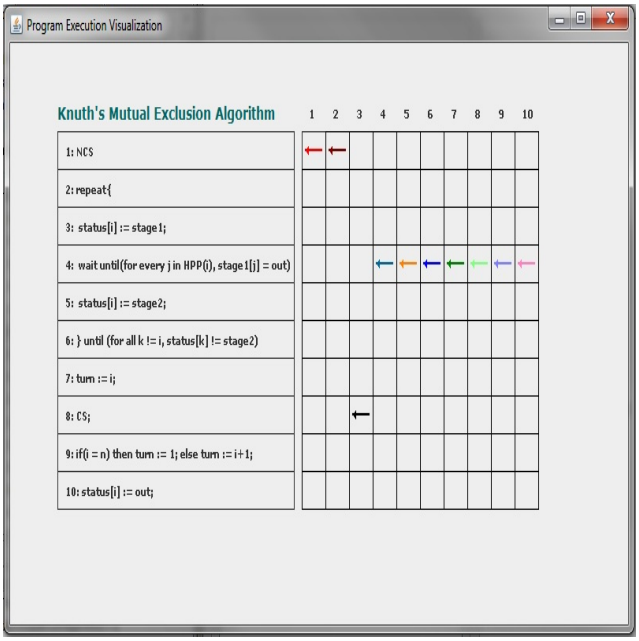

Figure 2: Animation Panel

read and write operations, and it has been referred in the literature as non-circularity property. It was widely believed that the read and write operations on an individual memory word are atomic (mutually exclusive) is a required assumption for any mutual exclusion algorithm of shared memory systems. It turned out later that the problem can be solved without this requirement. Bakery and LRU algorithms do not require read and write to be atomic and queue algorithm requires only no two writes overlap. By executing the above algorithms using VITTI under different scenarios, we believe the students can acquire deeper understanding of these basic properties.

3. VITTI

VITTI simulation tool, shown in Figure 1, is designed to teach some simple synchronization algorithms by illustrating the behavior of the threads executing those algorithms. The software has four logical components as explained next.

1. *Execution Engine:* Execution engine is the central component of the simulator. It is responsible for simulating the threads under various conditions and generate state information necessary to animate the algorithm behavior. That is, its main tasks are managing the execution of threads of a selected algorithm and providing the state information to the Animator and the graphical user interface (GUI).

2. *Graphical User Interface:* It has four panels - input panel, algorithm panel, animation panel, and state information panel. Input panel is responsible for getting the user inputs and initializing the simulation - the number of threads, selection of algorithm, and the scenario to be simulated. Algorithm panel displays the pseudo code of the selected algorithm. Animation panel provides visual effects to illustrate the behavior of the algorithm. State information panel is used to show the different states of the threads during their execution. State information panel displays the values of shared variables of all threads in the system.

   To illustrate the state of execution of different threads, we use a separate panel indicating which thread executes which instruction, as shown in Figure 2. Instead of displaying the code and execution of different threads in different panels (hard to view all of them simultaneously), VITTI uses one panel to accommodate the same. This idea we feel is unique and attractive as it has several advantageous over displaying thread executions separately.

   The arrow of each thread points to the instruction that it is executing currently. Combining this information with the display status, given in Figure 1, the students can get a deeper understanding of the behavior of the chosen algorithm.

3. *Geometry Generator:* Geometry generator is used to generate different geometry shapes to represent the objects involved in execution of the chosen algorithm. We use circle to represent threads with their thread id displayed in it. For the critical region and non-critical region, we use rectangles with their name displayed within it. The requirement of other geometrical objects depends on the representation of a particular algorithm. For example, Dijkdtra and Knuth's algorithms require a circle with positions marked and a turn pointer to move between positions. Queue algorithm requires queue positions for the threads to join and progress.

   Proper representation of the algorithm in terms of visible geometric objects is extremely cruicial to effec-

tively illustrate the behaviour of the threads. We chose a suitable model to represent and illustrate the components and the behavior of the algorithms. This is another interesting feature of VITTI.

4. *Animator:* This component shows the visual effects of the threads behavior during their execution. Animator receives the information of each thread from Execution Engine and then animates accordingly.

Although only a selected set of algorithms have been implemented for study, VITTI is designed to be modular and therefore can be extended to implement more algorithms.

4. CONCLUSION

At the least, VITTI can help teaching concurrent programming in two important ways. First, it can be used to understand some of the most important and interesting algorithms in concurrent programming. Secondly, it can be used as an interesting project component where students can be asked to come up with ways of extending the simulator and completing one. For example, a simplest project would be adding another algorithm to the simulator.

Acknowledgments

This work was supported by NSERC Discovery Grant (312162-10).

5. REFERENCES

[1] Alex A. Aravind, Yet Another Simple Solution for the Concurrent Programming Control Problem, *IEEE Transactions on PDS*, 22(6):1056-1063, 2011.

[2] Alex A. Aravind, Highly-Fair Bakery Algorithm using Symmetric Tokens, *Information Processing Letters*, 110:1055-1060, 2010.

[3] A. Aravind and W. Hesselink, A Queue Based Mutual Exclusion Algorithm, *Acta Informatica*, 46:73-86, 2009.

[4] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2006.

[5] M. Ben-Ari, A Suit of Tools for Teaching Concurrency, *ITiCSE*, 251, 2004.

[6] B. Bynum, After you, Alfonse: a mutual exclusion toolkit, *SIGCSE*, 170-174, 1996.

[7] E. W. Dijkstra, Solution of a Problem in Concurrent Programming Control, *CACM*, 8(9):569, 1965.

[8] S. Haldar and A. Aravind, *Operating Systems*, Pearson Education, 2010.

[9] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers, 2008.

[10] H.-M. Jarvinen, M. Tiusanen, and A.T. Virtanen, Convit, a Tool for Learning Concurrent Programming, *AACE E-Learn Conference*, 2003.

[11] D. E. Knuth, Additional Comments on a Problem in Concurrent Programming Control, *CACM*, 9(5):321-322, 1966.

[12] L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, *CACM*, 17(8):453-455, 1974.

[13] Y. Persky and M. Ben-Ari, Re-engineering a concurrency simulator, *SIGCSE*, 251, 1998.

[14] G. L. Peterson, Myths about the Mutual Exclusion Problem, *Information Processing Letters*, 12(3):115-116, 1981.

[15] A. Silberschatz, P. Galvin, and G. Gagne, *Operating Systems Concepts*, John Wiley, 2010.

[16] H. Sutter, The free lunch is over: a fundamental turn toward concurrency in software, *Dr. Dobb's Journal*, March 2005.

[17] H. Sutter and J. Larus, Software and the Concurrency Revolution, *ACM Queue*, 54-62, Sept. 2005.

[18] N. Shavit, Data Structures in the Multicore Age, *CACM*, 54(3):76-84, 2011.

[19] G. Taubenfeld, *Synchronization Algorithms and Concurrent Programming*, 2006.