BACI DEBUGGER: A GUI DEBUGGER FOR THE BACI SYSTEM

David Strite and Linda Null
Computer Science Department
Penn State Harrisburg
Middletown, PA 17057
djs352@psu.edu and lnull@psu.edu

Abstract

Due to the increasing importance of concurrent programming and distributed computing systems, possessing a good understanding of concurrency and its impact on process synchronization is essential. Since concurrency introduces design and execution issues not found in sequential programming, to learn about concurrency issues, it is important that students gain hands on experience actually doing concurrent programming. The best way to get this experience is by using a system developed specifically for teaching concurrent programming. The Ben-Ari Concurrent Interpreter (BACI) is such a system. Unfortunately, when errors are encountered in BACI programs, they are cumbersome to debug. The purpose of this paper is to introduce and describe a Java-based graphical user interface debugger for the BACI language.

1 INTRODUCTION

Concurrent programming and process synchronization are topics that are a part of every computer science curriculum. With the increasing importance of parallel and distributed processing, a thorough understanding of process synchronization is becoming essential. For students to really understand these topics, they must have an opportunity to try these concepts for themselves. Although many environments exist for hands-on practice with concurrent programming, the learning curve is often higher than acceptable for students in operating systems classes. BACI is a system designed to give students experience in concurrent programming while minimizing the learning curve. Although BACI is good for teaching concurrent programming, there has been no user-friendly environment for debugging these concurrent programs. This paper introduces and describes a GUI debugger for BACI to further assist students in learning about concurrent programming and process synchronization.

2 TEACHING CONCURRENT PROGRAMMING AND PROCESS SYNCHRONIZATION

ACM Computing Curriculum 1991 [1] and the new Curriculum 2001 [1] both recommend the introduction of concurrent programming into the computer science curriculum. There are several reasons why a computer science student should be interested in concurrency:

- The execution time for a particular calculation can be decreased if the calculation is broken into separate parts that can be run in parallel on multiple processors.
- The overall throughput of a system can be increased, because the CPU can be executing a process while another process is waiting for a resource.
- Some problems are more naturally solved with concurrent processes, especially simulations and real-time systems. Two examples are simulating the flow of customers in a supermarket and air traffic control systems.
- With multiple threads of control, it is easier to design a reactive system, where an action is triggered when a certain event occurs.
- Concurrent programs give greater control to the programmer than sequential programs, because various programs can be suspended and resumed.
- Understanding concurrency is important in understanding many topics related to computer architecture, especially pipelining and super scalar computers.

The topics of concurrent programming and process synchronization are typically taught in an operating systems class [23][24]. Currently, there are two approaches one can use when teaching concurrency: 1) use a completely theoretical approach, where the conceptual underpinnings are presented but little or no programming is required; or 2) integrate practical experience, where students do significant programming. It is quite difficult to fully appreciate process synchronization issues by simply reading over problems, as in the first approach. When taught using this method, concurrency will be little more than abstract, theoretical concepts to students. Because students need experience actually writing code to reinforce their understanding of these concepts, the preferred method is to actually require the students to write concurrent programs.

There are several ways in which students can get hands-on experience with concurrent programming: (1) use a programming language that supports concurrency constructs; (2) work with operating system calls in an operating system that supports concurrency constructs; or (3) use an environment designed for teaching concurrent programming. The first option requires the use of a language such as Concurrent Pascal [4][13][15], Ada [6][18][27], Modula [26], SR [2][12], or Java [17]. Depending on the background of the students, one or more new programming languages would need to be learned before concurrent programming could begin. The second option requires students to learn the details of a specific operating system [14]. Systems such as Nachos [21], MINIX [25], KMOS [20], OSP [16] or Topsy [11] may require specific hardware in addition to the knowledge necessary to install and run them, and often the instructor must commit to structuring the course around the specific system. The third option is to use an environment designed specifically for teaching concurrent programming. Such an environment should support the common concurrency constructs and should be similar to a programming language that most

students already know. The Ben-Ari Concurrent Interpreter (BACI) is such an environment designed for teaching concurrent programming [7].

3 BEN-ARI CONCURRENT INTERPRETER

The Ben-Ari Concurrent Interpreter is a system designed to allow students to write and experiment with concurrent programming problems. The system supports two programming languages: one is based on Pascal and the other is based on C++ (referred to as C--). Both of the BACI languages support a subset of their respective languages and are enhanced with additional features for concurrent programming. Since most students have experience with either Pascal or C++, the majority of students can easily use BACI. BACI supports several synchronization techniques, including general semaphores, binary semaphores, and monitors. In addition to allowing these process synchronization constructs, BACI provides additional features that allow students to implement their own synchronization constructs.

The original description and implementation of this system was provided by Ben-Ari in his book *Principles of Concurrent Programming* [3]. In this original implementation, Concurrent Pascal was the only language supported and semaphores were the only process synchronization construct supported. The system was later enhanced by Bynum and Camp to its current form, which supports Pascal and C--. Support was also added for monitors and binary semaphores [7].

It is important to note that the two BACI languages support only a subset of their respective languages [9][10]. No files are allowed except for standard input and output. A limited set of standard data types is allowed. Pascal allows only `integer`, `char`, and `boolean`. C-- allows `int` and `char`. A special `string` type has been added to both languages to support character strings. A set of functions similar to the standard C string functions is defined in both languages to perform common string operations. These simplifications slightly limit what can be done with a BACI program, but it is generally not a problem for the types of programs written when learning about concurrent programming [22].

BACI defines concurrent programming constructs in both languages. A `cobegin`/`coend` block is a special block that is defined in the main program. Each of the function/procedure calls within this block starts a new concurrent process. When the end of this block is reached, the main process is paused and the concurrent processes are executed using a random preemptive scheduler. When all of the concurrent processes complete, the main process continues execution. Figure 1 shows the syntax for a `cobegin`/`coend` block in both Pascal and C--. In both examples, a process is created for `foo1`, `foo2`, and `foo3`. These three processes then run concurrently.

Pascal

```
COBEGIN
  foo1();
  foo2();
  foo3();
COEND;
```

C--

```
cobegin {
  foo1();
  foo2();
  foo3();
}
```

Figure 1: COBEGIN block example

BACI also defines `semaphore` and `binarysem` data types for declaring semaphore variables. The functions `initialsem`, `wait`, and `signal` are defined for semaphores. The `initialsem` function is used to set the initial value of the semaphore. The `wait` and `signal`

functions perform the standard wait and signal operations on the semaphore. The data types `monitor` and `condition` are defined to support the declaration of monitors and condition variables. Monitor variables can only be declared as global variables and condition variables can only be declared within a monitor. The functions `waitc` and `signalc` are defined for waiting and signaling on a condition variable. There is also an `empty` function to check a condition variable queue.

Additional synchronization constructs are also supported. A function declared with the `atomic` keyword cannot be preempted, meaning that the function will be executed in its entirety before a process swap can occur. The `suspend` function puts the process calling the function to sleep. The `revive` function wakes up the specified process. The function `which_proc` returns the process number of the current process. These additional low-level constructs allow users to create their own concurrency constructs.

To use the BACI system, a user runs either the Pascal or C-- compiler on the file containing the program's source. The compiler takes this source code file and produces a file containing pseudo-machine code (P-code). As with real machine code, multiple P-code instructions may be needed for each source statement. The user then executes the BACI interpreter on this P-code file. The interpreter executes the P-code instructions. During the execution of the P-code file, the interpreter simulates the execution of multiple processes using a random preemptive scheduler. This means that multiple executions of the same program with concurrent processes can produce different results. The nondeterminism exhibited in interleaved schedules also makes locating programming errors difficult because subsequent runs of a program may produce different results.

4 THE BACI GUI DEBUGGER

The BACI interpreter includes a basic command line debugger. This command line debugger performs the basic functions of a debugger. Like most command line debuggers it is cumbersome to use and not very user friendly. Our BACI GUI Debugger allows users to run their BACI programs in an interactive graphical environment. This debugger replicates the functionality of the interpreter and adds the functionality of a GUI based debugger. The debugger displays the P-code instructions as they are executed as well as the source code that produced the P-code. Programs written in both Pascal and C-- are supported.

The BACI system is available for multiple operating systems, including several Unix flavors and DOS (including Windows). Since BACI can be used on these various operating systems, it was necessary to design our debugger for compatibility on all of these systems as well. The BACI GUI Debugger is written in Java so that it can be run on any platform on which the BACI system runs. A Java 2 runtime (version 1.2 or higher) is required. The debugger requires, as input, a P-code file; in addition, any source code files that the compiler used to create that P-code file must reside in the same directory, as the debugger must access these files as well.

The debugger allows users to control the execution of the program. Users can set the interpreter to execute the program without pausing or they can step through the code. Stepping is supported at both the source code statement level and the P-code instruction level. Breakpoints are supported to allow the user to specify a specific point for the interpreter at which to pause execution. Breakpoints can be set on specific lines of source code or on specific P-code instructions. Users can also view

the values of variables as the program executes. In addition, users have the option of turning off the random number generator that provides random schedules; this allows the users to debug the same interleaved schedule multiple times.

5 USING THE BACI GUI DEBUGGER

To use the BACI GUI debugger, one must first compile a source code program to produce a P-code file. The source and P-code files must reside in the same directory. The BACI GUI debugger, which is freely available as a Java jar file at `cs.hbg.psu.edu/~null/baci`, can then be started. To start the debugger, simply type the following on the command line (assuming a Unix environment):

```
java –classpath baci.jar baci.gui.Debugger &
```

A P-code file must then be loaded into the debugger. Once a P-code file is selected, the debugger will read in the P-code file and the corresponding source files. The information regarding which source files created the P-code file is stored in the P-code file. These source files must be in the same directory as the P-code file. The interpreter is initialized with the data read from both the P-code file and the source files. At this point, the program will be ready to run.

The main screen presented to the user is a multiple-document interface window that contains sub-windows. The advantage of this type of screen design is that the user can open, close, move, and resize the sub-windows to display the data he is interested in seeing. Multiple types of sub-windows are available, including: Process, Globals, Console, and History sub-windows.

When first started, the BACI GUI debugger will show a screen with a process list in the left windowpane. This process list contains entries for the processes that are currently running. When the interpreter is initialized, the process list will contain an entry for the main process only. As concurrent processes are started, new entries are added to this list. Selecting an entry in the list and clicking on the open button will open a Process sub-window for the selected process. (The entry can also be double-clicked to open the sub-window.) One of these sub-windows can be displayed for the main process and for each of the concurrent processes. BACI currently has an 18 process maximum, but the debugger is not limited in the number of processes it can display.

The data in a Process sub-window is divided into three tabs. The first tab, the Code tab, contains the source code and the P-code currently being executed, as well as the value of local variables for that block of code. The source code file is displayed and the source statement currently being executed is marked by a green arrow. The P-code generated by the current source line is displayed with the current instruction marked as well. The add and remove buttons in the source code and P-code areas are used to add and remove breakpoints at selected lines of source code or P-code. When a breakpoint has been set, a red circle will be displayed beside the appropriate source statement or P-code instruction. The bottom portion of the Code tab displays values of variables in the current process (the only exception to this is for main: main has a separate Global window for variables). If the process has arrays of values, these arrays can be expanded to see all entries.

The second tab of the Process sub-window, the Console tab, displays all output written to standard output by this process. The third tab, the Details tab, contains low-level details about the process. This tab displays the process stack, which is used to store the value of variables; it also provides temporary storage used by the

interpreter. The bottom and top fields define the range of the current stack frame. The active field is true if the process is in a runnable state and false if it is suspended. The finished field is true if the process has finished executing and false otherwise. The suspend field will contain the address of a variable on which the process is waiting. This could be a semaphore, a monitor, or a condition variable. The monitor field will contain the address of a monitor if the process is currently executing within a monitor. The priority field will contain the priority of the process.

There are also three additional windows available from the main menu bar. As previously mentioned, there is a Globals sub-window which displays the values of global variables. This includes normal variables, semaphores, and monitors. This is extremely beneficial from a debugging point of view, as the user can see the values of semaphores as they change. For monitors, all of the variables declared within the monitor can be displayed. The Console sub-window displays data written by all processes to standard output. The History sub-window displays a log of the most recent 100 P-code instructions that have been executed. The process index, source file name, source file line number, and the P-code instruction are displayed. This is useful for seeing how the preemptive scheduler has scheduled the concurrent processes.

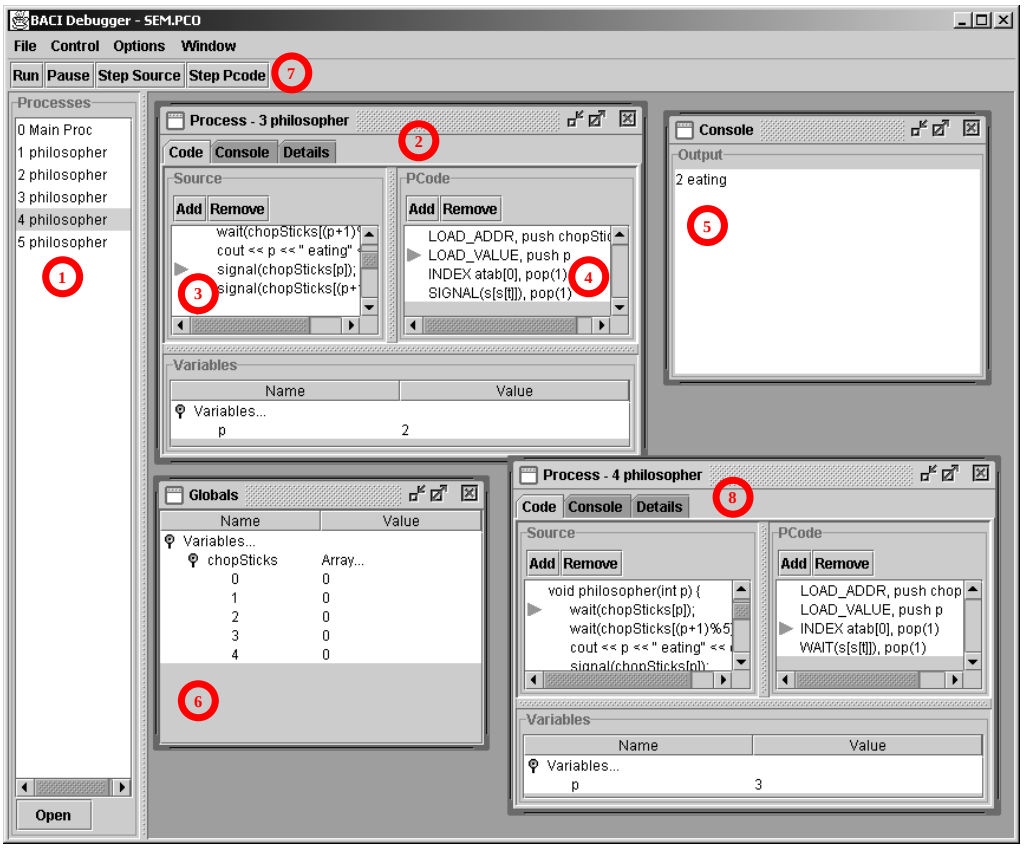

Figure 2: BACI GUI Debugger Screen Shot

Figure 2 shows a screen shot of the BACI GUI Debugger, with the following features marked:

1. The Process List. This lists all of the process currently running, including the main process and all of the concurrent processes. Selecting a process from this list will open a Process sub-window.

2. A Process sub-window for one of the concurrent processes. The Code tab, which displays source code, P-code, and local variables, is displayed.
3. The area of the Process sub-window displaying source code. The current line is indicated by the green arrow.
4. The area of the Process sub-window displaying P-code. The instructions displayed are the ones corresponding to the current source line. The current instruction is indicated by the green arrow.
5. The Console sub-window. This displays the output from all of the processes.
6. The Globals sub-window. This displays the values of all global variables.
7. Control buttons. These allow the user to step through code as well as to continue and pause execution.
8. A Process sub-window. This is an example of a second process sub-window.

The execution control buttons are displayed on a toolbar at the top of the main window. Figure 3 shows these buttons.

Figure 3: Execution Control Buttons

The Run button forces the debugger to run the program without pausing. Execution will stop when one of the following occurs: the end of the program is reached, the Pause button is pressed, or a breakpoint is encountered. (If a program has a deadlock, this will also terminate the program, but will also pop up a window indicating deadlock has occurred.) The Pause button causes the debugger to pause the program execution if it is currently running. Pressing the Step Source button tells the debugger to execute one line of source code. Execution will pause after the line is executed. The Step Pcode button tells the debugger to execute exactly one P-code instruction. Execution will pause after the instruction is executed. These options are also available on the Control menu. The Control menu additionally has a Restart option, which will restart the debugger at the beginning of the program.

The Options menu contains options that will allow the user to specify certain behaviors of the debugger. The Pause on Process Swap option determines if the debugger should pause when the preemptive scheduler swaps processes. If this option is checked, the debugger will pause when a process swap occurs. If unchecked, the debugger will not pause on process swaps. The Show Active Window option determines if the Process sub-window for the active process should always be displayed. If this option is checked, the Process sub-window for the new active process will be displayed at each process swap. On a process swap, the corresponding Process sub-window will be created if it has not yet been opened or brought to the foreground if it is already open. If this option is unchecked, the debugger will not automatically display Process sub-windows for the active process. Figure 4 shows the Options menu.

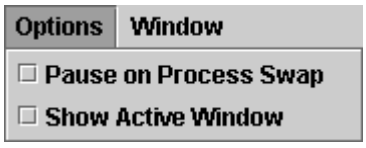

Figure 4: Options Menu

Figure 5: Input Dialog

If the program requests input from standard input, the input dialog will be displayed. This allows the user to type in the requested data. Entering no data will be treated as an EOL character. The EOF character is not supported. Figure 5 shows the input dialog.

The BACI GUI Debugger provides a nice environment for interactively debugging BACI programs. Because it is written in Java, it is portable to any platform on which BACI itself runs. The only disadvantage to writing this debugger in Java is speed: if a user has multiple process sub-windows open, the debugger is noticeably slow.

6 CONCLUSION

Concurrent programming and process synchronization are important topics for computer science students to understand. However, these concepts are typically very difficult for introductory operating systems students to grasp. BACI is a very attractive tool for providing hands-on programming experience with concurrency concepts. Because the BACI syntax is simple, the learning curve is low. This, combined with the fact that the kit itself is free, make BACI a very suitable tool for a two or three week unit on concurrency. However, the debugger provided with the BACI system is restricted to the command line only. This, combined with the fact that concurrent programs are nondeterministic, makes finding problems in concurrent programs very difficult. In this paper, we have introduced a GUI debugger for the BACI system. The BACI GUI Debugger is designed to help students learn about concurrent programming and process synchronization by allowing them to see what is happening while their concurrent program is running.

Students at our university have been using the BACI GUI Debugger for one semester. Feedback indicates the debugger is intuitive and quite useful. By allowing the user to see the interleaved schedules, students are developing a better understanding of concurrent programming. An added benefit we had not expected was an enhanced understanding of the compilation process and execution of machine code.

Future work for the BACI GUI Debugger could include adding support for Distributed BACI. Distributed BACI is currently being developed by Bynum and Camp to give students an opportunity to write concurrent programs that run in a distributed environment.

REFERENCES

[1] ACM Computing Curriculum: CC-1991 (http://www.acm.org/education/curr91/homepage.html) and CC-2001 (http://www.computer.org/education/cc2001).

[2] Andrews, G. and R. Olsson, *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Co., 1993.

[3] Ben-Ari, M., *Principles of Concurrent Programming*. Prentice Hall, Inc., 1982.

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

[5] Brumfeld, J.A. "Concurrent Programming in Modula-2", *Proceedings of the ACM Computer Science Education Technology Conference*, pp. 191-200, March 1987.

[6] Burns, A. and A. Wellings, *Concurrency in Ada*. Cambridge University Press, 1995.

[7] Bynum, Bill and Tracy Camp, "After You, Alfonse: A Mutual Exclusion Toolkit." *ACM Twenty-seventh SIGCSE Technical Symposium on Computer Science*. ACM Press, pp. 170-174, 1996.

[8] Bynum, Bill and Tracy Camp, BACI Source Code. Available at http://www.mines.edu/fs_home/tcamp/baci/bacisrc.tar.gz, 2001.

[9] Bynum, Bill and Tracy Camp, *User's Guide: Ben-Ari C-- Compiler and PCODE Interpreter*. Available at http://www.mines.edu/fs_home/tcamp/baci/bacidoc.tar.gz, 1999.

[10] Bynum, Bill and Tracy Camp, *User's Guide: Ben-Ari Pascal Compiler and PCODE Interpreter*. Available at http://www.mines.edu/fs_home/tcamp/baci/bacidoc.tar.gz, 1999.

[11] Fankhauser, G., Conrad, C, Zitzler, E., Plattner, B. *Topsy, A Teachable Operating System*. (http://www.tik.ee.ethz.ch/~gfa/papers/papers.html#topsy).

[12] Hartley, S., *The SR Programming Language:Operating Systems Programming*. Oxford University Press, 1995.

[13] Hayden, C., "A Brief Introduction to Concurrent Pascal." *SIGPLAN*, ACM Press, pp. 353-354, 1993.

[14] Higginbotham, C., Morelli, R. "A System for Teaching Concurrent Programming", *Proceedings of the ACM Computer Science Education Technology Conference*, pp. 309-316, March 1991.

[15] Jipping, M., Toppen, J., Weeber, S. "Concurrent Distributed Pascal" A Hands-On Introduction to Parallelism", *Proceedings of the ACM Computer Science Education Technology Conference*, pp. 94-99, March 1990,.

[16] Kifer, M. and Smolka, S. *OSP: An Environment for Operating System Projects*, Addison-Wesley, 1991.

[17] Lea, D., *Concurrent Programming in Java*. Addison-Wesley, Second Edition, 1999.

[18] Leach, J., "Experiences Teaching Concurrency in Ada," *Ada Letters*, Vol. 17, No. 5, pp. 40-41, 1987.

[19] Leveson, Nancy and Clark S. Turner, "An Investigation of the Therac-25 Accidents." *IEEE Computer*, Vol. 26, No. 7, pp. 18-41, July 1993.

[20] Milenkovic, M. *Operating Systems: Concepts and Design*, 2nd Edition, McGraw-Hill, 1992.

[21] Nachos Home Page. (http://HTTP.CS.Berkely.EDU/~tea/nachos/)

[22] Null, L. "Integrating Concurrent Programming into an Introductory Operating Systems Class Using BACI," The Journal of Computing in Small Colleges, Vol. 14, No. 3, pp. 288-298.

[23] Silberschatz, A. and Galvin, P. *Operating System Concepts*, 5th Edition, Addison-Wesley, 1997.

[24] Stallings, W. *Operating Systems*, 3rd Edition, Prentice-Hall, 1998.

[25] Tanenbaum, A. and Woodhull, A. *Operating Systems: Design and Implementation*, 2nd Edition, Prentice-Hall, 1997.

[26] Wirth, N., "Modula: A Language for Modular Multiprogramming," *Software Practice and Experience*, Vol. 7, pp. 3-35, 1997.

[27] Yue, K., "An Undergraduate Course in Concurrent Programming Using Ada." *SIGSE Bulletin*, Vol. 26, No. 4, pp. 59-62, 1994.