Teaching Concurrency in the Programming Languages Course

Dorian P. Yeager
Birmingham-Southern College

November 14, 1990

Abstract:

Concurrency features in Programming Languages are an essential part of a Computer Scientist's education. Each set of features presents a model for concurrent thinking and a framework for designing concurrent algorithms. A review is given of programming language features in support of concurrency, with emphasis on the important teaching points. Strengths and weaknesses of alternative languages as teaching vehicles are discussed, and suggestions are given of appropriate examples and exercises.

1 Introduction.

A course in Programming Languages is a difficult course to design because of the great breadth of the subject. Some major questions to be answered in constructing such a course are as follows.

- Which programming paradigms should be emphasized?
- How many languages should be studied, and in what depth?
- Should the emphasis be on language design or language implementation?
- Should the case studies be of actual languages or of smaller languages created for the purpose of teaching a particular paradigm [Ledgard and Marcotty]?
- How much should be said about formal languages and translation issues?
- Should the major organization of the course be around a series of case studies [MacLennan] [Peterson] or around a methodology for classification of language features [Dershem and Jipping] [Pratt] [Tennent]?
- To what extent should the study of programming environments be included?

With these difficulties in mind it is not surprising that a full discussion of language features in support of concurrency is not often seen in the Programming Languages course. In fact, concurrency as a topic is likely to be seen only once by an undergraduate Computer Science major, in the Operating Systems course. It may even be that there is some treatment of languages for concurrency in that course (see, for example, [Peterson and Silberschatz, pp. 393-400]). But an effective argument can be made for the position that such treatment belongs in a course in Programming Languages.

The topics which should be considered essential in a given core course in Computer Science are those (a) which every Computer Science major should see, and (b) which are not customarily covered in any other but the given course. It is certainly important for every Computer Science major to see concurrency models in programming languages. Also, since the primary vehicles used for teaching programming to Computer Science majors are the Pascal and C languages, the student's ordinary experience does not include concurrency at the programming language level. Pascal has no concurrency primitives, and the Unix functions fork(), exec(), etc., are not actually C language primitives, but rather they are operating system features for which an interface has been provided in versions of C supplied for Unix systems. Exposure to such features is often provided in the Operating Systems course, but the Unix system calls by themselves do not give a full picture of the alternative design issues to be resolved when

incorporating concurrency features into a language.

2 Review of concurrency features.

The advantage of using a programming language to communicate a concept is that the student's act of internalizing a semantic mapping from the relevant syntactic structures to the abstract domains defined by the concept has far better mnemonic value than the direct study of the domains themselves. That is why students in the first course in Computer Science learn better if a programming language is taught in parallel or required as a prerequisite. In the same way, concurrency concepts are more easily digested in the context of a programming language. Let us review some of the major concurrency concepts and make comments as we go concerning which languages might be suitable for communicating those concepts.

2.1 Coroutines and tasks.

The most basic concurrency concept is that of the coroutine. Coroutines are fundamental to the study of concurrency because a coroutine carries with it a local context which progresses from an initial state at coroutine creation to a final state at program termination, and that context is not destroyed when control leaves the coroutine. In other words, once a coroutine is activated its lifetime is independent of that of its caller. However, true concurrency is not possible in a language which uses the coroutine model of a process, because that model depends on having the coroutines explicitly relinquish control to each other. There is in fact only one thread of control which is woven into and out of each coroutine in turn.

Still, coroutines can be used to good advantage as a mechanism for getting students ready for the idea of concurrent processes. Modula 2 provides a facility for dynamically creating a coroutine from a parameterless procedure, using a chunk of heap storage for environment space. The coroutine is not the procedure, but is created from the procedure and placed in a *process variable*. The process variable is the language's representation of the coroutine's context. The value stored in the process variable is a *(code pointer, environment pointer)* pair. The Modula 2 "TRANSFER" operation provides for the transfer of control from one "process" to another. Thus the state of the program as execution begins is with one process active, called the main process, and when new processes are created they are not destroyed until program termination.

A modification of the coroutine concept produces the *task* concept. When a task is initiated, it immediately begins execution in parallel with its caller. (The parallelism may be real or it may be simulated by a time- sliced scheduler. The language typically cannot differentiate between the two.) PL/I and Ada both incorporate the task concept but PL/I's facility is old-fashioned and error-prone.

Although Modula-2 does not have a tasking facility as such, its coroutines can be made to behave like tasks using a scheduler. To achieve this, Modula 2 provides a modified form of coroutine transfer, called "IOTRANSFER", which is a facility for replacing hardware interrupts, and it also provides a facility for masking interrupts upon entry to a module. On MS-DOS machines, which allow access to the timer interrupt, these features allow a time-sliced scheduler to be used to schedule the invocation of coroutines. This is an especially valuable educational tool, since the code for the interrupt handler and scheduler is all in Modula 2 and available for the student's inspection.

Like the Modula 2 process, Ada's task is a true data type and all the other structuring principles of the language are orthogonal to it. This means that one can build arrays or linked lists of tasks, or a task could be a component of a record. Ada provides the additional capability to statically declare a task so that the code which implements the task is truly identified with one and only one task. This gives Ada somewhat of an edge in the teaching of tasks, since a student can be given a sample piece of code in which the various players in the example application can be geographically identified.

2.2 Synchronization and Communication

When the language includes facilities for tasking it should also include facilities for synchronization and communication between tasks. Whereas coroutines can synchronize their actions via the pattern of "resume" calls, tasks must be provided with special synchronization primitives. Also, coroutines can communicate via shared variables without running into a "critical region" problem, whereas tasks must either avoid using shared variables or rely on

language facilities for identifying critical regions and protecting the variables involved from simultaneous access.

The "semaphore" data type [Dijkstra], or some variant thereof, is often found in languages that support concurrency. Algol 68 has a semaphore data type [Tanenbaum, p. 190], and PL/I has its event variables, which are essentially binary semaphores [Pollack and Sterling, p. 568]. Modula 2 comes with a library module providing a "signal" data type. The prototype version of this module is found in [Wirth, p. 129], but there are many variations. The signal is not actually a semaphore, since it serves coroutines, not tasks, but it is easy to modify the basic signal operations so that when a time-sliced scheduler is added the signal behaves like whatever flavor of semaphore one desires. Ada does not have semaphores as such, but it is a simple matter to write an Ada package to implement a semaphore data type.

Although critical region protection is always available by means of semaphores, the student should be strongly impressed with the fact that relying on semaphores for critical region protection is akin to relying on goto's and labels for sequence control. The semaphore is actually a synchronization feature, not a critical region feature, so its design does not address the problem directly. If the standard protocol is used whereby a P() operation begins the critical region and a V() operation ends the critical region, there is no way that the language can enforce protection; the programmer has that responsibility. In the same way that hardware conditional jumps are used to implement higher level control structures, so are semaphore operations the basic tools for constructing higher level features. Thus semaphores should be viewed as an implementation technique rather than a high-level mechanism. Devices such as monitors [Brinch Hansen] and conditional critical regions [Hoare] are important tools in the design of high level concurrency features. Their implementation is clearer once the lower level concept of a semaphore is introduced and explained.

The extreme position taken by languages such as Smalltalk, namely that shared memory should be avoided as much as possible and that task communication should normally be accomplished via message passing, is akin to Wirth's position that the accessing of subscripted variables should always be accompanied by a range check. The security thus obtained is paid for by a dramatic increase in overhead. Facilities like message passing and remote procedure call are indeed more secure means of communication, but they require all the usual stack maintenance of procedure calls and parameter passing, as well as the maintenance of buffers to simulate the needed communication channel. Students need to be made aware that this design choice is simply another example of the tradeoff between security and execution time efficiency.

The Ada rendezvous has crystal-clear semantics and is a very powerful facility for task synchronization and communication. Its main problem is that it requires that two tasks be synchronized in order to communicate. The rendezvous dictates that there be a "calling task" and a "called task" and the two must participate in the rendezvous together. The two control flows merge. To the calling task, the rendezvous looks much like a procedure call. To the called task, a rendezvous takes place when control flows into a local scope defined by an "accept" statement. When the called task exits the accept statement the two flows of control split back into the original two tasks. To achieve true asynchronous communication there must be an intermediary task to buffer the transferred information.

3 The Relationship between Concurrency and ADT's.

Tasks add a new dimension to the teaching of the concept of the abstract data type (ADT). For example, an ADT which provides storage allocation and deallocation primitives might come with a garbage collection task that runs in parallel with the clients of the ADT. The garbage collection task is viewed as a "hidden" part of the ADT, initiated during the execution of the package's (or module's) startup code; its code can be examined (or written) by the students for their edification. Even in a language like Modula 2, which does not have sophisticated task communication facilities, the interface between tasks may be defined and presented to the client module as a set of procedures. The ugly details of coding needed to manage communication of information, critical regions, and synchronization can be hidden in the implementation module.

4 Classroom Examples and Assignments.

It can be somewhat of a challenge to come up with well motivated examples of concurrent programs to present in class. The bounded buffer problem [Peterson and Silberschatz, page 345] makes a good first example. Other classic problems like the readers/writers problems and the dining philosophers are interesting and relevant in the context of an Operating Systems course, but the issues they address (deadlock, starvation, Bernstein's Conditions) are not the central teaching points in a Programming Languages class.

To get the basic idea of concurrency across, a set of random walkers (each represented on the screen by a single character unique to that particular walker) can be made to tour across the screen, taking care not to run into each other. One shared variable is needed here, a data structure to model the screen. A list of screen positions currently occupied by all of the walkers is one option, or a two-dimensional boolean array. Every reference to the screen data structure and every screen read or write operation must be enclosed in a critical region, but the walkers are completely autonomous otherwise. Since the walkers all use the same logic, one can strongly push the point that there is a single task type and all walkers are simply instances of that type.

If a good windows library is available, the students can be asked to program some simple interactions between tasks represented as windows on the screen. For example, task one generates random numbers, displays them in its window, and passes them on to task two, which sorts them by fours then displays and passes the sorted 4-tuples to task three. Task three merges the 4-tuples to get sorted 8-tuples, displays the 8-tuples, and passes them to task four. Task four maintains a sorted linked list of numbers and updates that list by merging it with the incoming sorted 8-tuples. Task four displays each number as it is inserted, along with the portion of the list where it is being inserted. All four tasks maintain a display of the number of numbers they have processed at any given time.

If students understand the problems of multiple tasks contending for fixed resources, then some more interesting kinds of things can be done. Each of two or more windows may be made to maintain a cash drawer with some random initial assignment of twenty, ten, five, and one-dollar bills. The windows then can present requests to each other to make change, with the eventual goal to obtain at least one of each denomination of bill. A window with six or more ones and no fives might request to exchange five ones for a five, but would have its request ignored until a corresponding window requested change for a five. Thus each window has to present its request, wait a judicious amount of time, then perhaps try another request. Of course, deadlock and starvation can occur here.

In [Liss and McMillan], a class project is described based on a simple game of skill. The teaching point is the abstract data type, and the motivation comes in large part from competition between students. Each team of students writes a strategy module for the game and a tournament is held which competes one against the other. This kind of experience is very rewarding for the students and much can be taught in this way about abstract data types both from the designer's point of view and from the client's point of view. On the other hand, most games do not make very good concurrency exercises because the purely internal activity of planning strategy is the only asynchronous activity that goes on.

There is a very interesting class of card games, however, where much concurrent activity takes place. These are the games in which the players engage in asynchronous "swapping" of cards. In the popular game called "Spoons", two or more people sit in a circle and the one who is designated the dealer deals four cards to each person, face down. The dealer places the rest of the cards to his/her right, face down. The players pick up their hands, and the dealer then begins to try to collect four cards of the same face value ("four of a kind") by drawing cards from the right and discarding them to the left. The player on the dealer's left can begin doing the same thing as soon as the dealer makes the first discard. At no time does any player have more than five cards in his/her hand. The score is kept in an interesting manner. In the center of the ring of n players is a pile of $n-1$ spoons. As soon as any player achieves four of a kind she or he is allowed to pick up a spoon. As soon as a spoon is picked up, all players are allowed to pick up a spoon. Of course, one player fails to get a spoon and thus has "lost" that hand. The loser gets a mark against him/her, the deal passes to the left, and play continues until all players except one have six marks (enough to spell "spoons"). Whenever a player receives six marks, that player exits the game immediately, taking a spoon with her/him.

The above game was the basis for two successful exercises in a graduate level course in Programming Languages at the University of Alabama. A version of Modula 2 for the IBM PC was used, along with a modified "`Process`" library module, which provided a time-sliced scheduler and a `SIGNAL` data type (semaphore), among other things. In the first exercise, students were given the definition module "`SpoonsManager`" of figure 1, along with a "`Cards`" module which provided a playing card abstract data type with limited operations, and told to use the facilities of the provided modules to program one hand (strategy module) of a spoons game. They tested their strategy modules by informally competing them against one another. The `SpoonsManager` module takes care of the dealing, display, and windowing functions. All that a strategy module really does is obtain a player ID from the `SpoonsManager`, then enter an infinite loop. Inside the infinite loop, a player requests cards from the `SpoonsManager` then enters an inner hand-playing loop which is exited when a spoon is obtained or no spoons remain. An example strategy module is seen in figure 2.

In the second exercise, students were split into teams and told to write the implementation code for the `SpoonsManager` module. They were given an expanded version of the `Cards` module, and they were responsible for providing safe access to shared variables and maintaining sufficient security to see that no player module "cheated" in any way.

Both exercises were quite successful. All students completed exercise 1, and all teams successfully completed the spoons manager. The students enjoyed the experience and indicated that they understood concurrency much more thoroughly after having applied the essential concepts in an actual simulation.

5 Conclusion

Concurrency features are a necessary part of a course in programming languages, and teaching them there provides an excellent opportunity to help the student to think in terms of concurrent processes. A well-chosen set of exercises can be used to show the student the naturalness and usefulness of the task concept.

References

1. Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, (June 1975), pp. 199-207.

2. Dershem, H. L., and M. J. Jipping. *Programming Languages - Structures and Models.* Belmont, CA, Wadsworth, 1990.

3. Dijkstra, E. W. "Cooperating Sequential Processes", in *Programming Languages*, ed. F. Genuys. London, Academic Press, 1968.

4. Hoare, C. A. R., "Towards a Theory of Parallel Programming", in *Operating System Techniques*, ed. C. A. R. Hoare and R. H. Perrot. London, Academic Press, 1972.

5. Horowitz, Ellis B., *Fundamentals of Programming Languages.* Rockville, MD, Computer Science Press, 1983.

6. Ledgard, H., and M. Marcotty, *The Programming Language Landscape.* Chicago, Science Research Associates, 1981.

7. Liss, Ivan B., and T. C. McMillan, "An Example Illustrating Modularity, Abstraction, and Information Hiding Using Turbo Pascal 4.0", *SIGCSE Bulletin*, Vol. 21, No. 1 (February 1989), pp. 93-97.

8. MacLennan, B. J. *Principles of Programming Languages - Design, Evaluation, and Implementation.* Second Edition. New York, Holt, Rinehart, and Winston, 1987.

9. Peterson, J. L., and A. Silberschatz, *Operating System Concepts.* Second Edition. Reading, MA, Addison-Wesley, 1985.

10. Peterson, W. W. *Introduction to Programming Languages* Englewood Cliffs, NJ, Prentice-Hall, 1974.

11. Pollock, S. V., and T. D. Sterling, *A Guide to PL/1.* Second Edition. New York, Holt, Rinehart, and Winston, 1976.

12. Pratt, T. W., *Programming Languages - Design and Implementation.* Second Edition. Englewood Cliffs, NJ, Prentice-Hall, 1984.

13. Tanenbaum, A. S., "A Tutorial on Algol 68", *Computing Surveys*, Vol. 8, No. 2 (June 1976), pp. 155-190.

14. Tennent, R. D. *Principles of Programming Languages.* Englewood Cliffs, NJ, Prentice-Hall, 1981.

15. Wirth, N. *Programming in Modula-2.* New York, Springer-Verlag, 1982.

```
DEFINITION MODULE SpoonsManager;

FROM Cards IMPORT
     FaceValueType, CardType,
     SuitType, HandType;

TYPE IDType;

PROCEDURE StartPlayer(
   PlayerProc: PROC;
   PlayerName: ARRAY OF CHAR
);
(* Starts a player process for
   the 'Spoons' game.  May be
   called up to eight times.
   The 'PlayerProc' parameter
   contains the player logic
   for this process.  The Name
   parameter is used by the
   Spoons Manager to identify
   the process on the screeen.
*)
PROCEDURE SelectCard(
   VAR Card: CardType;
   Hand: HandType;
   Face: FaceValueType;
   Suit: SuitType;
   VAR Found: BOOLEAN;
   id: IDType
);
PROCEDURE GetID(
   PlayerProc: PROC;
   VAR id: IDType
);
PROCEDURE GetSpoon(
   Hand: HandType;
   VAR GotSpoon: BOOLEAN;
   VAR SpoonsRemain: BOOLEAN;
   id: IDType
);
PROCEDURE GetCardsFromDealer(
  VAR Hand: HandType;
  id: IDType
);
PROCEDURE CardWaiting(
   id: IDType
): BOOLEAN;

PROCEDURE TakeCardFromRight(
   VAR Card: CardType;
   VAR Hand: HandType;
   id: IDType
);
PROCEDURE PassCardToLeft(
   VAR Card: CardType;
   VAR Hand: HandType;
   id: IDType
);
PROCEDURE NumberOfPlayers(
): CARDINAL;

PROCEDURE ManageSpoonsGame;

END SpoonsManager.
```

Figure 1. Definition Module for the Spoons Manager.

```
IMPLEMENTATION MODULE Yeager;

FROM Cards IMPORT FaceValueType,
   HandType, CardType, FaceCount;

FROM SpoonsManager IMPORT IDType,
   GetID, GetSpoon, TakeCardFromRight,
   CardWaiting, PassCardToLeft,
   GetCardsFromDealer, SelectCard;

PROCEDURE play;
VAR
  id: IDType;
  GotSpoon, SpoonsRemain,
              FoundCard: BOOLEAN;
  Hand: HandType;
  Card: CardType;
  FaceValue: FaceValueType;
BEGIN
  GetID(play,id);
  LOOP
    GetCardsFromDealer(Hand,id);
    GotSpoon := FALSE;
    REPEAT
      REPEAT
        GetSpoon(Hand,GotSpoon,
                 SpoonsRemain,id);
      UNTIL CardWaiting(id) OR
            GotSpoon OR
            NOT SpoonsRemain;
      IF NOT GotSpoon AND
             SpoonsRemain THEN
        TakeCardFromRight(Card,
                          Hand,id);
        FaceValue := deuce;
        LOOP
          CASE FaceCount(FaceValue,
                         Hand) OF
          |  1: SelectCard(
                  Card, Hand,
                  FaceValue, anysuit,
                  FoundCard,id);
                EXIT;
          |  2: SelectCard(
                  Card, Hand,
                  FaceValue, anysuit,
                  FoundCard,id);
          |  4: GetSpoon(
                  Hand,GotSpoon,
                  SpoonsRemain,id);
          END;
          IF FaceValue = Ace THEN
            EXIT
          ELSE
            INC(FaceValue)
          END
        END;
        IF NOT GotSpoon THEN
          PassCardToLeft(Card,
                         Hand,id);
        END
      END
    UNTIL GotSpoon OR
          NOT SpoonsRemain;
  END
END play;

END Yeager.
```

Figure 2. Example implementation module for player strategy.