Introducing Concurrency in CS 1*

Kim B. Bruce
Department of Computer Science
Pomona College
Claremont, CA 91711
kim@cs.pomona.edu

Andrea Danyluk
Department of Computer Science
Williams College
Williamstown, MA 01267
andrea@cs.williams.edu

Thomas Murtagh
Department of Computer Science
Williams College
Williamstown, MA 01267
tom@cs.williams.edu

ABSTRACT

Because of the growing importance of concurrent programming, many people are trying to figure out where in the curriculum to introduce students to concurrency. In this paper we discuss the use of concurrency in an introductory computer science course. This course, which has been taught for ten years, introduces concurrency in the context of event-driven programming. It also makes use of graphics and animations with the support of a library that reduces the syntactic overhead of using these constructs. Students learn to use separate threads in a way that enables them to write programs that match their intuitions of the world. While the separate threads do interact, programs are selected so that race conditions are generally not an issue.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Design, Algorithms

Keywords

CS 1, concurrency, Java, objectdraw

1. INTRODUCTION

The issue of how to effectively write concurrent and parallel programs has become increasingly important. Over the last several years, the Computing Research Association has sponsored a number of conferences on Grand Research Challenges in Computer Science and Engineering. The most recent of these, "Revitalizing Computer Architecture Research", held in December of 2005, identified four grand challenges. One of those was making parallel programming more mainstream. The report states:

> "The grand challenge is to make parallelism pervasive and parallel programming mainstream in order to enable software to make effective use of the widely available parallel hardware and continue the performance improvement trend of the past several decades. ... The goal is to make parallel programming accessible to the average programmer. *The development of parallel software should be a core component of the undergraduate computer science and engineering curriculum.*"[1]

Given the the rapid proliferation of multi- to many-core architectures and the fact that increases in processor speed are slowing, it is clear that all students need to learn parallel programming as an essential skill. In the past, programmers could count on new generations of processors to speed up their programs' execution time with little or no intervention by the programmer. However, increases in the number of processors on a chip will not necessarily speed up existing sequential programs. Programmers will have to rewrite sequential programs to take advantage of multiple processors. While hard problems will likely continue to be solved by specialized "gurus" in parallel programming, all of our students must be familiar enough with the ideas of parallel programming to solve the sort of problems they will encounter in their professional careers.

Good overviews of the importance of concurrent programming are available in recent articles by Sutter and Larus [8, 9]. The following text is excerpted from the latter article:

> "For several reasons, the concurrency revolution is likely to be more disruptive than the OO revolution. First, concurrency will be integral to higher performance. ...
>
> The second reason that concurrency will be more disruptive than OO is that, although sequential programming is hard, concurrent programming is demonstrably more difficult."

Comments such as these have helped convince many computer scientists that we are heading into (yet another) crisis of software development. However, most students do

*Research partially supported by NSF CCLI grant DUE-0088895.

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

[1]Emphasis added.

not yet obtain much exposure to high-level concurrent programming. While Computing Curricula 2001 [1] and the recent 2008 [5] revision specify a four-hour knowledge unit PF/EventDriven-Programming, few schools appear to be teaching this material in their introductory course sequences.[2] Most students likely see a brief mention of concurrency in operating systems or architecture courses,[3] while many programs provide advanced electives in parallel programming and concurrency.

We believe that an introduction to the ideas and difficulties of parallel programming can and should take place in the traditional CS 1 – CS 2 course sequence. In this paper we report on our experiences on introducing concurrency as an integral part of a CS 1 course. As explained below it is our experience that students find (simple) concurrency a very natural concept that matches well with their world view. It is more productive to take advantage of their intuition rather than trying to force program designs into a single-threaded sequential mode.

2. CONCURRENCY IN CS 1

There has been a great deal of interest in introducing concurrency in a more central way in the undergraduate CS curriculum.[4] But how early can these ideas be introduced? Does it even make sense to attempt this in the first two CS courses?

In the fall of 1999, the authors implemented a major update of the Williams College CS 1 course. With the support of the specially designed `objectdraw` library for Java, this course takes an objects-first approach, uses truly object-oriented graphics, incorporates event-driven programming techniques from the beginning, and includes concurrency quite early in the course. The authors now have ten years of experience teaching this course first at Williams and later at Pomona Colleges, and have authored a text book [3] using this approach.

Our course introduces event-driven programming on the very first day of class. Students define simple methods that respond to a variety of input events including the mouse button being clicked and the mouse being moved. Event-handling systems typically handle events sequentially. If the code executed to respond to a mouse click takes a long time, the system will not be able to react to other events until it is complete. As a result, code executed in response to such events must execute quickly or it will destroy the responsiveness of the system.

We also make extensive use of graphics and animations. The display of an animation is often triggered by an event like a mouse click. Displaying an animation, however, inherently takes a long time. Therefore, it is not reasonable to execute all of the code to produce the animation in the method that responds to the event. In particular, because screen updates are done by the event-handling thread, performing an animation in the event-handling thread will generally result in only the last frame being displayed.

The standard solution to this problem is to include code in the event handler to create a separate thread to display the animation. It is possible to design libraries that disguise this use of concurrency in performing animations by defining a type of event that occurs each time the next frame of an animation should be displayed. We wished, however, to provide an introduction that taught students the correct programming techniques so that they would not later have to learn an entirely different way of programming in standard Java.

As a result, in the third week of the semester we teach students how to create and execute separate processes. Students create classes that extend an `objectdraw` library class `ActiveObject`. `ActiveObject` is a trivial extension of the standard Java `Thread` class. The new class is necessary only because the standard `sleep` method of `Thread` throws an `InterruptedException` and we don't introduce exception handling until late in the term. The only new feature in `ActiveObject` is a `pause` method that does not throw exceptions. Students write their first simple program using separate processes in the fourth week of the term. This program implements a simple game that involves dropping a ball into a box. The `Ball` class from this program is illustrated in Figure 1.

When the ball is created, a filled oval is drawn on the screen, and the other instance variables are initialized using the parameters (omitted in the code displayed). At the end of the constructor, the `start` message is sent, which results in the creation of a new thread that begins executing the `run` method. The `run` method results in the ball falling to the bottom of the screen, where it is determined whether or not it is completely contained in a box that was passed in with the constructor.

While this assignment introduces students to the creation of threads, it is not a very compelling example of a concurrent program. While a separate thread is necessary for the animation to be displayed, there are really never two programmer-controlled threads active at the same time while this program is running. The event-handling thread awakens to launch the `Ball` thread when the user clicks and then it waits until another input event occurs.[5] Other than the creation of the thread, there is no interaction between the thread created in the `Ball` object and the main event thread. However, the `BoxBall` example provides an introduction to the mechanics of creating threads that prepares the students to complete a program that depends heavily on concurrency the following week.

In the fifth week of the term, students implement a simple version of the `Frogger`™ game (see Figure 2). For those who have not played it, this game involves guiding a frog across a four-lane highway without getting hit by a car. The program involves the creation of a separate thread for each car as well as one thread for each lane whose purpose is to periodically generate new cars.

The interactions between the cars and frog are very simple. Every time a thread associated with a car moves the car, it checks to see if the car has run over the frog. If so,

[2]Somewhat puzzlingly, the description of the knowledge unit does not mention concurrency even though concurrency is needed for non-trivial event-driven programs.

[3]There is a 6 hour knowledge unit OS/Concurrency in Curricula 2001, but it is more concerned with the support of concurrency in operating systems rather than writing concurrent programs. The Curriculum 2008 interim report expresses much concern about the growing importance of concurrency, but makes no changes in the core curriculum to reflect this importance.

[4]See for example the Workshop on Curricula for Concurrency and Parallelism at OOPSLA 2009.

[5]The event thread is busy updating the window, but this is outside of the programmer's control.

```
import objectdraw.*;

public class Ball extends ActiveObject {
   private static final int PAUSE_TIME = 50;
   private static final int DROP_DIST = 10;

   private int size;
   private FilledOval theBall;
   private Box theBox;
   private int bottom;

   // Create ball and start it falling
   public Ball(int size, Location pt, DrawingCanvas c,
               Box theBox, int bottom, Text score)  {
      theBall = new FilledOval(pt,size,size,c);
      ...
      start();
   }

   // Drop the ball to the bottom of the playing area.
   // At bottom, check if it landed in the box.
   public void run() {
      while (theBall.getY() < bottom - DROP_DIST) {
         theBall.move(0,DROP_DIST);
         pause(PAUSE_TIME);
      }
      if (insideBox()) {
         score.setText("You got it in!");
         theBox.moveBox();
      } else {
         score.setText("Try again!");
      }
      theBall.removeFromCanvas();
   }

   // Return true iff this ball is inside theBox.
   public boolean insideBox()  {
      double leftPoint = theBox.getLeft();
      double rightPoint = theBox.getRight();
      boolean leftIn = theBall.getX() > leftPoint;
      boolean rightIn = (theBall.getX() + size
                                      < rightPoint);
      return (leftIn && rightIn);
   }
}
```

Figure 1: Simplified version of Ball class from BoxBall game.

it sends a `kill` message to the frog, which results in setting its instance variable `isAlive` to false. When the user clicks to move the frog, the program tests to see if the frog is still alive. If so, then it is moved appropriately. If not, the click is ignored unless the user has clicked in a special part of the screen to reincarnate the frog.

To introduce concurrency at this early point in a student's education we must carefully select examples that limit how much students need to know about concurrency. All of our example programs are designed so that race conditions will generally not be an issue. Variables are typically only set by a single thread or are set consistently by threads with access to the variable, so the relative timing of threads does not make a difference in the results of program runs. For example, in Frogger, the fact that all a car does when it thinks it has killed the frog is set a boolean variable to be false means that interference is not an issue even if two cars detected that they overlapped with the frog's position at almost the same time.

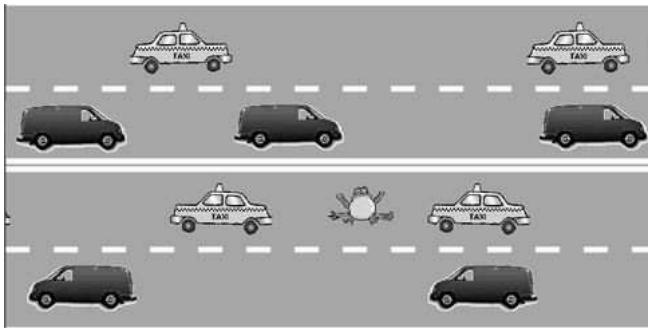

Figure 2: The Frogger game uses separate threads.

Our experience has convinced us that students find that concurrent programming matches well with their intuitions. When students look at a game like Frogger, they see the cars as independent entities. Forcing them to implement the combined motion of all of the cars in one sequential thread would seem less natural.

This is exactly what would happen if we tried to avoid introducing threads by instead using a library that provided a form of "animate" event that occurred each time a new frame of an animation needed to be drawn. All the cars would need to be moved one step each time this event occurred. To do this, the students would have to maintain an array of car positions and write a loop to update the array and the screen at each step. The resulting solution would not only seem less natural, it would in fact be more complex. In the concurrent solution, on the other hand, the class used to implement the motion of the cars is no more complicated than the `Ball` class shown in Figure 1. In particular, because each car moves independently (all cars in the same lane move at the same speed so that they don't run into each other), we are able to introduce this lab before we introduce arrays or other techniques to handle collections of objects.

If we avoid introducing concurrency early, we force students to learn to sequentialize naturally concurrent processes. If we instead introduce concurrency early using examples designed to avoid complex synchronization problems, we enable our students to appreciate the power of this approach while it still seems natural to them.

Threads are used pervasively in the course after the `BoxBall` and `Frogger` examples. In some programs presented or assigned in the course, concurrency is necessary in order to support the timely response to controls in event-driven programming, while in others it is used in support of many interacting objects and threads. Examples include labs to play the Simon game (where a separate thread is used to play a sequence of notes stored in an array) and the Nibbles™ game (where a separate thread is used to move the snake around the screen – under the control of a user who employs arrow keys to determine the direction to move). Concurrency also plays an important role in most of the final projects for the course – typically variations on games like PacMan™, Space Invaders™, Tetris™, Brickout™, Centipede™, etc.

Finally as part of our discussion of streams, students learn about web and mail servers, as well as about sockets and general client-server programming. In one of their lab assignments they write a simulator for the "Pictionary"™ game in which the program is started on two different computers. One is designated by a user as a server and the other is run as a client that connects via a socket. The user interacting with the server draws a picture which is transmitted one line segment at a time to the client, where it is drawn on a canvas. The client then sends guesses to the server until the client guesses the correct answer. While quite primitive, it gives the students a feel for how distributed computing works.

Thus by the end of our first course students have a general feel for the use of concurrent threads and simple client-server programming.

3. AN INFORMAL EVALUATION

While we worried a great deal about the introduction of concurrency during the first offering of the course in the fall of 1999, we discovered that most students found the notion of concurrency very natural, with traditionally difficult issues, such as parameter passing, overwhelming concurrency issues.

While we did not perform a formal evaluation of the material, we do have several indicators that suggest that the introduction of concurrency material was successful. At the end of the spring 2009 term we asked our students to rank our labs on educational value, difficulty, and fun. We used a scale of 1 to 5 where 1 is low and 5 is high.

Lab	Ed Value	Difficulty	Fun
Intro/eclipse	3.3	1.5	2.8
Laundry/conditionals	3.8	2.3	3.1
Magnets/class	3.4	2.7	2.8
Boxball/parameters	3.7	2.8	3.3
Frogger/concurrency	4.4	3.7	3.8
GUI	3.8	1.9	3.0
Recursive Picts	3.8	3.1	3.2
Recursive Lists	4.3	3.3	3.5
Simon/arrays	3.7	3.3	3.6
Nibbles/2-dim arrays	4.3	3.7	4.1
Datamining/strings	3.8	3.9	2.0
Pictionary/streams	3.6	2.3	3.5
TestProg 2 (**Tetris**)	3.9	4.4	3.6
Overall Average	3.8	3.0	3.3
Average/Concurrency	3.9	3.4	3.7
Average/Nonconcur.	3.7	2.7	2.9

Figure 3: Student evaluation of CS 1 labs. Labs with significant use of concurrency are in boldface.

The laboratories that involved concurrency were heavily weighted toward the end of the course, including the final test program and both array programs, no doubt impacting student ratings of difficulty. Nevertheless, we found the comparisons interesting. Because they involved some of the most complex programs, it was not a surprise that the average difficulty of the concurrent programs was distinctly above those of the others (3.4 versus 2.7). However the educational value of the concurrent programs were slightly higher than the others, while the "fun" rating was considerably higher. Interestingly, the highest scores in "fun" were also associated with higher levels of difficulty and educational value, as seen with the Frogger and Nibbles labs.

As another data point, for the final project in the 2009 spring term offering of the CS 1 course at Pomona College, students had a choice of writing a version of the Tetris game where the tetris pieces are animated by separate threads (the main event thread handles user interactions – key presses – with the game) or of writing a simplified facebook-style application that involved no concurrency. 26 of the 28 students enrolled in the course chose to do the Tetris game. The average grade on the assignment was 91.9 out of a maximum of 100. Other semesters have resulted in similar scores. The final project at Pomona in the fall of 2007 was a simplified version of the centipede game. In that game, one thread animated all of the pieces of the centipede, separate threads animated the movement of each of the bullets shot to destroy the centipede, and the main event thread responded to user interactions via the keyboard. That semester the average score for the final project was 98.4 (extra credit resulted in a top score of 103 for that assignment).

These grades, while not a perfect measure of student learning, do indicate that students were able to succeed in writing relatively sophisticated programs that use simple forms of concurrency. In particular the concurrency was used in the programs to animate motion of game components in the games and to respond to user input during this animation. Moreover, threads were able to communicate by calling methods that resulted in changes to instance variables whose values were accessed by other threads. While race conditions were avoided in the programming assignments, they were discussed in class and are tested on most final exams.

4. WHAT ABOUT RACE CONDITIONS?

As noted above, we are able to hold off on the discussion of race conditions and the need to use synchronized methods by carefully assigning programs that minimize opportunities for interference. However, as a matter of principle, we do introduce the problem of race conditions and the use of `synchronized` later in the semester as we want students to be aware of the complexities of concurrent programming.

We introduce a simple example to illustrate race conditions. We demonstrate a program to simulate two ATMs that are simultaneously accessing the same account. One ATM withdraws $100 from the account a total of 200 times, while the second deposits $100 to the account the same number of times. When the program is run, the students are surprised when the final balance of the account is different from the initial balance. They are even more surprised when running the program multiple times results in different final balances.

The difficulty lies in the (unsynchronized) method that changes the balance in response to a deposit or withdrawal.

```
public void changeBalance(int amount) {
  int newBalance = balance + amount;
  display.setText("" + newBalance);
  balance = newBalance;
}
```

The problem with this code (which is written a bit oddly in order to increase the odds that a race condition will

result)[6] is that between the first line and last line of the method, another thread can come in and reset the balance.

We then illustrate how, with only one transaction on each ATM, we can obtain a balance of the expected $1000 or the unexpected values of $900 or $1100. The unexpected values result when one process starts executing the `changeBalance` method while the other is in the midst of executing the same method – having updated `newBalance`, but not yet resetting `balance`.

The students find this a compelling example (they all interact with ATM machines regularly and tend not to think of them as behaving randomly!). A question on the final exam consistently illustrates that the vast majority of students understand the issue and how to fix it (in this case by simply making the `changeBalance` method `synchronized`).

While this doesn't address all the possible problems with concurrent programming, it sets students up to understand that there are issues, which can be explored in later courses.

5. RELATED WORK

There are many possible approaches to introducing concurrency in a CS 1 course. We discuss a few here.

Lynn Andrea Stein's approach [7] is the most similar to ours. She proposed focusing the introductory course on interactions, using concurrency. More recently that course seems to have shifted from Java to Python, while still emphasizing interaction. We have not shifted our course as dramatically as hers to focus on interaction, but instead emphasize concurrency as a natural way of modelling problems and of programming solutions to them.

Ernst and Stevenson [4] describe their efforts to integrate concurrency into their undergraduate curriculum, including CS 1. Their focus is on traditional data parallel processing where processing a collection of data is divided into several threads, each of which handles a subset of the data. Generally there is little to no interaction between threads aside from the main thread waiting for all of the subthreads to finish their computations before resuming. This is the traditional style of concurrency that shows up frequently in scientific computation, but leads to limited kinds of programs compared with those where different objects interact more significantly. Our approach is different in that we provide a more task-parallel approach where different threads are performing different kinds of actions and have some (limited) interactions while executing, rather than waiting for subtasks to terminate.

Yet another approach is exemplified by Sanders & van Dam [6], who choose to hide concurrency by using a timer class to trigger regularly-spaced events that can be handled. Students write simple methods that are designed to react to a single event. However, as these events can be generated frequently, they can trigger animations generated as the responses to these events.

Ben-Ari and Kolikant [2] describe a course on concurrent and distributed systems designed for high school students. This was a much more specialized course that introduced students to more of the difficulties of concurrency with more emphasis on topics such as mutual exclusion, deadlock, and livelock. We expect that their work will be helpful in designing follow-up courses to the one described here.

There are indeed many ways that concurrency can be integrated into an introductory course. Our own approach has the advantage of matching students' intuitions of agents interacting while executing concurrently. While we limit the interactions between threads to avoid race conditions, students learn how to use concurrent processes to solve problems in ways more natural than attempting to sequentialize the processes.

6. CONCLUSIONS

In this paper, we have described the treatment of concurrency in the CS 1 course that the authors have taught over the last ten years. Students have reacted very positively to this course and informal evidence presented indicates that students both enjoy this material and find it educationally valuable. We hope to see many more groups experimenting with how to introduce concurrency early in students' undergraduate careers so that they will learn to think of concurrency as a natural approach to problems.

With the increasing importance of many core computers, it is becoming critical that our students become more comfortable and knowledgeable about concurrency. While what we have described here is only a first step for students, and many important complications are not addressed, we believe that it ise an important first step that can be built on in more advanced courses, starting with CS 2.

7. REFERENCES

[1] Computing curricula 2001. *J. Educ. Resour. Comput.*, page 1.

[2] M. Ben-Ari and Y. B.-D. Kolikant. Thinking parallel: the process of learning concurrency. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 13–16, New York, NY, USA, 1999. ACM.

[3] K. B. Bruce, A. Danyluk, and T. Murtagh. *Java: An eventful approach*. Prentice Hall, 2006.

[4] D. J. Ernst and D. E. Stevenson. Concurrent CS: preparing students for a multicore world. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 230–234, New York, NY, USA, 2008. ACM.

[5] Interim Review Task Force. Computer science curriculum 2008: An interim revision of CS 2001. Technical report, ACM / IEEE CS, 2008.

[6] K. E. Sanders and A. van Dam. *Object-Oriented Programming in Java: A Graphical Approach*. Addison-Wesley, 2006.

[7] L. A. Stein. What we've swept under the rug: Radically rethinking CS 1. *Computer Science Education*, 8(2):118–129, 1998.

[8] H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[9] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[6]The `setText` method executed in the second line refreshes a value displayed on the screen, providing a likely opportunity for another thread to grab the processor and resume executing.