VISUALIZATION OF MONITOR-BASED EXECUTIONS

by

KEERTHIKA KOTEESWARAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2008

To My Family.

ACKNOWLEDGEMENTS

I would like to thank Dr. Jeff Lei for trusting my abilities and giving me an opportunity to work on this thesis. Working with him and his research has been an excellent learning experience for me. I would also like to thank Dr. Bahram Khalili and Dr. Gautam Das for accepting to serve on my thesis committee.

I would also like to thank Arun Ramani for being a great team member. His cooperation and technical skills played a major role in this thesis work. I would also like to acknowledge creator of JSWAT Mr. Nathan Fiedler for his valuable inputs and his patience in answering my questions. I would also like to thank all my friends and roommates for their constant support.

Most important of all, I would like to thank my parents G. Koteeswaran and Vathsala Koteeswaran and my sister Kavitha for their complete belief in my abilities and unwavering encouragement. They inculcated good values in me and taught me to set high goals. They are my greatest strength.

March 7, 2008

ABSTRACT

VISUALIZATION OF MONITOR-BASED EXECUTIONS

KEERTHIKA KOTEESWARAN, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Jeff Lei

Concurrent programs play a vital role in the world of software engineering. Concurrent programs are used in various fields such as safety critical applications, applications involving advanced graphical user interface, operating system implementation etc. Concurrent programs due to their inherent multi-threaded nature pose a challenge to software engineers not only in coding but also in ensuring data consistency and accuracy. One of the important software constructs used by concurrent programmers to handle critical sections and data synchronization is monitors. A monitor is an encapsulation of shared data, operations on the data and any synchronization required to access the data [1]. Monitors are widely used by concurrent programmers since it guarantees mutual exclusion and can be implemented to ensure synchronization between all the threads in a multi-threaded application.

All concurrent programs are unpredictable in nature since the output directly depends on the sequence of thread execution. This unpredictability poses a great challenge to concurrent programmers. This thesis mainly proposes and implements a method to visualize the working of monitors in concurrent programs. This would facilitate concurrent programmers to view, understand, analyze and test the monitor implemented in their program. The GUI application is incorpo-

rated with JSWAT which is a open-source graphical debugger front end based on Java Platform Debugger Architecture. The motive behind integrating the visualization application and JSWAT is to provide useful features like sophisticated breakpoints, colorized code display, panels displaying call stacks, visible variables apart from visualization of monitor execution. The visualization application analyzes each statement in the code to capture data pertinent to visualization. This data is communicated to the GUI which maps it to the appropriate visualization rule. The visualization rules determine where each thread is to be positioned in the GUI. When the concurrent program is executed in the debugger, the user would be able to view the different threads in different positions in the GUI. The positions would directly correspond to the states of the threads with respect to the monitor.

Visualizing the interaction of the different threads with each other and with the monitor would facilitate a concurrent programmer to analyze the program and ensure its accuracy and correctness. This tool can also be used as a teaching aid to instruct software engineering students and novice concurrent programs by helping them to visualize, understand and appreciate the working of concurrent programs in general and monitors in particular. This tool aims to contribute towards easing the burden of concurrent programmers by enhancing their understanding of monitor based executions.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We are currently living in an era where computers have touched every aspect of our lives. Behind every software that has enhanced our quality of life is an enormous amount of effort contributed by software programmers. During the process of creating a software, a major portion of time and effort is spent in testing the debugging it. So it is necessary to explore new ways to minimize the time, effort and other resources spent in testing and debugging a software application.

1.1 Goal of this thesis

In the current scenario where software is used to solve more and more complex problems of mankind, concurrent programs have proved to be the solution. Since they are inherently multi-threaded, concurrent programs can handle multiple tasks in any given time. So they are used even in safety critical applications like in a nuclear power station, space exploration probes, satellites etc. They are also used in advanced graphical applications and animation. But concurrent programs are even tougher to debug than sequential programs since there are multiple threads executing together. This may result in issues such as guaranteeing mutual exclusion to protect shared data, avoiding deadlocks, ensuring progress, non-determinism in execution, synchronization and communication between different threads etc. There are several constructs used in concurrent programs such as locks, semaphores, monitors etc. Among these software constructs, monitors are one of the most popular features used in programming concurrent applications.

For verifying and validating concurrent applications, it is absolutely essential to use a debugging environment aimed at addressing the issues involved in

programming them. This thesis work aims in narrowing the gap between the features offered by a traditional debugging environment and the tools expected by a concurrent programmer. This is the goal achieved by developing the visualization tool in this work. The visualization tool focuses on monitors and an efficient way to visualize the interaction of the various threads in a multi-threaded application with the monitor in real-time. The tool aims to display graphically the status of each thread with respect to the monitor in any given time. The tool also handles different types of signaling disciplines such as Signal-and-Continue, Signal-and-Urgent-Wait and Signal-and-Exit. The programmer also has the option to start, pause and stop the visualization at any point of time and view the status of the application. This gives the programmer control over the visualization process.

To further enhance the visualization tool, it is integrated with JSWAT which is a proven advanced graphical Java debugger. JSWAT is based on the Netbeans platform and offers extremely useful features such as colorized breakpoints, watching specific variables and methods, identifying syntax errors, stopping and starting execution using appropriate buttons and view threads, call stacks and visible variables. The JSWAT and visualization tool integrated environment will alleviate the problems concurrent programmers face during the process of testing and debugging. This tool will also serve as an user-friendly teaching aid for instructors for explaining advanced multi-threading concepts.

My contribution in this thesis work is to design and implement the visualization panel. I implemented the communication interface between the user application and the visualization tool. I also collaborated in the visualization rules that decide what visualization steps are executed based on the monitor type and messages.

1.2 Structure of this thesis

The thesis is structured as follows: Chapter 1 gives a brief overview of the thesis work. Chapter 2 deals with the background and related work for concurrent program testing and debugging. Chapter 3 explains monitors and signaling disciplines. Chapter 4 deals with JSWAT and Chapter 5 explains the working of the visualization tool. Chapter 6 explains how JSWAT and the visualization tool are synchronized. Chapter 7 includes use case diagram and sequence diagram and sample executions. Lastly, Chapter 8 concludes with the goals achieved by this work.

CHAPTER 2

RELATED WORK

2.1 Dynamic Java Visualizer

Dynamic Java Visualizer is the work of Steven P. Reiss and his research group in Brown University. The aim of the visualizer is to display program-specific information in real time with minimum overhead. The concept behind the tool is to divide the execution of a Java program into intervals and display information about what the program has done in terms of classes, memory and threads [7]. The information to be collected in each interval consists of:

- What is being executed. This data is grouped by classes, packages or collection of packages.
- How much time each threads spends in each class.
- How much time is spent in each class for synchronization.
- How many memory allocations occur for each class of object.
- How much memory is being deallocated.
- The current set of threads created or destroyed by the application.
- What is the current status of a particular thread. The thread might be active, sleeping, blocked, doing I/O, executing in a synchronized region. Also the time each thread spends in a particular state is displayed.
- Which thread is blocking which thread.

All the above information is obtained by inserting method calls wherever there is a significant event. There is a vast amount of data to be collected during the execution of a program. Therefore, to better manage the information, data is segregated according to classes, packages and package hierarchy. For example, all classes in java.io.* is represented by a single visualization construct.

The classes themselves are categorized into three categories.

- Detailed classes are those directly in the users application. Events are generated for all methods.
- Library classes are grouped into packages. Events are generated only for initial entry into the library. Events are not generated for calls within the library class.
- Classes that are neither detailed nor library are treated at an intermediate level of granularity. Here nested classes are merged with their parent and only public methods are counted.

Each event is then processed appropriately and sent to the visualizer. Method entry and exit events are used to determine what was executing and the state of each thread. To augment that, information about synchronization events are collected by inserting calls immediately before and after synchronized entry. A buffer stores all these data. A monitoring thread wakes up at the end of each interval to generate a report which is in XML. The trace package sends this output directly to the visualizer along with general information about the intervals such as totals and the time represented by the interval.

Visualization is done using a box display. Each class or thread is represented as a box in the display. Each property of the box denotes a characteristic of the program. First, the height of the rectangle is used to indicate the number of calls. Second the width of the rectangle is used to represent the number of allocations by methods of the class. Third, the hue of the rectangle is used to represent the number of allocations of objects of the given class. Fourth, the saturation of the rectangle is used as a binary indicator as to whether the class was used at all during the interval. Finally, the brightness of the box is used to represent the number of synchronization events on objects of this class.

The front end is invoked by using the *jive* command in place of the standard *java* command when running the application. This command starts the interface of

the visualizer and runs the application as well. The programmer can dynamically specify what portions of the application should be viewed as detailed and library classes. The front end interprets these selections and inserts the necessary method calls. Finally, the front end provides separate windows for text input and output.

The dynamic java visualizer displays important statistics of the application in real-time. It helps the programmer to understand what the application is doing in a particular time interval. But the information collected by the visualizer might not be sufficient to give a programmer a detailed insight into a multithreaded application. This tool can only display states where a thread is waiting on a monitor, running inside a monitor or releasing a monitor. The interaction among the threads with respect to the monitor is not shown. Also, the visualizer displays the events occurring during a time interval and not in real-time. This might lead to some interactions being omitted from the visualization.

2.2 Visualization of Concurrent Program Executions

The authors of this research work are Cyrille Artho (Research Center for Information Security, Japan), Klaus Havelund (NASA Jet Propulsion Laboratory, USA) and Shinichi Honiden (National Institute of Informatics, Japan).

The authors have worked on a approach based on UML diagrams to visualize concurrent programs [8]. They have extended UML sequence diagrams by adding features to it that will help in visualizing concurrent programs. Traditional sequence diagrams cannot illustrate the following:

- A Thread as a data structure and executable task.
- Context switches induced by the scheduler.
- Activations and suspensions of threads. Actions such as threads waiting on events cannot be illustrated.
- Time-based suspension.
- The happens-before relation.

- Locking, which is used to guarantee mutual exclusion and also to synchronize different threads.

The visualization is based on the Java programming language but is also applicable for other programming languages. This approach distinguishes between the two roles of a Java thread as an executable task and a data structure. The thread data structure holds information such as thread name and ID. A thread as a *task* constitutes a light-weight process that shares the global heap with other threads. The first extension of the UML sequence diagram is the visualization of a thread as an executable task by a hexagon. A dashed arrow pointing to the left symbolizes the thread scheduler running a thread(task). As in UML sequence diagrams, solid arrows depict a method call or return, and solid squares show a method being executed. Dotted lines show event dependencies according to the happens-before relation. If there is a dotted line from a point p to a hexagon t, then any events following an activation of thread t could have started right *after* p.

The start of a thread is shown by a corresponding action in the thread scheduler, using a dashed arrow pointing from a hexagon to the left. Thread termination is not shown since there is no further action of that thread to be visualized. But thread termination may trigger another threads action that is depicted as a happens-before relationship. Thread notification is considered as re-activation of the thread after suspension. Calls to *wait* and *notify* are illustrated by a solid black box just like regular method calls. A *notifyAll* involves multiple threads in a happens-before relation. Locks are not directly visualized but are shown as secondary notations or annotations.

This approach visualizes a concurrent program in a slightly higher abstraction which improves scalability. Since this approach is an extension of UML sequence diagrams it is easily understandable but is restricted by the scope of UML diagrams. Moreover, this approach requires thorough knowledge of UML diagrams

to interpret the visualization. Also, the visualization cannot be controlled by the user.

CHAPTER 3

MONITORS

3.1 Introduction to Monitors

A monitor encapsulates shared data, all the operations on the data, and any synchronization required for accessing the data [1]. Mutual exclusion is provided automatically by the monitors implementation, freeing the programmer from the burden of implementing critical sections. An object-oriented definition of a monitor is that a monitor is a synchronization object that is an instance of a special *monitor* class. A monitor class defines private variables and a set of public and private access methods. The variables of a monitor represent shared data. Threads communicate by calling monitor methods that access the shared variables. The need to synchronize access to shared variables distinguishes monitor classes from regular classes.

3.1.1 Mutual Exclusion

At most one thread is allowed to execute inside a monitor at any time. However, it is not the programmers responsibility to provide mutual exclusion for the methods in a monitor. Mutual exclusion is provided by the monitors implementation. If a thread calls a monitor method but another thread is already executing inside the monitor, the calling thread would wait outside the monitor. A monitor has an entry queue to hold the calling threads that are waiting to enter the monitor.

3.1.2 Condition Variables

Condition synchronization is achieved using condition variables and operations *wait()* and *signal()*. A condition variable denotes a queue of threads that are waiting for a specific condition to become true. A condition variable *cv* is declared as ConditionVariable cv. Operation *cv.wait()* is analogous to a P operation in that it is used to block a thread. Operation *cv.signal()* unblocks a thread and is analogous to a V operation. A monitor has one entry queue plus one queue associated with each condition variable. A thread that is executing inside a monitor method blocks itself on condition variable *cv* by executing cv.wait(). Executing a *wait()* operation releases mutual exclusion (to allow another thread to enter the monitor) and blocks the thread on the rear of the queue for *cv*. The threads blocked on a condition variable are considered to be outside the monitor. If a thread that is blocked on a condition variable is never awakened by another thread, a deadlock occurs.

A thread blocked on condition variable *cv* is awakened by the execution of cv.signal(). If there are no threads blocked on *cv*, the *signal()* operation has no effects; otherwise, the *signal()* operation awakens the thread at the front of the queue for *cv*. What happens next depends on the signaling discipline used. Operation *cv.signalAll()* wakes up all the threads that are blocked on condition variable *cv*.

A *notify()* operation notifies one of the waiting threads, but not necessarily the one that has been waiting the longest or the one with the highest priority. If no threads are waiting, a *notify()* does nothing. A *notifyAll()* operation awakens all the waiting threads.

The *synchronized* modifier is one of Javas built-in synchronization constructs. Each Java object is associated with a built-in lock. If a thread calls a method on an object, and the method is declared with the *synchronized* modifier, the calling thread must wait until it acquires the objects lock. Only one thread at a time

can execute in the *synchronized* method of an object. Java‘s implementation of a *synchronized* method ensures that the objects lock is properly acquired and released. If an object‘s data members are only accessed in *synchronized* methods, the thread that owns the objects lock has exclusive access to the object‘s data members.

3.2 Monitors in Java

There are significant differences between Java‘s monitor-like objects and general monitors. First, adding *synchronized* to the methods of a Java class automatically provides mutual exclusion for threads accessing the data members of an instance of this class. The second major difference is that there are no explicit condition variables in Java. When a thread executes a *wait* operation, it can be viewed as waiting on a single, implicit condition variable associated with the object.

3.3 Signaling Disciplines

3.3.1 Signal-and-Continue (SC)

After a thread executes an SC signal to awaken a waiting thread, the signaling thread continues executing in the monitor and the awakened thread is moved to the entry queue. That is, the awakened thread does not reenter the monitor immediately; rather, it joins the entry queue and waits for its turn to enter. When SC signals are used, signaled thread have the same priority as threads trying to enter the monitor via public method calls.

3.3.2 Signal-and-Urgent-Wait (SU)

Behavior of *wait* and *signal* operations for an SU monitor are as follows: When a thread executes *cv.signal()*:

- If there are no threads waiting on condition variable *cv*, this operation has no effect.
- Otherwise, the thread executing *signal* (which is called the *signaler thread*) awakens one thread waiting on *cv* and blocks itself in a queue, called the *reentry queue*. Threads blocked in the reentry queue are considered to be outside the monitor. The signaled thread reenters the monitor immediately.

When a thread executes *cv.wait()*:

- If the reentry queue is not empty, the thread awakens one signaler thread from the reentry queue and then blocks itself on the queue for *cv*.
- Otherwise, the thread releases mutual exclusion (to allow a new thread to enter the monitor) and then blocks itself on the queue for *cv*.

When a thread completes and exits a monitor method:

- If the reentry queue is not empty, it awakens one signaler thread from the reentry queue.
- Otherwise, it releases mutual exclusion to allow a new thread to enter the monitor.

In an SU monitor, the threads waiting to enter a monitor have three levels of priority (from highest to lowest):

- The awakened thread (A), which is the thread awakened by a *signal* operation
- Signaler threads(S), which are the threads waiting in the reentry queue
- Calling threads (C), which are the threads that have called a monitor method and are waiting in the entry queue.

In an SU monitor, the relative priority associated with the three sets of threads is $A > S > C$.

3.3.3 Signal-and-Exit (SE)

Signal-and-Exit is a special case of signal-and-urgent-wait. When a thread executes an SE *signal* operation it does not enter the reentry queue; rather, it

exits the monitor immediately. Thus, an SE *signal* statement is either the last statement of a method or is followed immediately by a return statement. As with SU signals, the thread awakened by a *signal* operation is always the next thread to enter the monitor. In an SE monitor, since there are no signaling threads that want to remain in or reenter the monitor, the relative priority associated with the sets of awakened (A) and calling (C) threads is $A > C$.

3.3.4 Urgent-Signal-and-Continue (USC)

In the *urgent-signal-and-continue* (USC) discipline, a thread that executes a *signal* operation continues to execute just as it would for an SC signal. But unlike SC signals, a thread awakened by a *signal* operation has priority over threads waiting in the entry queue. That is, a thread waiting in the entry queue is allowed to enter a monitor only when no other threads are inside the monitor and no signaled threads are waiting to reenter. When *signal* operations appear only at the end of monitor methods, which is usually the case, this discipline is the same as the SE discipline, which is a special case of the SU discipline.

3.4 Monitor Toolbox

A monitor toolbox is used to simulate the monitor construct. Using the toolbox a regular Java class can be used to simulate a SC or SU monitor. This is achieved as follows:

- Extend class *monitorSC* or *monitorSU*.
- Declare *conditionVariables* as required by the application.
- Use operations *enterMonitor()* and *exitMonitor()*.
- Use operations *waitC()*, *signal()*, *length()*, *empty()* on the *conditionVariables* as required by the application.

The advantages of using a monitor toolbox is that it can be used to simulate monitors in languages that do not generally support monitors, such as C++. It

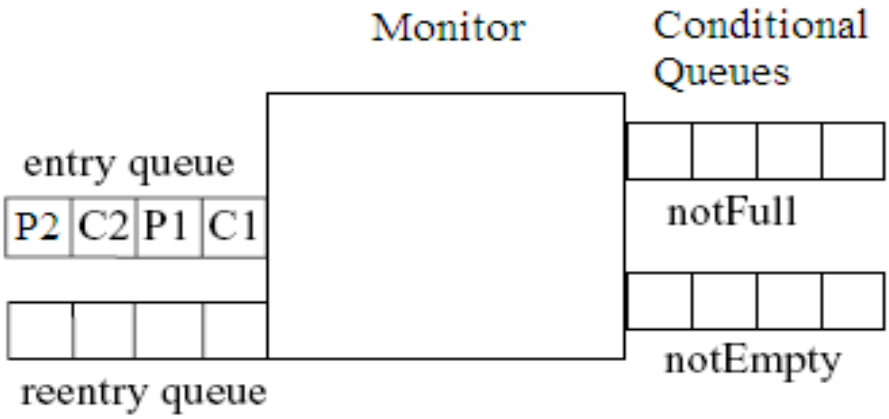

Figure 3.1 Producers and consumers competing to enter the monitor

can also be used to implement different signaling disciplines. Also, it offers greater flexibility for testing and debugging.

3.5 Monitor-Based Solutions to Concurrent Programming Problems

3.5.1 Bounded Buffer Problem

The bounded-buffer problem is a classical multithreaded synchronization problem introduced by Dijkstra . A bounded buffer has n slots. Each slot is used to store one item. Items are deposited into the buffer by a single producer and withdrawn from the buffer by a single consumer. A producer is not permitted to deposit an item when all the slots are full. A consumer is not permitted to withdraw an item when all the slots are empty [6].

Let us consider a system with a SU monitor, two producers and two consumers. The producers are denoted by P1and P2 respectively. The consumers are denoted by C1 and C2 respectively. The conditional queues are notFull and notEmpty. The buffer contains only one slot.

In Figure 3.1, the producer and consumer threads are lined in the entry queue in the following order: C1, P1, C2, P2. The order in which the threads attempt to enter the monitor is assumed to be first-come first-served.

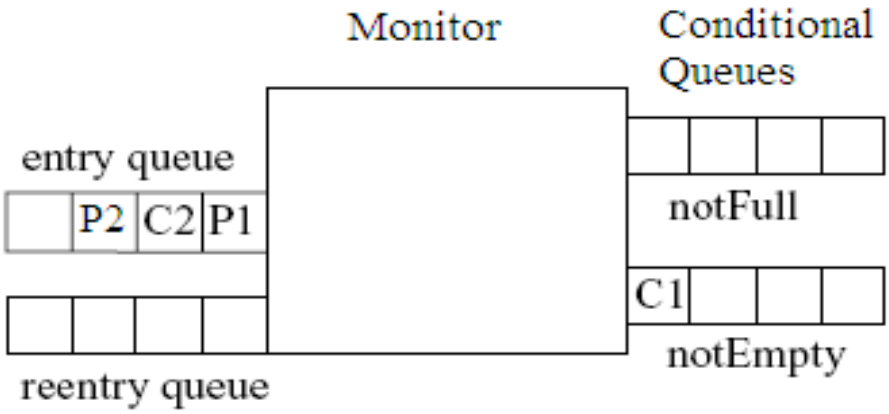

Figure 3.2 C1 waiting in notEmpty queue for a producer to deposit

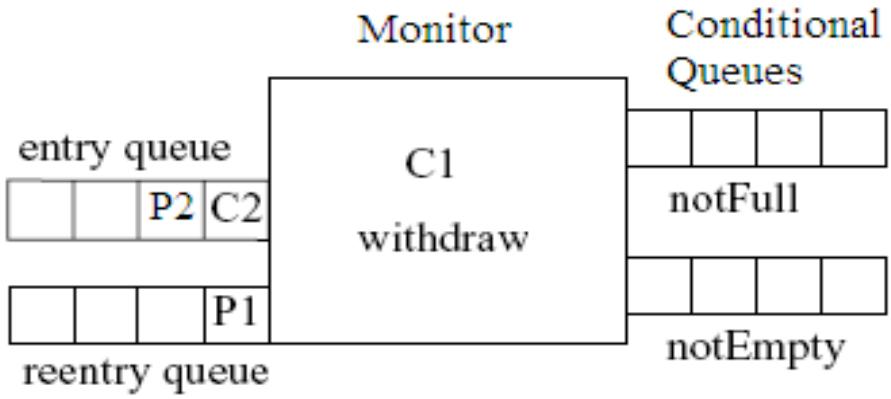

Figure 3.3 C1 withdrawing from buffer and P1 waiting in reentry queue.

In Figure 3.2, C1 first attempts to withdraw, but since the slot is empty, it is made to wait in notEmpty conditional queue.

Next in Figure 3.3, P1 deposits and since it is a SU monitor, it is made to wait in the reentry queue. Before, it waits in the reentry queue, it wakes up C1 from the conditional queue. C1 enters the monitor and withdraws from the slot. Since the reentry queue is not empty, it wakes up P1 before exiting the monitor.

In Figure 3.4, P1 reenters the monitor and since it has already deposited, it exits the monitor.

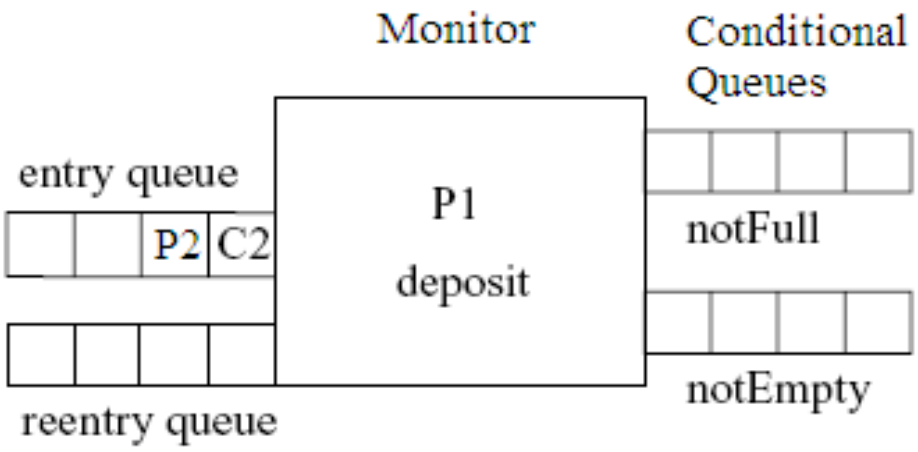

Figure 3.4 P1 re-entering the monitor

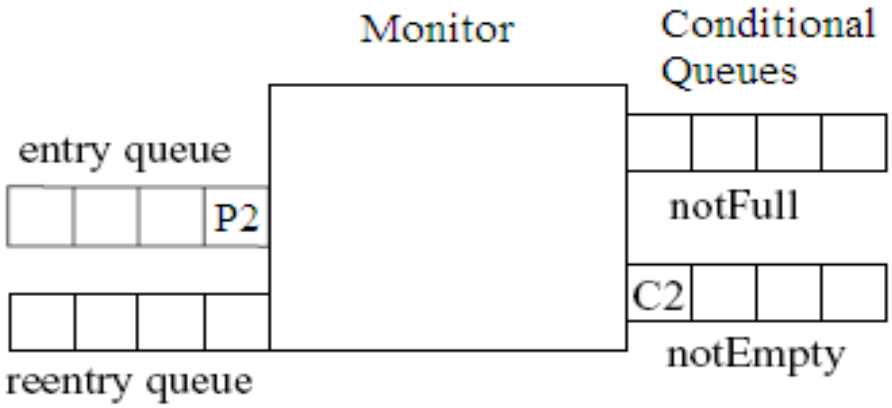

Figure 3.5 P2 depositing and C2 waiting in notEmpty conditional queue

In Figure 3.5, the next thread in the entry queue, C2, attempts to withdraw. The buffer is empty so it is made to wait in the notEmpty conditional queue.

In Figure 3.6, the next thread, P2 enters the monitor, deposits in the buffer. It wakes up C2 from the conditional queue and waits in the reentry queue. C2 enters the monitor and withdraws from the buffer. Since the reentry queue is not empty, C2 wakes up P2 before exiting the monitor.

Next in Figure 3.7 P2 reenters the monitor but since it has already deposited, it immediately exits the monitor.

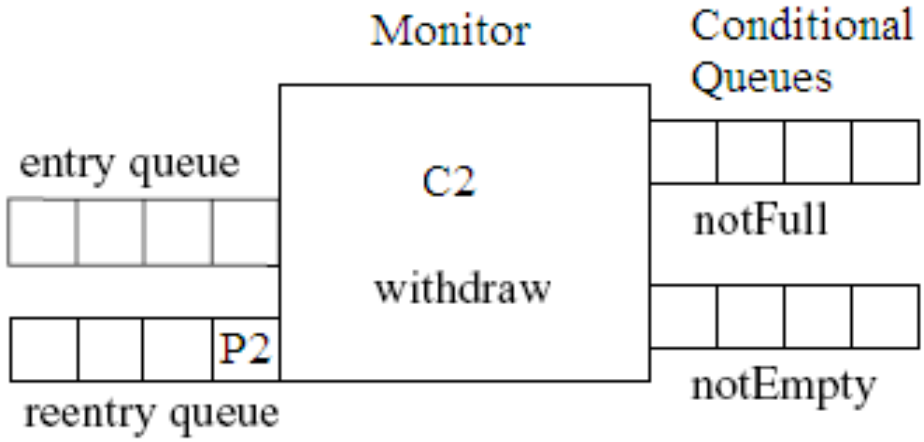

Figure 3.6 P2 waiting in reentry queue and C2 withdrawing from buffer

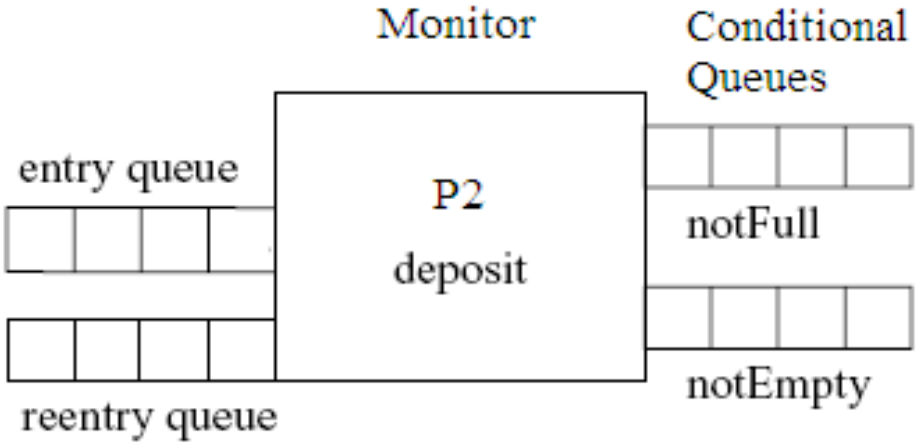

Figure 3.7 P2 re-entering the monitor

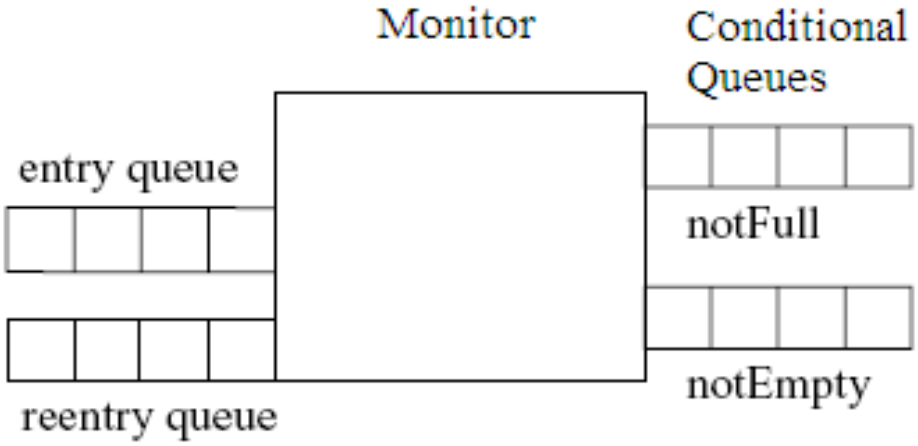

Figure 3.8 Monitor after P2 exits the monitor

The above example illustrated how monitors can be used to solve the bounded buffer problem as they inherently guarantee mutual exclusion.

CHAPTER 4

JSWAT

JSWAT is a standalone, graphical Java debugger developed by Nathan Fiedler. It is based on the Netbeans Platform and its infrastructure is provided by Java Platform Debugger Architecture(JPDA) [4]. It includes the following three-layered APIs as in Figure. 4.1:

- Java Debug Interface (JDI), a high-level Java programming language interface, including support for remote debugging. This interface facilitates the integration of debugging capabilities into development environments.
- Java Debug Wire Protocol (JDWP), which defines the format of information and requests transferred between the process being debugged and the debugger front end. It does not define the transport mechanism. The specification of the protocol allows the debuggee and debugger front-end to run under separate VM implementations and/or on separate platforms. It also allows the front-end to be written in a language other than Java, or the debuggee to be non-native.
- The JVM Debugger Interface (JVMDI), a low-level native interface that defines the services a Java virtual machine provides for tools such as debuggers and profilers. It defines the services a VM must provide for debugging. It includes requests for information, actions, and notification. This is the source of all debugger specific information. Specifying the VM interface allows any VM implementor to plug easily into the debugging architecture.

JPDA provides a standard interface which allows Java programming language debugging tools to be easily written without regard to platform specifics such as hardware, operating system and virtual machine implementation. It de-

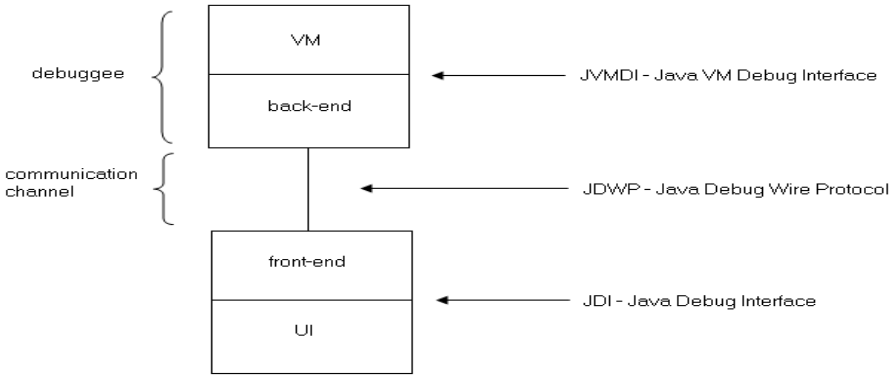

Figure 4.1 Java Debugger Platform Architecture

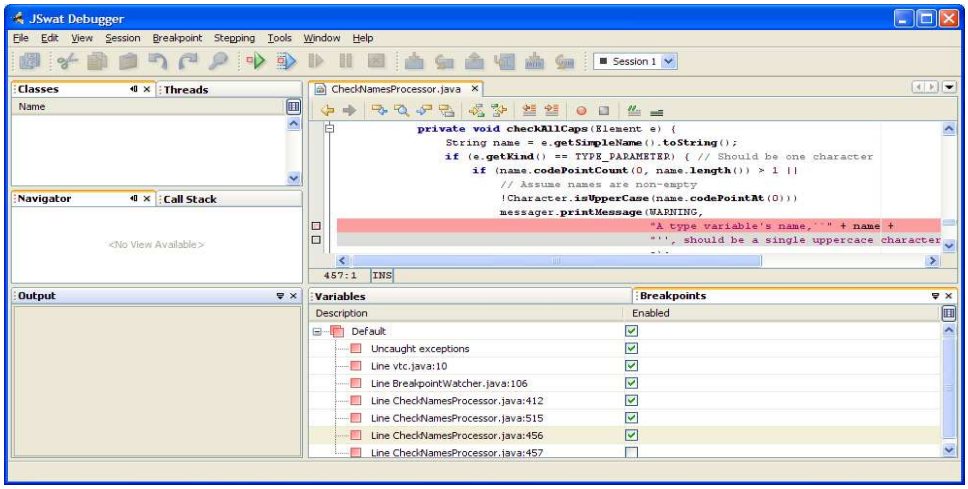

Figure 4.2 JSwat Main Window

scribes a complete architecture for implementing these interfaces, including remote and cross-platform debugging. It also provides a reference implementation of this architecture. It provides a highly modular architecture where the implementation and client of an interface can be different than the reference implementation or different from the JPDA component. The back-end of the debugger is responsible for communicating requests from the debugger front-end to the debuggee VM and for communicating the response to these requests to the front-end. The back-end communicated with the front-end over a communications channel using the Java Debug Wire Protocol (JDWP). The transport mechanism used in the com-

munication channel is left unspecified; possible mechanisms include: sockets, serial lines, and shared memory. However, the format and semantics of the serialized bit-stream flowing over the channel is specified by the Java Debug Wire Protocol. The back-end communicated with the debuggee VM using the Java Virtual Machine Debug Interface (JVMDI). The debugger front-end implements the high-level Java Debug Interface. JSwat is an open-source software and its binary as well as source code are freely available here [2]. Its features include:

- Sophisticated breakpoints
- Colorized source code display with code navigator
- Byte code viewer
- Movable display panels showing threads
- Call stack
- Visible variables and loaded classes
- command interface for more advanced features
- Java-like expression evaluation; including method invocation

Since JSwat is a standalone application, it defines its own menu structure and default window arrangement, as well as a splash screen and about dialog, refer Figure. 4.2. It is also possible to build and run JSwat as a plugin for the NetBeans IDE. The plugin form of JSwat is built by modifying a property file, build.properties. The modifications are building.plugin = yes and test.user.dir = buildideuserdir. The *building.plugin* property controls the building of the plugin module itself. It also disables the product module, and the usual product branding. Setting *test.user.dir* is optional, but highly recommended as it prevents polluting the JSwat test userdir with files and settings pertinent only to the IDE binary. After this preparation is complete, the entire JSwat module suite is cleaned and built and the JSwat Plugin module is rebuilt. The plugin module is not a part of the module suite because it has different module dependencies than the standalone form of JSwat, so it is preferable to keep it separated logically.

4.1 JSwat GUI

The display areas within the main JSwat window are themselves called windows. These windows display the variables, threads, classes, breakpoints, sessions, and so on. Initially not all of the windows are visible, so the *Window* menu can be perused to open other available windows. To arrange the windows within the main window, the title bar of the window is clicked and dragged to a different location. An outline indicates where the window will be displayed when the mouse button is released. The size of the window can be changed in relation to one another by dragging the dividers between the window areas.

4.2 Starting the debuggee

To start debugging code, the application is launched from the debugger, or started separately and the debugger is connected as needed. The launching dialog is appropriate for small applications with a simple launching mechanism, whereas the attaching method is used if the application has a launcher of its own, or requires significant setup. To launch the debuggee, *Start* is selected from the *Session* menu or by clicking the corresponding toolbar button as in [3]. The dialog that appears has a *Help* button that will display the help page to explain all of the input fields. The name of the application's main class as well as the classpath that is used to launch the application must be provided. Once the debuggee is launched, it is in a paused state, waiting for the signal to start. To start the debuggee, *Continue* is selected from the *Session* menu or by clicking the corresponding toolbar button. The debuggee will run until it hits a breakpoint, or it exits normally.

Attaching to a debugee after it has been launched requires that the debuggee is launched with certain debugging flags. A different number can be chosen for the address value as long as it is between 1024 and 65535, inclusive, and not already in use by another program. If Microsoft Windows is being used, shared memory transport can be used instead of the sockets transport. This can be done

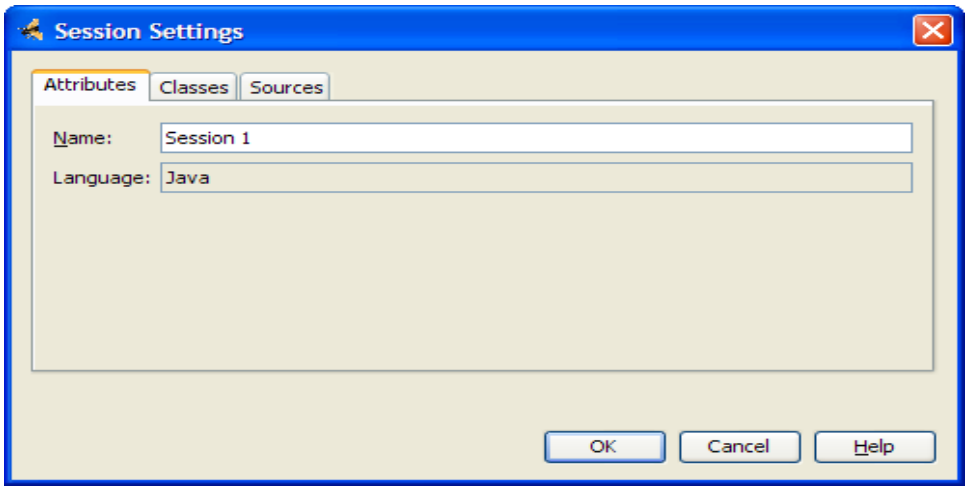

Figure 4.3 JSwat Settings Dialog

by changing the dt_socket to dt_shmem and the address value to another name. Once the debuggee has been launched with the flags, *Attach* is selected from the *Session* menu. The host field can be left blank to default to the local machine, otherwise a machine name or an IP address can be entered. If a name is entered, the name must be resolvable by the networking interface on the local machine. For the port number field, the number given as the address should be entered in the debug flags (e.g. 5000). If the shared memory transport is used, then the shared memory option is selected from the *Transport* field in the attach dialog, and the name from the address option is entered in the debug flags. If the debuggee was launched with suspend = y then the debuggee can be started by selecting *Continue* from the *Session* menu (or by clicking on the corresponding toolbar button). The debuggee will run until it hits a breakpoint, or it exits normally.

4.3 Setting the classpath and sourcepath

The classpath for the application can be already set when launching it, either from the debugger or in a separate command window. In any case, the classpath can be inspected via the *Settings* item in the *Session* menu. While the session is connected to the debugee, the classpath is prevented from being modified, since it is

impossible to modify the classpath of a running JVM. In addition to the classpath, the Settings dialog can be used to specify the directories and archives containing the source code from the user's application as in Figure. fig: JSwatSettings. This is referred to as the sourcepath. The sourcepath has the same structure as the classpath, except it refers to the location of source code, rather than the location of class files. Thus, if there is a classpath that looks like */home/me/java/build*, which contains *.class* files such as *org/mine/Widget.class*, and the source code is similarly located in a directory such as */home/me/project/src/*, which contains *.java* files such as *org/mine/Widget.java*, then the sourcepath would be set to */home/me/project/src*. It should be noted that the sourcepath can contain directories as well as archives, such as *.jar* and *.zip* files.

4.4 Setting breakpoints

To set breakpoints, the debugger is used to open the source file containing the code in which it is to be stopped. The editor is scrolled to the desired line, then the mouse is clicked in the gray margin on the left side of the editor view. Clicking in the margin will create a line breakpoint, and clicking on the line breakpoint icon will remove the breakpoint.

Additional types of breakpoints may be created from the *Breakpoint* menu, including class, exception, method, thread, trace, and variable breakpoints. The dialog for creating breakpoints has a *Help* button that displays a help topic explaining the various input fields.

4.5 Stepping through code

Once the debuggee has been launched and it hits a breakpoint, stepping through the code can be begun. This is done by selecting one of the items in the *Stepping* menu. The *Step Into* item will perform a single-step operation, stepping into method calls, while *Step Over* will step through the method call in one action.

Step Out will finish the current method and stop at the calling method. *Run to Cursor* will set a breakpoint at the current cursor location, the debuggee is resumed so that it will hit the breakpoint, and then breakpoint is automatically deleted. As with many of the other menu items, there are corresponding keyboard shortcuts.

4.6 Displaying variable values

There are three ways in which the value of variables can be viewed. The first is with the *Variables* window, which shows all local variables, as well as the fields of the current object. This display is automatically updated each time the debuggee hits a breakpoint, as well as when the code is stepped through. In addition to that view, there is the editor tooltip, whereby the value of the variable under the mouse pointer will be displayed as tooltip in the source editor. This requires having a source file open in the editor that contains references to the desired variables. This also requires that the debuggee has stopped at a breakpoint in order for the evaluator to have a current thread and stack frame from which to evaluate the variable reference. The third option is to use the *Evaluator* view. Java-like expression can be typed and it will be evaluated and the result is displayed in the window. This requires that the debuggee has stopped at a breakpoint in order to evaluate any variable references. When a flashing red icon is seen in the lower right corner of the main window, it indicates that an unexpected error occurred. The icon can be clicked to see an explanation, along with a stack trace.

CHAPTER 5

VISUALIZATION TOOL

This chapter explains the working of the visualization tool which forms the crux of this thesis work. The goal of the visualization tool is to understand the working of a concurrent program and display it in a graphical format for easy understanding. So the first step in this task to be done by the tool is to find the path where the application is located, the type of monitor used in the application and the number and names of the conditional queues.

The next step would be to analyze the program and extract those events which are to be displayed. Based on this analysis, the information to be gathered will depend on the user input in the previous task. The events that have to be identified are:

- Entry of a thread into a monitor.
- All the threads in the entry queue competing to enter the monitor. Inclusion of a thread in this entry queue.
- A thread being moved from inside the monitor to a condition queue when a condition is satisfied.
- A thread being moved to the entry monitor when a condition is satisfied and if the monitor follows Signal and Urgent Wait signaling discipline.
- In the case of Signal and Continue signaling discipline, a thread being moved to the entry queue from the condition queue when it is awakened.
- In the case of Signal and Urgent Wait, a thread being moved back to the monitor from the reentry queue.
- A thread exiting the monitor.

The event recognizer initially detects the names of all the conditional queues that are created in the application. The conditional queue display in the visualization panel is labeled with the names of the conditional queues. Then probes are inserted at strategic positions in the code. These probes generate messages according to their position in the application. The messages are sent to the visualizer for display in the visualization panel.

5.1 Parser

The parser is the component that gathers information from the application and decides where the probes are inserted. To extract information from the application, *regular expressions* are used. A regular expression is a string that is used to describe or match a set of strings , according to certain syntax rules. Regular expressions are a powerful concept which allows us to search for very complex patterns. This strength of regular expressions is used to a great extent in this thesis work to identify code corresponding to important events in the user's multithreaded application.

Regular expressions in Java require importing java.util.regex.Pattern and java.util.Matcher. The application program is tokenized using String Tokenizer.

5.1.1 Pattern

The pattern object is used to identify the events in the application program that have to be visualized. The steps involved in this process are:

- A regular expression is declared as a string.
- This string is compiled as an object of the Pattern class.
- Matcher class is used to search for the pattern in the program.

The snippet of code in Figure 5.1 shows how a pattern containing the sequence of words join and Monitor is searched in a string. First, a regular expression containing the sequence is created. The character class [a-zA-Z] indicates a through

```
String ptnEntMtr = "([a-zA-Z]+[0-9]) (enterMonitor)
                    (Monitor [A-Z]+)";
Pattern patternEntMtr = Pattern.compile(ptnEntMtr);
Matcher matcherEntMtr = patternEntMtr.matcher(message);
boolean result = matcherEntMtr.matches();
```

Figure 5.1 Regular Expressions Usage

```
String ptnJoin = "([a-zA-Z]+[0-9]) (join) (Monitor [A-Z]+)";
```

Figure 5.2 Regular Expression

z and uppercase A through Z inclusive. The quantifier '+' indicates that the characters from the character class can occur one or more times. The character class [0-9] includes all natural numbers. Therefore, this regular expression looks for a pattern containing words join and Monitor that may be preceded by alphanumeric characters and followed by characters. The string containing this regular expression is compiled into a Pattern object. The matcher methods looks for this pattern in the string message.

5.1.2 Matcher

The Matcher class checks to see if the pattern has occurred in a string. An instance of the Pattern class is used to create a Matcher object. When initializing the object, the string in which the pattern is to be searched is specified. The *matches()* method finds if the pattern occurs in the string. If the pattern is present it returns true otherwise it returns false. The messages matched are captured in *groups*. Groups are numbered by counting their opening parenthesis from left to right. For example, in Figure 5.2, there are three groups.

- Group 1: ([a-zA-Z]+[0-9])
- Group 2: (join)
- Group 3:(Monitor [A-Z]+)

Group 0 always contains the entire expression. Group 1 will contain the name of the thread that calls the *join()* method. Group 2 contains the word join.

```
c1.join(); p1.join();
p2.join(); c2.join();
c3.join(); p3.join();
```

Figure 5.3 Sample block of code to illustrate pattern matching

```
String ptnJoin = "([a-zA-Z]+[0-9]) (join) (Monitor [A-Z]+)";
```

Figure 5.4 Regular expression to identify a join() method

Group 3 contains the monitor type chosen by the user. During a match, each subsequence of the input sequence that matches a group is saved. The captured subsequence may be used later in the expression, via back reference, and may also be retrieved from the matcher once the match operation is complete. The usage of groups can be further illustrated using the snippet of code in Figure 5.3

The entire block of code in Figure 5.3 is tokenized based on semi-colon. Suppose each token is stored in a String array called *message*. Each element of the array is then subjected to the regular expression as in Figure 5.4.

This expression looks for alphanumeric characters followed by the words join and Monitor followed by arbitrary characters which would actually denote the monitor type.

The statement in Figure 5.5 compiles the regular expression and initalizes a Pattern object with it. The Matcher class looks for the pattern in the string *message* like in Figure 5.6.

The result of the search is stored in the boolean variable *matchJoin*. If the pattern is found in the string, the value true is returned and stored in *matchJoin*. If the pattern is not found, the value false is stored in *matchJoin*.

If the pattern is found in the string, the name of the thread calling the *join()* method is stored in *group(1)*, which is displayed in a textbox in Figure 5.7. Thus,

```
Pattern patternJoin = Pattern.compile(ptnJoin);
```

Figure 5.5 Pattern

```
Matcher matcherJoin = patternJoin.matcher(message);
boolean matchJoin = matcherJoin.find();
```

Figure 5.6 Matcher

```
if(matchJoin) {
            textbox.setText(matcherJoin.group(1));
        }
```

Figure 5.7 Group

regular expressions can be used effectively to find pre-determined keywords in the user's application program that correspond to events that have to be displayed.

5.1.3 Using Regular Expressions in Visualization Tool

A crucial part of the visualization process is capturing events that are significant enough to be visualized in the application. These events are captured using the corresponding keywords used in the source code. The keywords are located using regular expressions as explained in the previous subsection. Important information like the name of the thread calling a specific method, the method being called, type of monitor etc is collected using groups and are strung together to form a message. These message are sent to the visualizer as soon as they are generated using a communication channel such as a socket. As soon as the visualizer receives the message, it makes changes in the visualization panel to reflect the latest event. The visualization panel will then indicate to the user the latest monitor event in the application program such as a thread joining the entry queue, entering a monitor or waiting in a conditional queue etc. In this way, the status of the threads with respect to the monitor in the application program is converted to messages and sent to the visualizer to be displayed in the visualization panel.

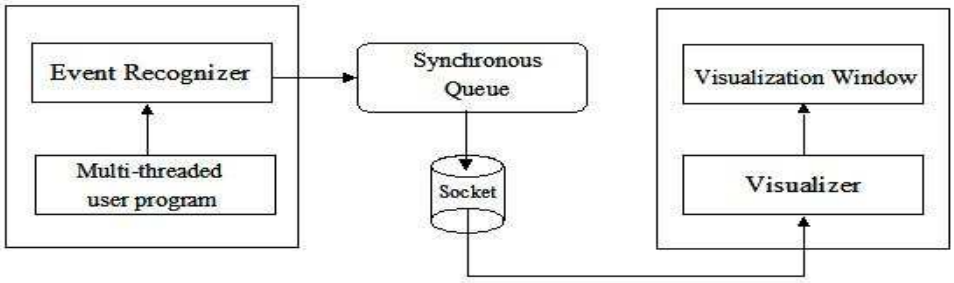

Figure 5.8 Components of Visualization Tool

5.2 Visualizer

The visualizer consists of the core logic that controls the visualization process. The work of the visualizer is to read the messages sent from the user's application program and visualize it in the visualization panel. The visualization is done according to the visualization rules. For each message, there is a corresponding rule that governs how it is interpreted by the visualizer. Each thread is represented by an abbreviation of its name in the visualization panel. The rules are explained below:

- Join: When a *join monitor* message is received, the panel displays the thread joining possible other threads in the entry queue. This message is treated in the same way by monitors with SC, SU and SE signaling discipline.
- EnterMonitor: This message signifies the calling thread entering the monitor. This message again is visualized in the same way regardless of the monitor type.
- SignalCall: This message is typically generated by monitors of SC signaling discipline. When the parser generates this message, it also includes information such as the name of the conditional queue associated with the method call. Since the conditional queue textboxes are pre-labeled with the names, a thread name is moved from the appropriate conditional queue textbox to the entry queue, where it will compete with other threads to enter the monitor.

- SignalC-and-exitMonitor: Monitors of type SU and SE generate this message. The corresponding conditional queue name is included in the message. This message indicates that the signaling thread will immediately exit the monitor after calling this method. In the case of SU monitor, the signaling thread will be moved to the reentry queue and if a thread is waiting on the corresponding conditional queue it is moved to the monitor.
- WaitC: When this message is received the visualization depends on the type of monitor. In the case of a SC monitor, the calling thread is moved from the monitor to the appropriate conditional queue. In the case of SU and SE monitor, if the reentry queue is not empty, the thread from it is moved to the monitor, otherwise a thread from the entry queue is moved to the monitor. The calling thread moves to the conditional queue.
- ExitMonitor: When this message is received, the name of the thread is removed from the monitor textbox.

The visualization rules control the visualization process and they are separated from the modules that create the messages and send them to the visualization panel. This ensures scalability so that it is easier to add new rules or modify existing rules.

CHAPTER 6

INTEGRATING JSWAT AND VISUALIZATION TOOL

This chapter explains in detail how the visualization tool is integrated with JSWAT. The first step in integration is to design and develop the graphical user interface through which users can interact with the visualization tool. The output that is conveyed to the users by the visualization tool is a graphical display of the working of threads and monitor. JSWAT is implemented as an integration of the following modules:

- JSwat BCEL Library
- JSwat Command
- JSwat Core
- JSwat Debugger
- JSwat Help
- JSwat Interface
- JSwat Java Parser
- JSwat Nodes
- JSwat Product Definition
- JSwat Views

Each module implements the corresponding feature of JSWAT. The visualization tool is implemented by augmenting the files in *com.bluemarsh.jswat.ui.components* package in *JSwat Interface* module. The visualization tool has been made available as a feature of JSWAT. The visualizer is invoked by selecting Monitor Visualization option in Window drop-down menu. The user then selects the option from the menu to run the debugger or clicks the run button. This launches the Launch Debugger Panel. This panel is used to capture the classpath, sourcepath and

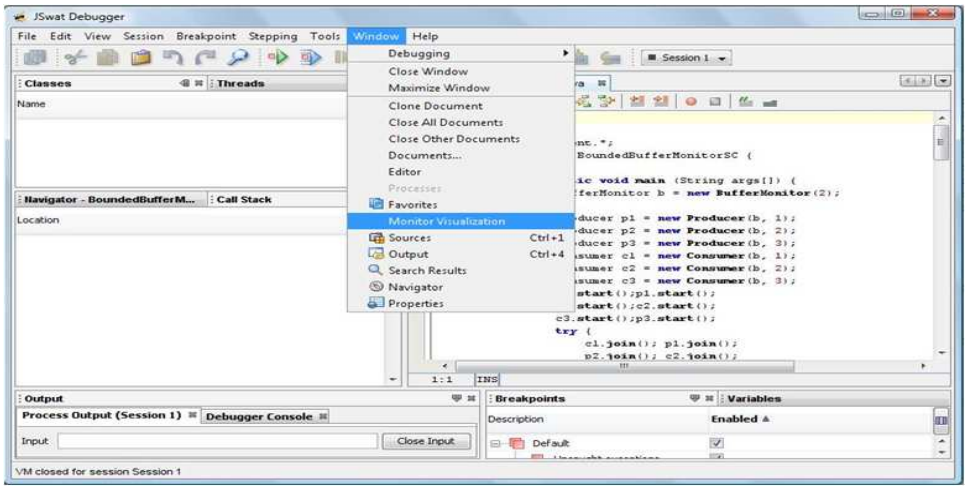

Figure 6.1 Opening Visualization Window

related JARs and folders. JSWAT and the visualization tool are integrated in such a way that when the JSWAT pauses at a breakpoint, the visualization tool also pauses to display the status of the threads with respect to the monitor at that point in the visualization panel. That is, it displays if the thread is inside the monitor or in a queue.

6.1 Monitor Visualization

The user opts for the visualization tool by selecting *Monitor Specification* from Windows menu in the menu bar, as in Figure 6.1. This action opens the visualization window. The visualization window is a panel where the monitor execution is visualized. The panel is initialized with graphical constructs depicting an entry queue and monitor.

The Monitor Specification menu is added to the Windows menu by creating a java file, *NewVisualizationWinAction.java* in *com.bluemarsh.jswat.ui.components* package. The main class in this java file extends the *CallableSystemAction* class. Using this class, the action to be performed when the menu is selected is specified.

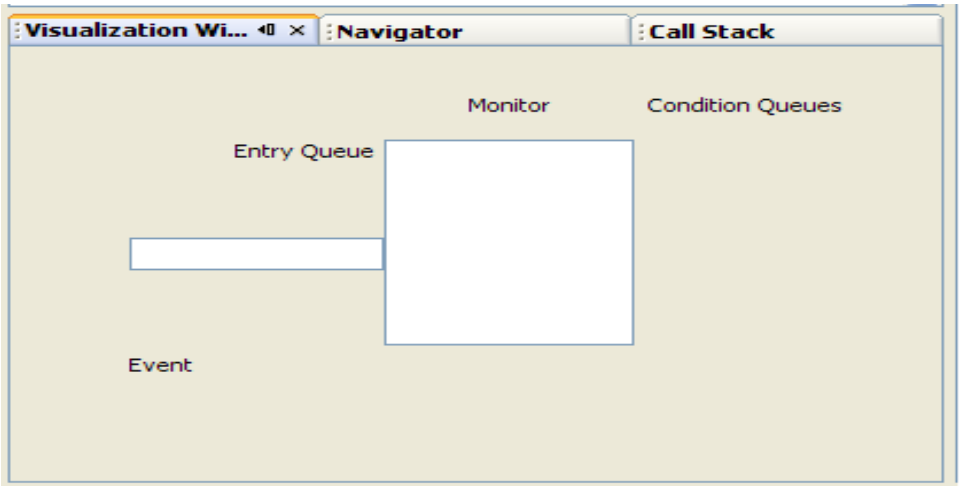

Figure 6.2 Visualization Window

6.2 Visualization Window

The visualization window is a tabbed panel as seen in Figure 6.2 and 6.3. Apart from the entry queue and the monitor, conditional queues and reentry queue are also displayed here. The number of conditional queues is determined by the tool itself. Based on this number, conditional queue text boxes are displayed. The tool also identifies the names of the conditional queues. Each event in the monitor execution is interpreted by the application and depicted in this panel. The event that is being interpreted is displayed in a label in real time.

Each thread in the user's multi-threaded application is represented by an abbreviation of its name. For example, when a thread named *Producer1* executes a method to enter the monitor, the text *P1* will be displayed inside the monitor textbox. This will intuitively inform the user that the thread is inside the monitor at that point of time.

The visualization window is implemented by creating a window component in *com.bluemarsh.jswat.ui.components* package. The window component has both source and design views. In the design view, the UI components are designed. In the source view, the action to be performed by the UI components are specified.

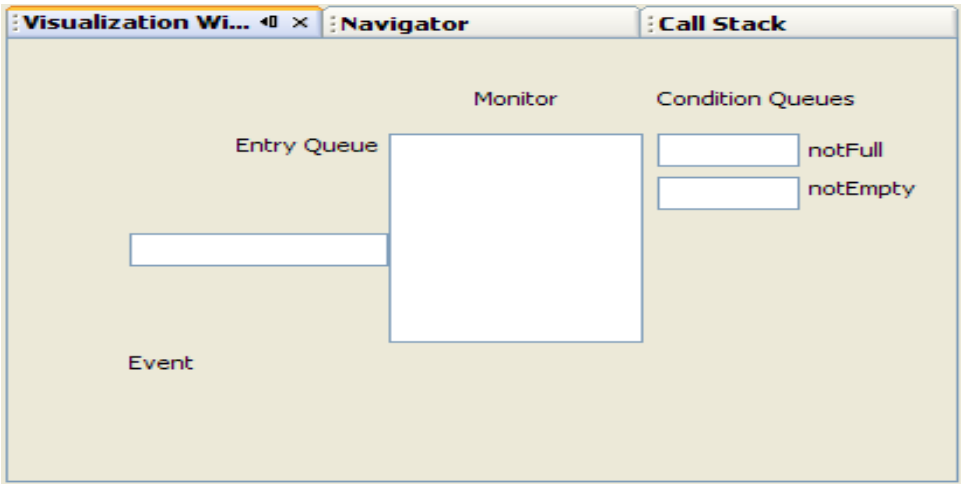

Figure 6.3 Visualization Window

6.3 Launch Debuggee Panel

The Launch Debuggee panel is used to capture vital information for JSWAT to run a program. In the Parameters tab of the panel, the user enters the JRE runtime, JVM arguments, class name and class arguments if any, as seen in Figure 6.4. In the Classpath tab, the user enters JARs and folders that the application requires to execute. It is necessary to list the dependencies in the right order for correct execution, refer Figure 6.5. The visualizer uses this information to identify the concurrent program that is to be visualized. When a user opts for visualization, Launch Debuggee panel handles initialization of the visualization window with the pertinent display structures. This panel is implemented in LaunchDebuggeePanel.java file in *com.bluemarsh.jswat.ui.components* package. The file contains the constructor for initializing the UI components of the panel. The input obtained by the panel is set by *loadParameters()* . The session parameters are then attached to each debugging session by *saveParameters()* using the PathManager. The sourcepath obtained in the launch debuggee panel is passed to the visualization panel through the PathManager.

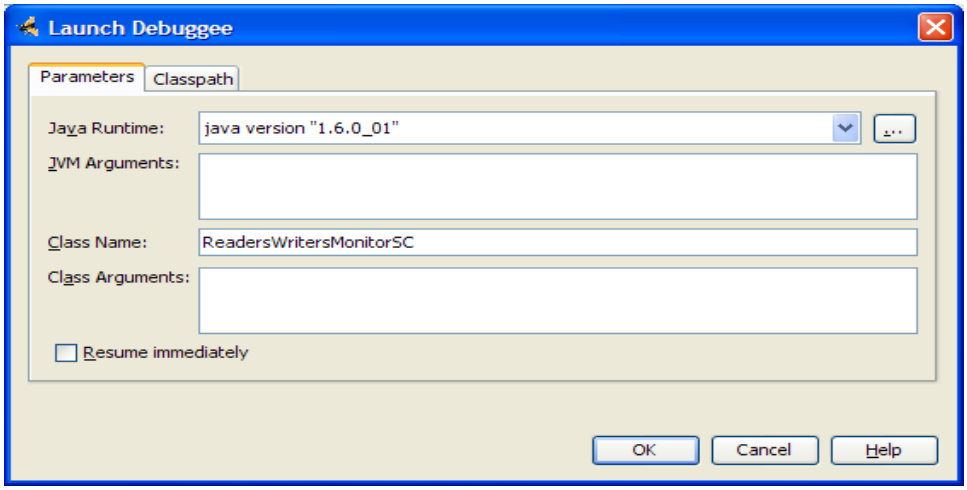

Figure 6.4 Parameters tab in Launch Debuggee Panel

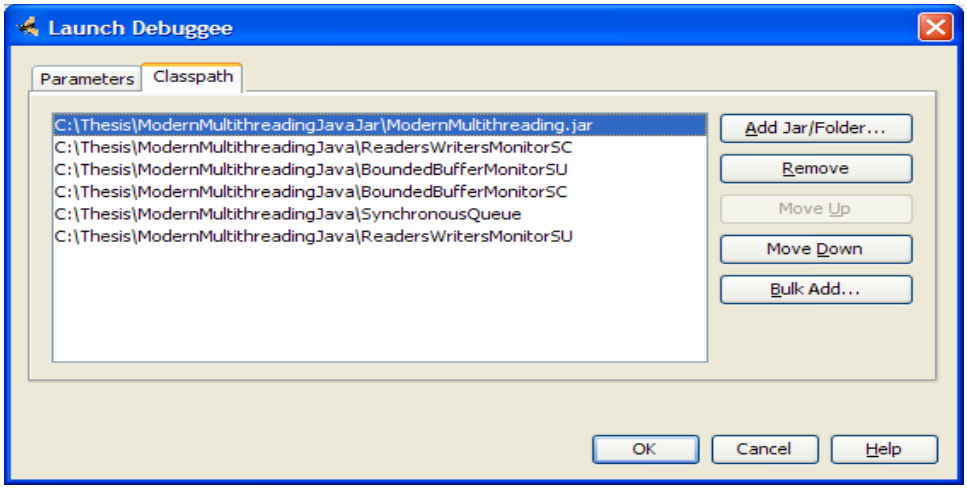

Figure 6.5 Classpath tab in Launch Debuggee Panel

6.4 Visualizer and User Application Interface

A communication interface has to be constructed between the visualizer and the user application so that events can be captured in the user application and displayed in the visualizer. This communication is achieved using a client-server paradigm. As soon as the visualizer is initialized, it creates a socket and waits for data to arrive through the socket.

The user application acts as the client which sends information regarding the events that have to be visualized by the visualizer. The visualizer contains a multi-threaded server to handle the event messages from the user application.

This approach ensures that the visualizer is not tied up with the communication process alone and can handle visualization tasks simultaneously. Once the user application starts, the visualizer and the application are synched using a common port number. The user application sends a request to connect to the server. The server spawns a separate thread to handle this request. This child thread creates a listen socket to communicate with the application.

Each event that has to be visualized is converted to a message and sent through the socket to the server. Therefore, one thread is responsible for accepting event messages and another thread interprets those messages and invokes the corresponding graphical action.

Asynchronous sockets are used for communicating between the visualizer server and the application client. Asynchronous sockets have been chosen for this purpose due to several reasons. In the case of asynchronous sockets, the CPU or other resources to be applied to each connection is cooperatively scheduled. This is ideal in this situation since the connections do not require much state. An asynchronous socket is more efficient and elegant. It scales better than a synchronous socket. The server would be able to process multiple requests simultaneously which will not be possible when using a synchronous socket. This type of socket is ideal for graphical user interface applications since the server and the client are not inhibited by each other.

The method to send and receive messages through the socket is synchronized so that different threads in the user‘s multithreaded application do not overwrite the data in the socket. Also the asynchronous socket is created as a static class object. This ensures that all the threads send and receive only through a single, common socket object. When the visualization is complete the sockets are closed and the port numbers are reused for the next execution.

6.5 Synchronous Queues

One of the challenges faced during implementation was that since the socket is a static object and is synchronized, only one thread can access the socket at any given time. Also, the threads can access them only when their turn is assigned by the scheduler. This led the threads to behave in a sequential fashion as opposed to an inter-leaved way. The result of this was that all the messages from a thread were sent in a batch through the socket. The solution to this issue is to queue the messages in the order they are generated and send them across the socket one after the other. This solution is implemented using synchronous queues.

A synchronous queue is a Java blocking queue used for communication and synchronization [5]. It is a blocking queue in which each *put* must wait for a *take* and vice versa. A synchronous queue does not have any internal capacity. The element that the first queued thread adds to the queue becomes the *head* of the queue. If there are no queued threads, then there are no elements in the queue and the head is *null*. The queue does not permit *null* elements though. They are well suited for this design where an object running in one thread much sync up with an object running in another thread in order to hand it some information.

The *SynchronousQueue* class is a member of the Java Collections Framework. The data type of the elements that will be put in the queue must be specified when initializing the queue. For example, if a synchronous queue sq is to be created which accepts strings as its elements, then it should be specified as $SynchronousQueue< String > sq$.

The method *poll()* in this class retrieves and removes the head of this queue, if another thread is currently making an element available. It returns the datatype specified when creating the queue. The method *poll(long timeout, TimeUnit unit)* waits if necessary upto a specified time interval, if an element has not been inserted, before it tries to retrieve the element from the queue. It returns the head of the queue or *null* if an element has not been inserted before the time has elapsed.

The public method *put()* adds the specified element to this queue, waiting if necessary for another thread to receive it. It throws *InterruptedException* if interrupted while waiting and *NullPointerException* if a null value is tried to be inserted. Method *take()* retrieves and removes the head of this queue, waiting if necessary for another thread to insert it. It returns the head of the queue. Using the *put()* method the user application places an event message in the synchronous queue. It is then taken from the queue using *take()* as soon as it is inserted and sent to the server through the socket for visualization.

CHAPTER 7

EXPERIMENTAL RESULTS

7.1 Use Case Diagram

The use case diagram in Figure 7.1 illustrates the different actions the user can peruse in the application.

7.2 Sequence Diagram

The sequence diagram in Figure 7.2 explains the sequence of steps followed in the execution of the visualization tool.

7.3 Sample Visualization Tool Execution With SU Monitor

7.3.1 Visualization of Bounded Buffer Problem

Let us consider a Bounded Buffer implementation for illustrating the working of the visualization tool (refer Figure 7.3). The solution employs a SU monitor for solving the Bounded Buffer problem. The Buffer consists of two slots. There are three producers and three consumers. The producer threads are P1, P2 and P3 respectively. The consumer threads are C1, C2 and C3 respectively.

The first step is to invoke the LaunchDebuggee panel by clicking the green arrow button in the menu bar as in Figure 7.4. Then the main class name is entered under the Parameters tab.

The next step is then to enter the classpath dependencies as in Figure. 7.5.

There are three producers and consumers created each. The producer and consumer threads are queued in the entry queue as in Figure. 7.6.

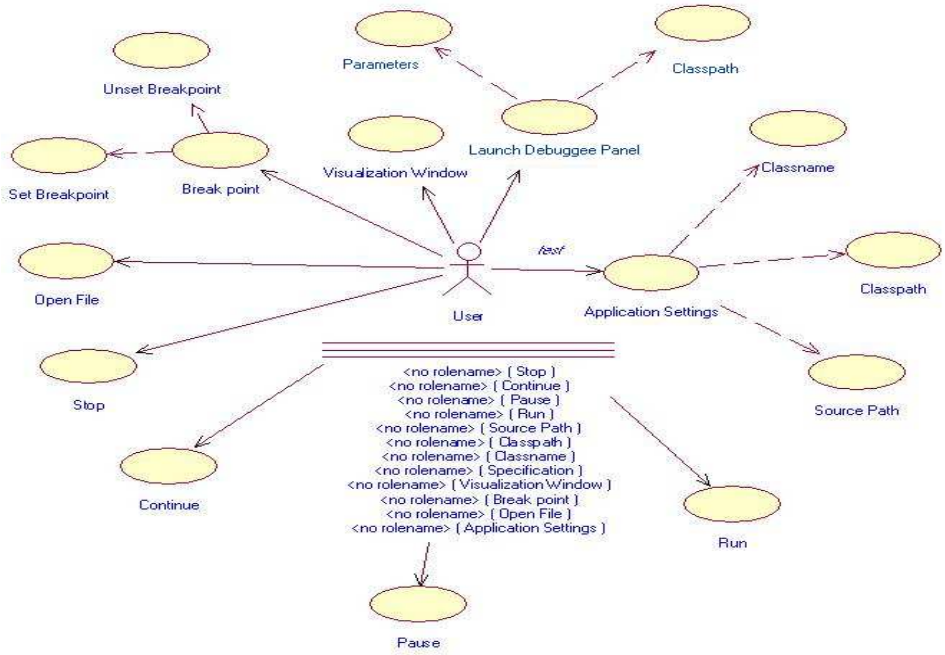

Figure 7.1 Use Case Diagram

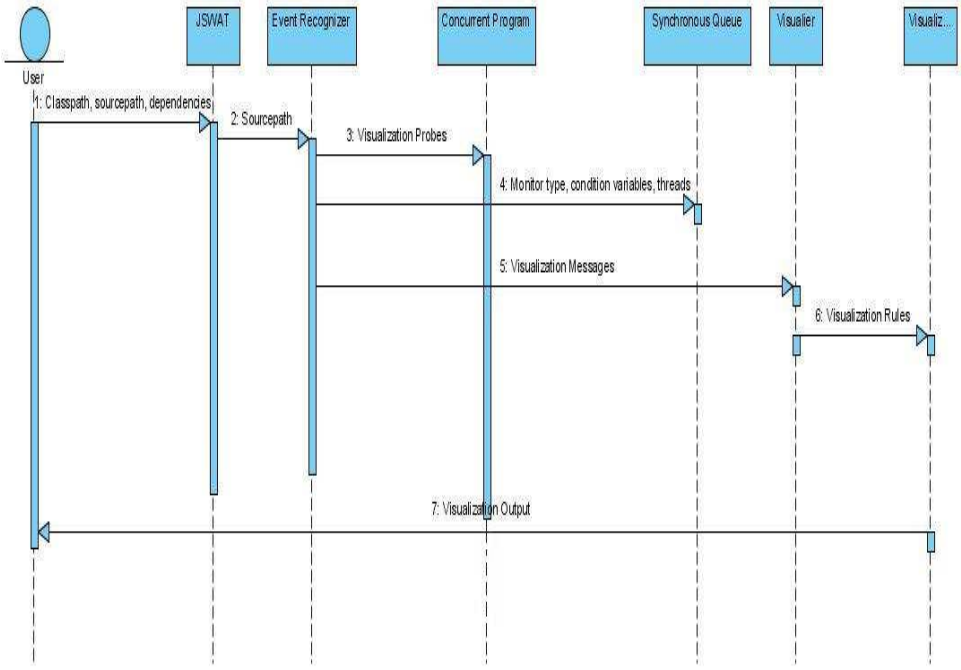

Figure 7.2 Sequence Diagram

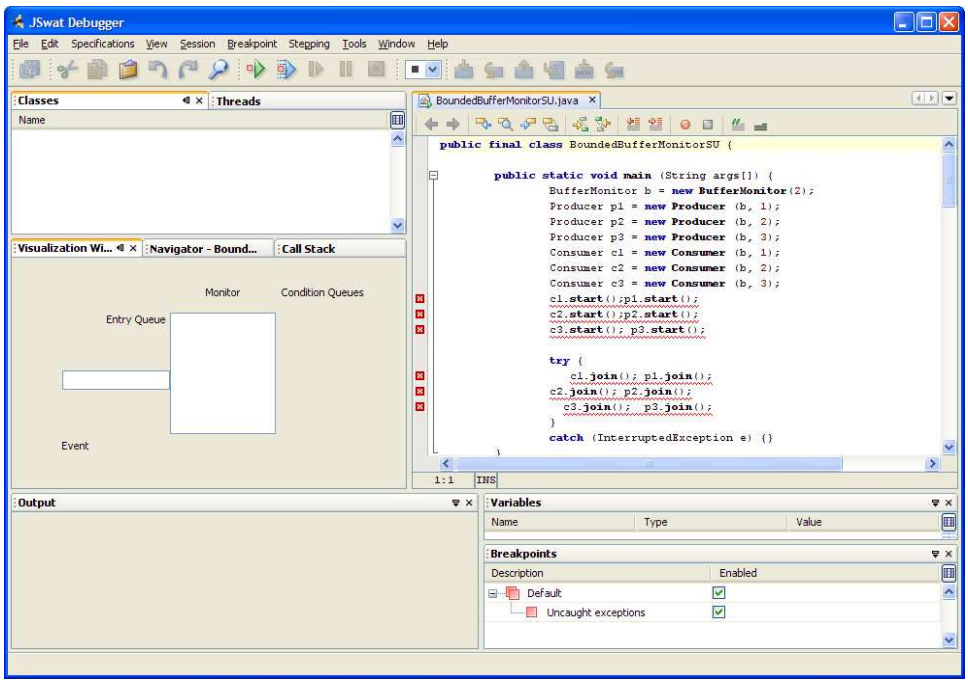

Figure 7.3 Monitor Visualization menu

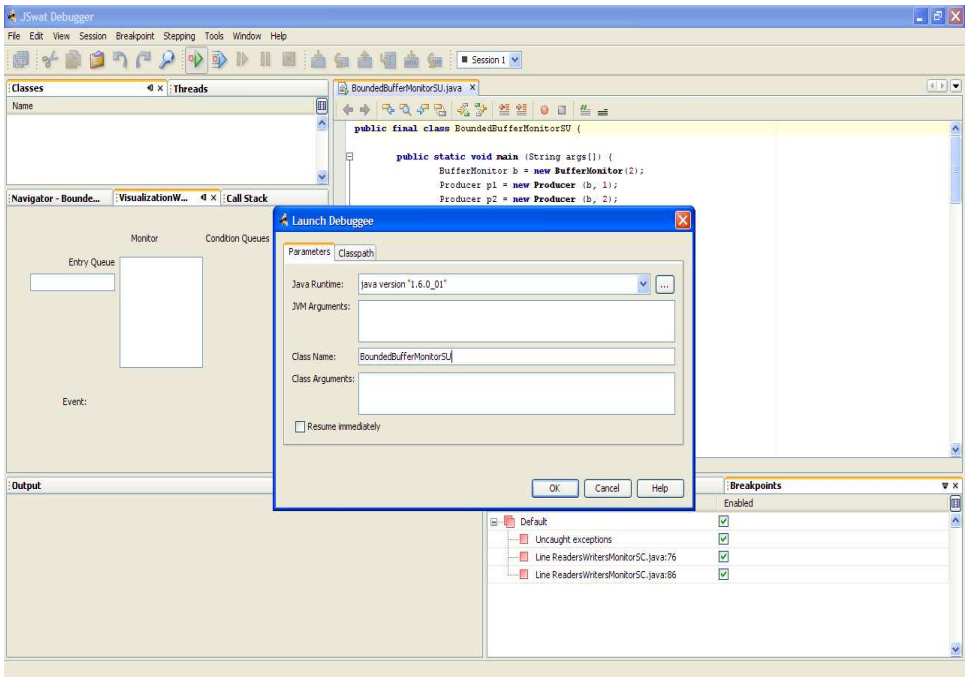

Figure 7.4 Launch Debuggee Panel: Specifying classname

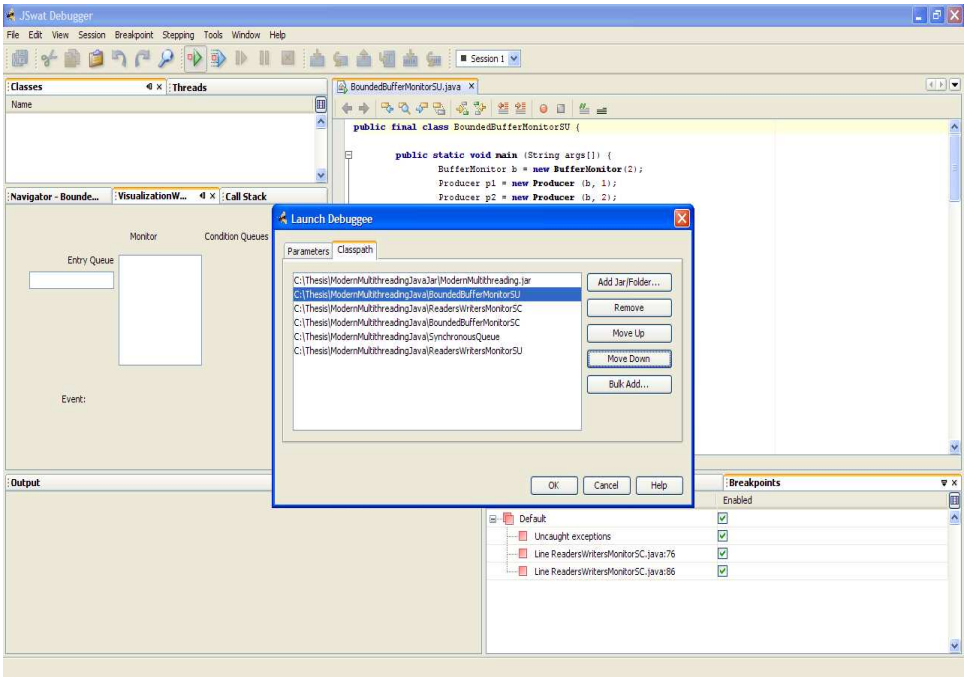

Figure 7.5 Launch Debuggee Panel: Specifying dependencies

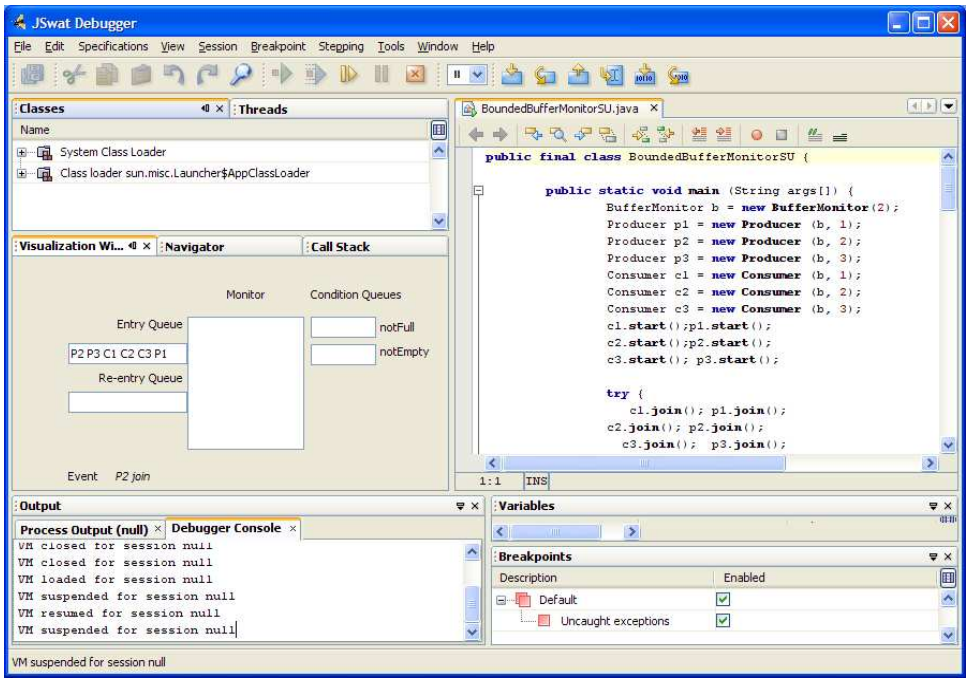

Figure 7.6 Producers and consumers lined up in the entry queue.

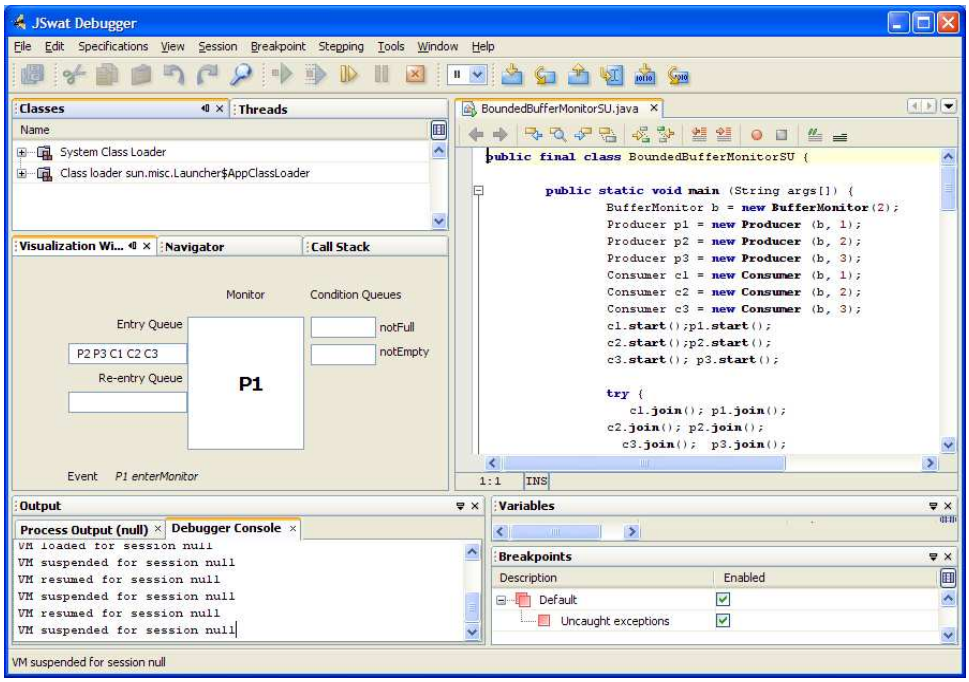

Figure 7.7 P1 entering the monitor

The first thread which is a producer thread enters the monitor. The producer is denoted as P1 and it is depicted inside the monitor textbox as in Figure. 7.7.

P1 executes *signal-and-exitMonitor()* and exits the monitor as in Figure. 7.8. There are no threads queued in the reentry queue so there are no threads awakened from that queue. In the next step, the next thread which is C3 attempts to enter the monitor. Since the previous thread P1 has deposited, the consumer is allowed to enter the monitor. So C3 enters the monitor, consumes and exits the monitor.

The next thread C2 attempts to enter the monitor but it is not allowed to enter the monitor since there are no full slots. So it is queued in the notEmpty conditional queue. This is depicted by painting C2 in the notEmpty textbox as in Figure. 7.9.

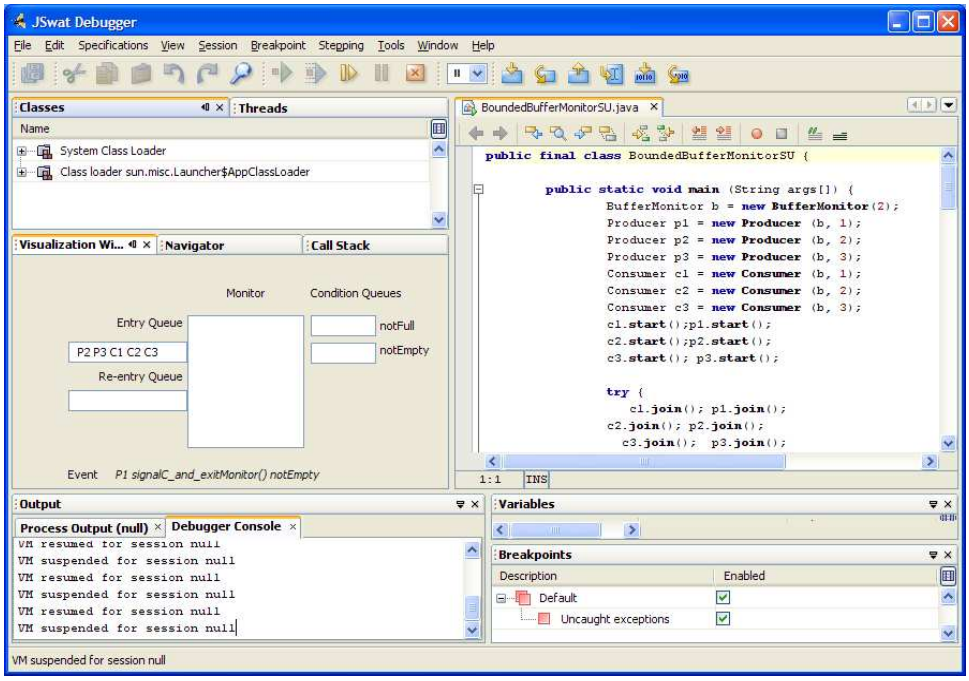

Figure 7.8 P1 exiting the monitor

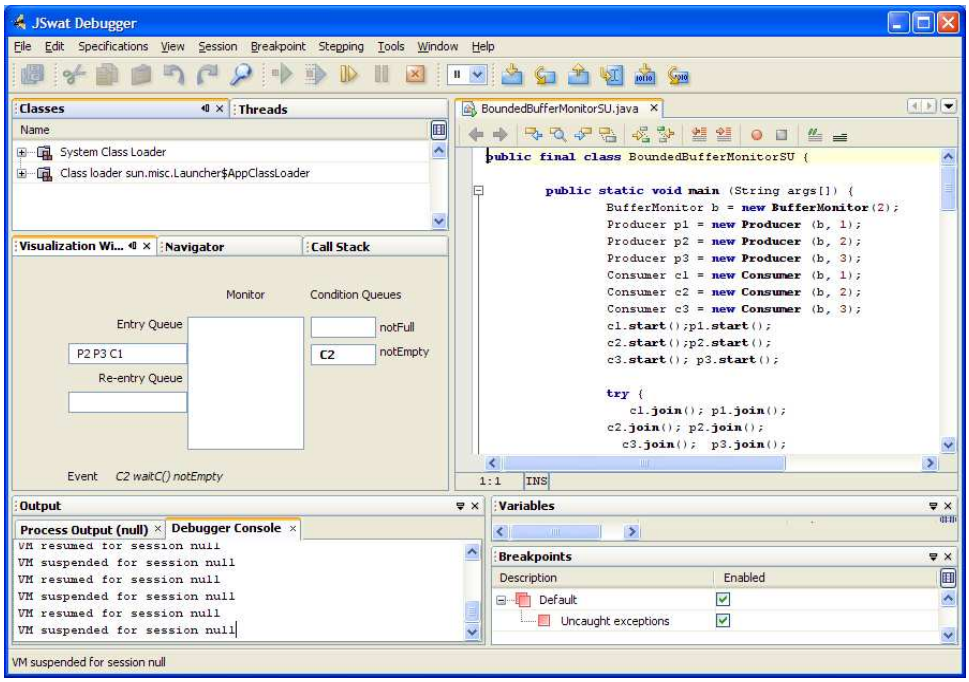

Figure 7.9 C2 has withdrawn from buffer. C3 waiting in notEmpty queue

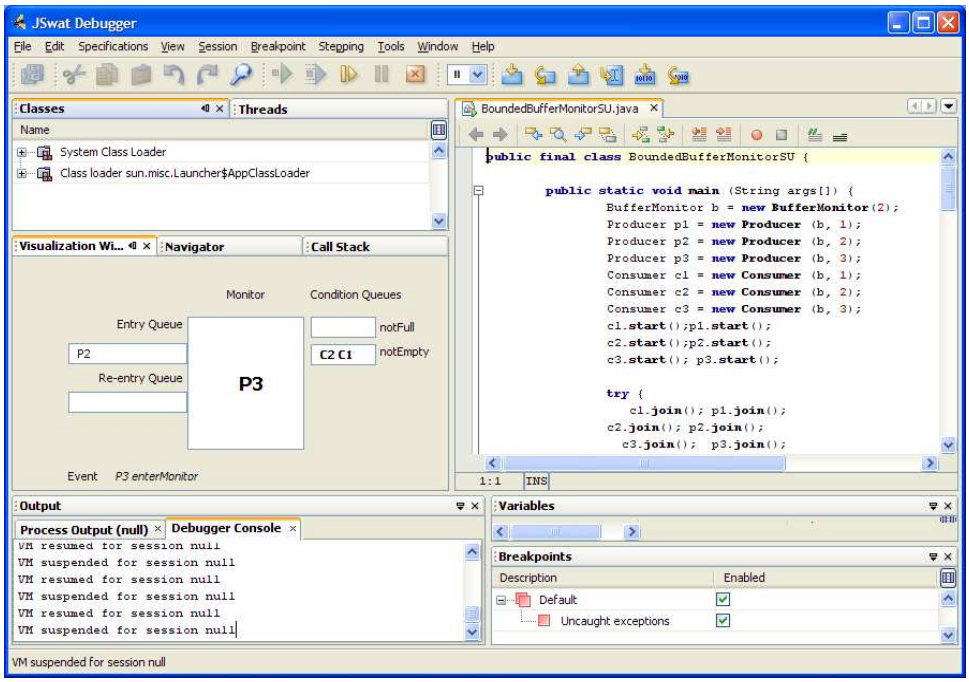

Figure 7.10 P3 entering the monitor.C1 and C3 wait in notEmpty queue

The next thread in the entry queue is C1. Since slots are still empty, C1 is also queued in the notEmpty conditional queue along with C2. The next thread in the entry queue, P3 enters the monitor as in Figure. 7.10.

P3 deposits in the monitor. Before P3 exits, it checks that the notEmpty conditional queue is not empty and awakens one thread from that queue. P3 then waits in the reentry queue as in Figure. 7.11.

C2 is awakened by P3 and enters the monitor as in Figure. 7.12.

C2 consumes and awakens P3 waiting in the reentry queue. It then exits the monitor as in Figure. 7.13.

In the next step, a thread is awakened from the entry queue. Therefore, P2 enters the monitor as in Figure. 7.14.

In Figure. 7.15, P2 checks the notEmpty conditional queue and finds it not empty. Therefore, it awakens C1 and waits in the reentry queue.

In Figure. 7.16, C1 enters the monitor, awakens the thread in the reentry queue and exits.

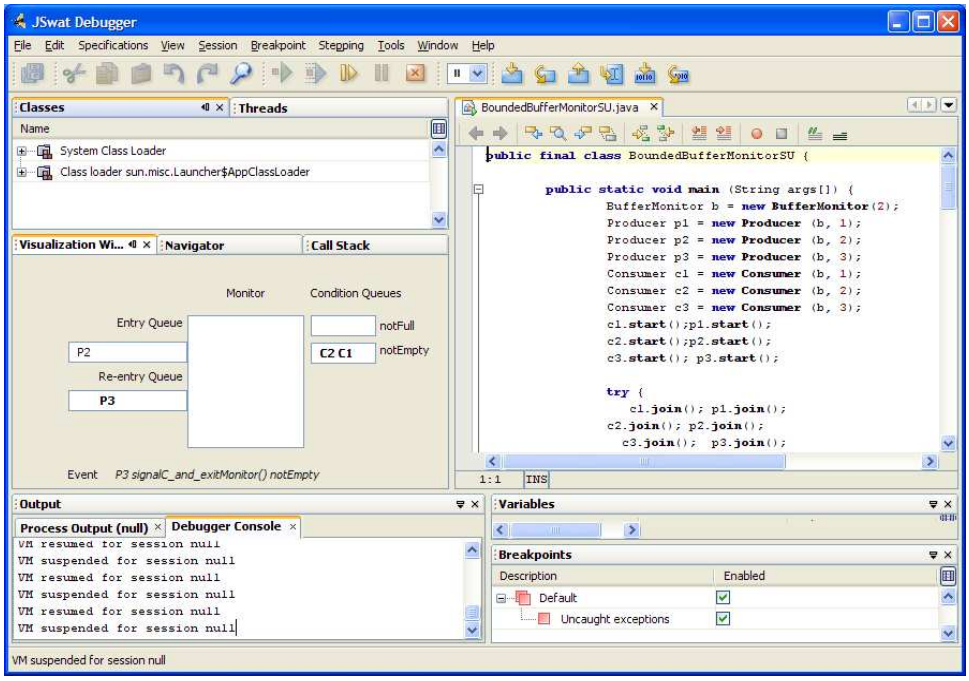

Figure 7.11 P3 waiting in reentry queue since notEmpty queue is not empty

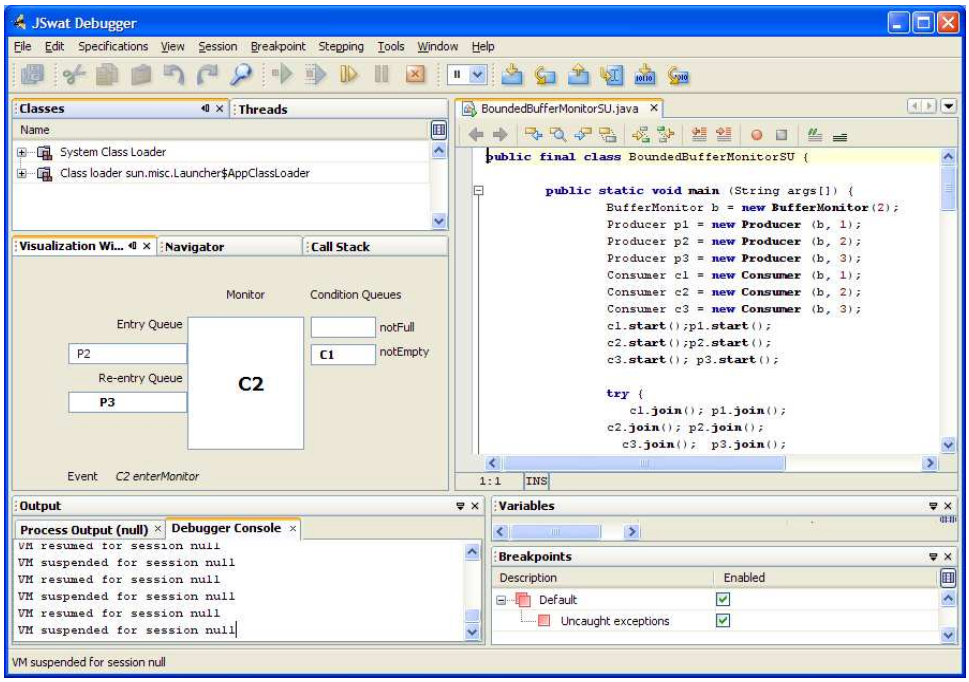

Figure 7.12 P3 signalling C2 from conditional queue after depositing

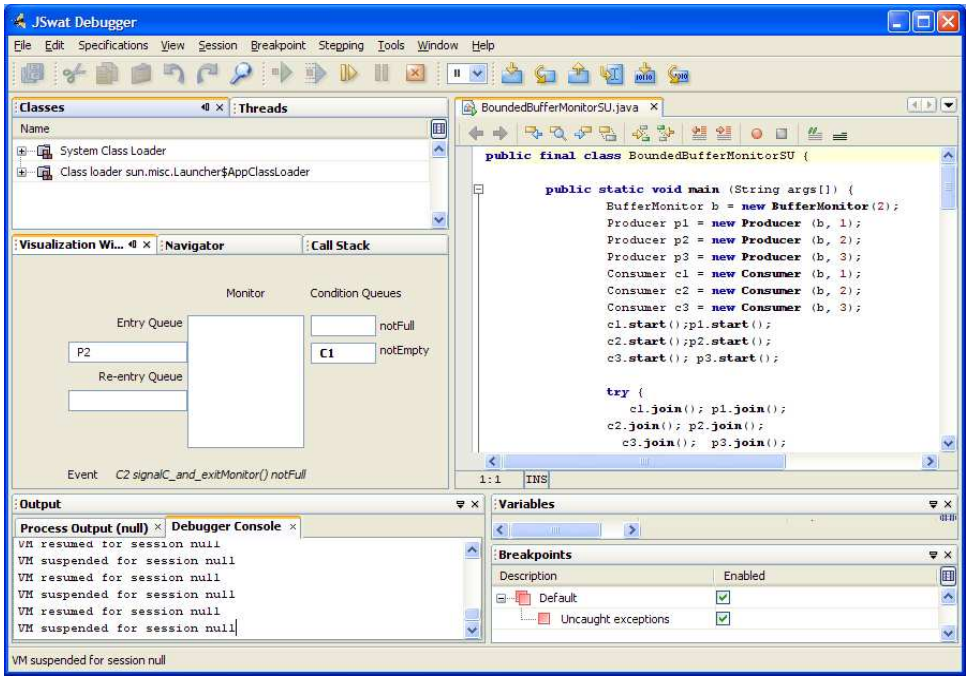

Figure 7.13 C2 signalling P3 from reentry queue before exiting monitor

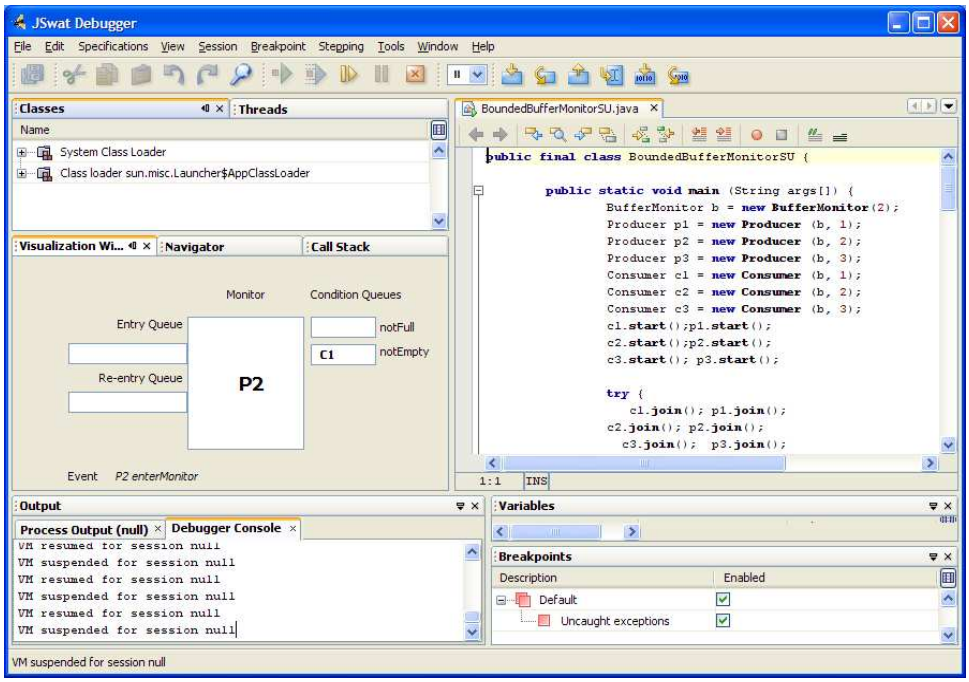

Figure 7.14 P2 entering the monitor

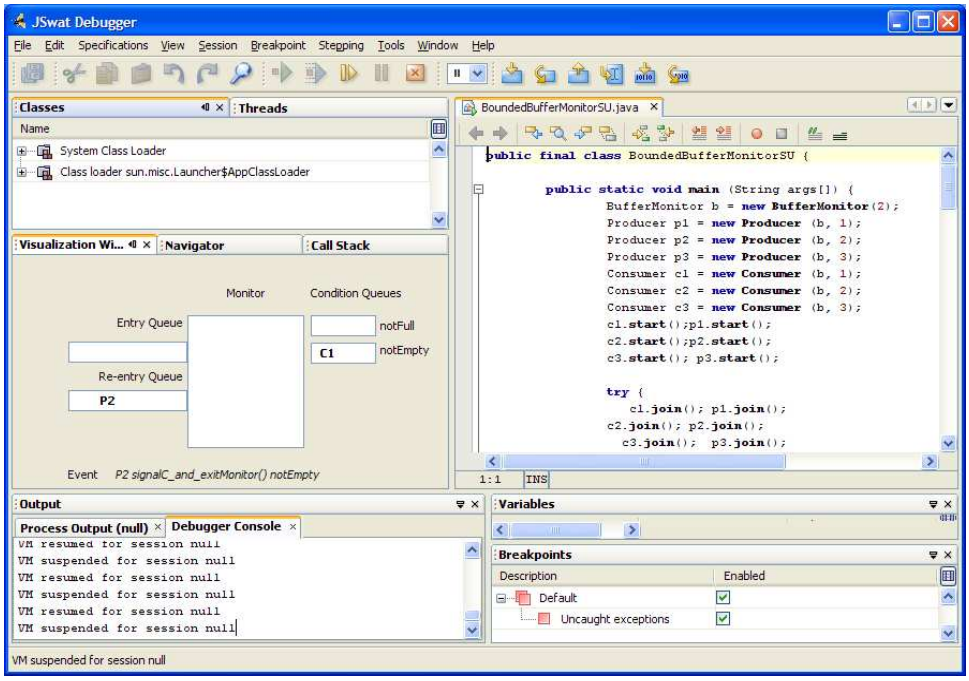

Figure 7.15 P2 waiting in reentry queue since notEmpty queue is not empty

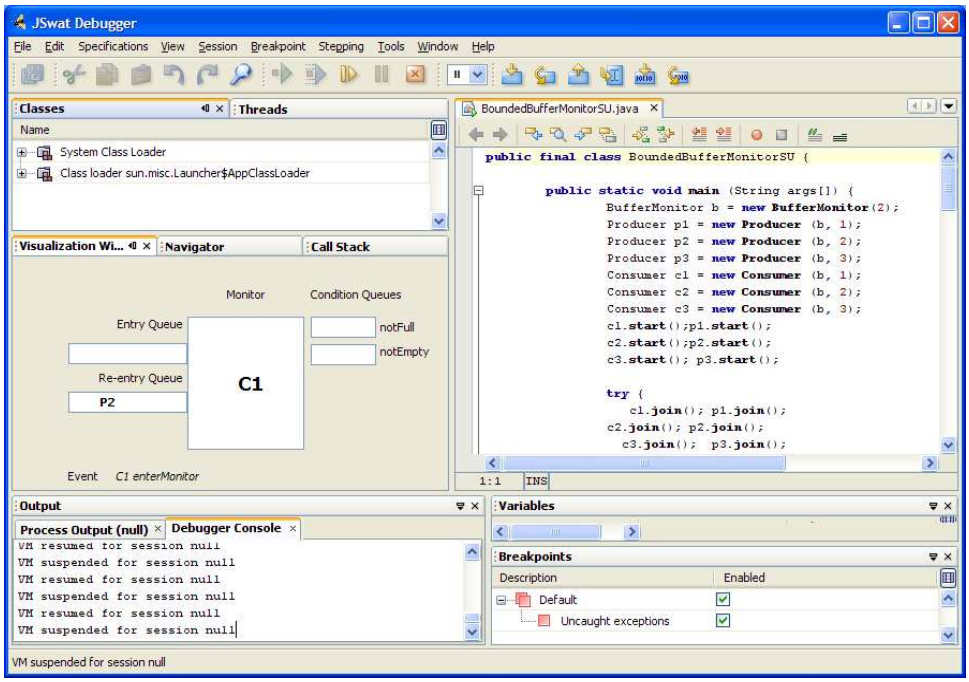

Figure 7.16 C1 entering the monitor after signalled by P2

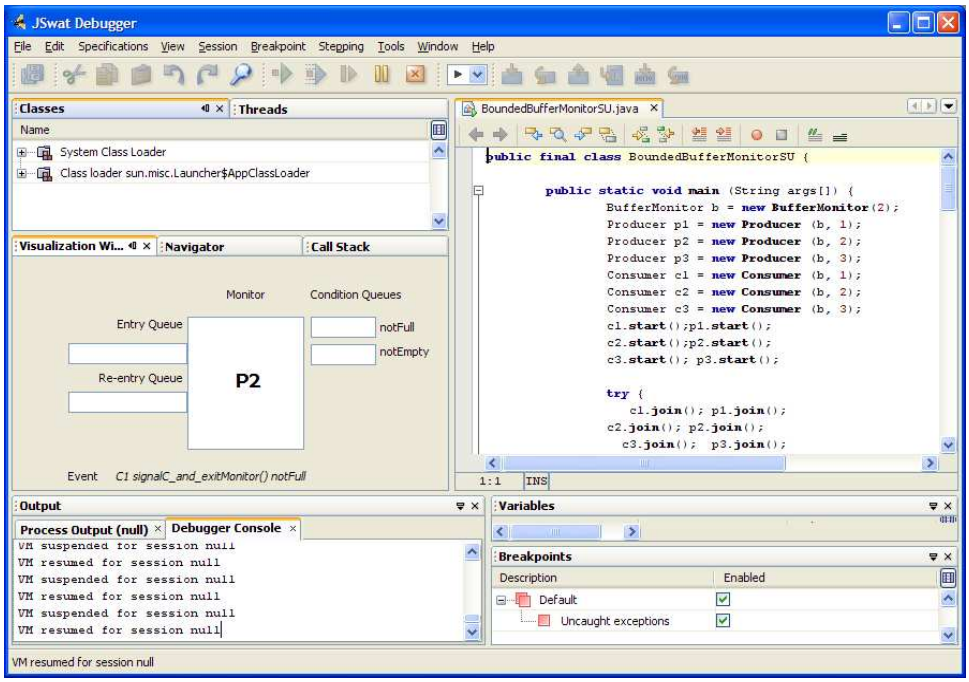

Figure 7.17 P2 reenters the monitor after being signaled by C1 and P2 exits.

In Figure. 7.17, P2 is awakened from the monitor and it exits the monitor.

7.3.2 Synchronous Queue Implementation

A synchronous queue is a blocking queue construct implemented in the class *SynchronousQueue* belonging to *java.util.concurrent* package. Since synchronous queues have been extensively dealt with in Chapter 6, let us proceed to the implementation. This synchronous queue implementation takes the Bounded Buffer problem as the basic framework. There are three producers and consumers and they enter and exit the monitor through the queue. A thread depicting a producer is named as Putter and a thread depicting a consumer is named as Taker. The threads are named after the frequently used *SynchronousQueue* operations - *put()* and *take()*. A Putter thread deposits in the buffer using *put()* and a Taker thread withdraws from the buffer using *take()*. The conditional queues are inCapacity

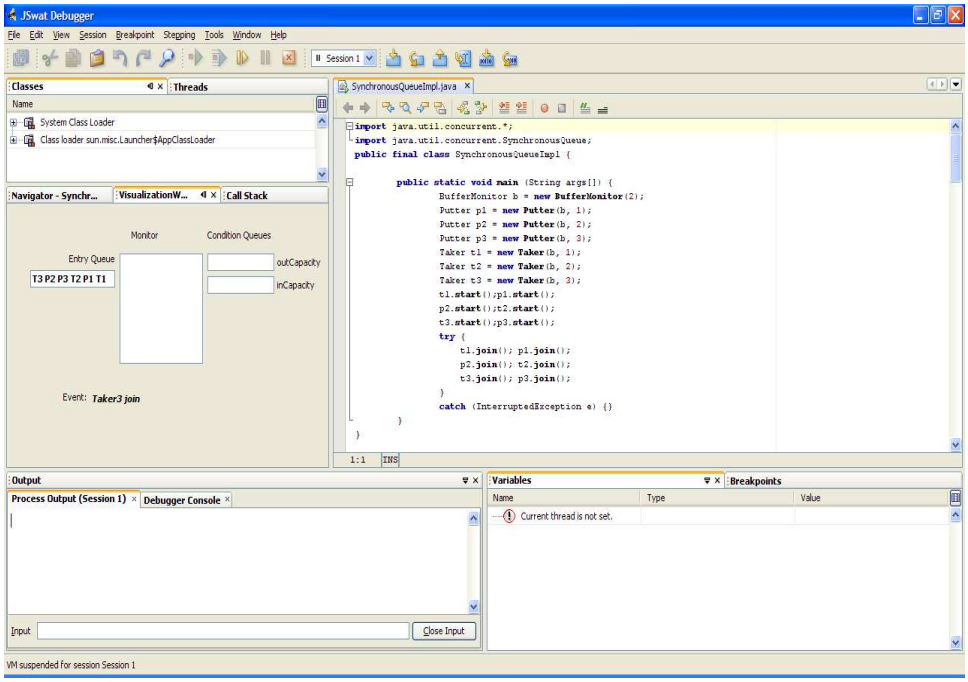

Figure 7.18 All threads wait in the entry queue

and outCapacity. In the first step, all the threads are queued in the entry queue as in Figure. 7.18.

In Figure. 7.19, T3 attempts to enter the monitor but the it is not allowed to consume since there are no full slots. Therefore, it waits in incapacity conditional queue.

In the next step, T2 tries to enter the monitor from the entry queue and is made to wait in the incapacity conditional queue as shown in Figure. 7.20.

In the next step, T1 tries to enter the monitor from the entry queue and is made to wait in the incapacity conditional queue as shown in Figure. 7.21.

In Figure. 7.22, P3 is allowed to enter the monitor and deposit since there are empty slots in the buffer.

After depositing, P3 checks the incapacity conditional queue and finds that is not empty. Hence, it signals all the threads in the conditional queue. Therefore, threads T1, T2 and T3 are moved to the entry queue as in Figure. 7.23. P3 then exits the monitor.

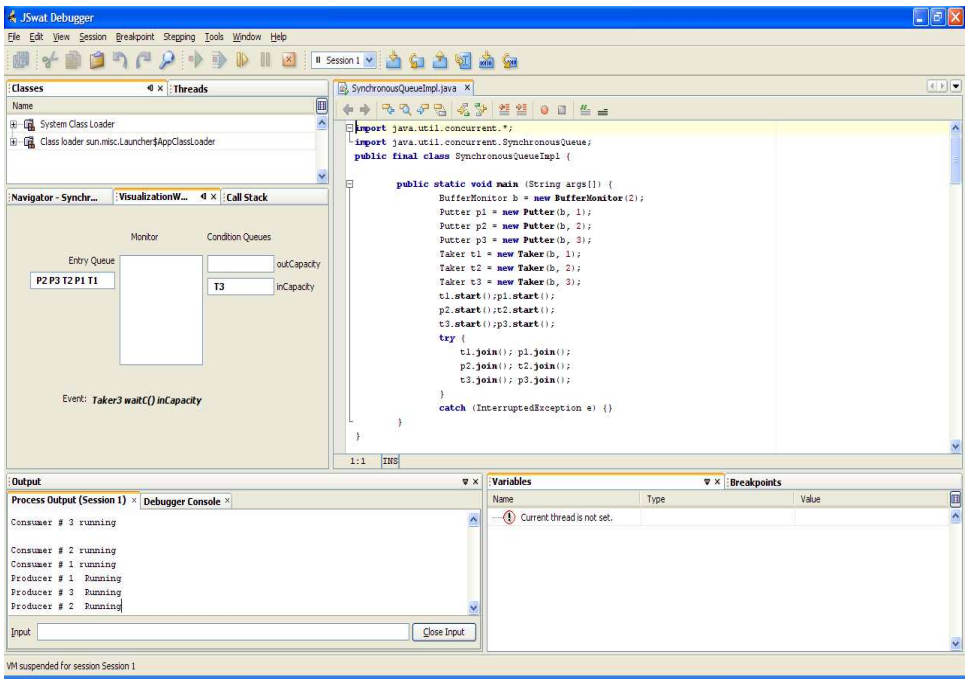

Figure 7.19 T3 waits in conditional queue.

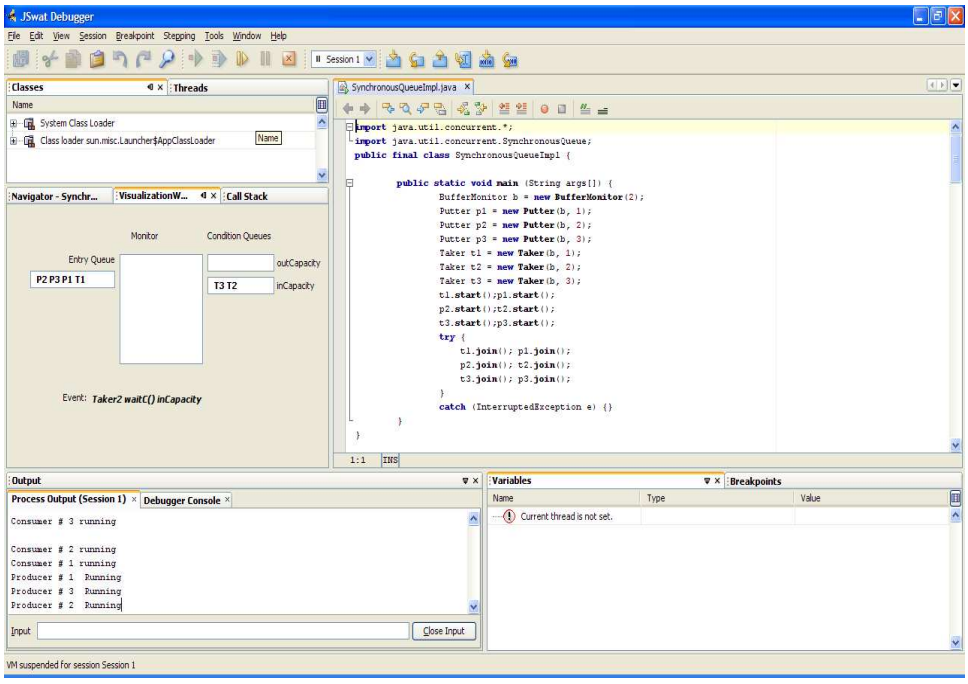

Figure 7.20 T2 joins T3 in the conditional queue.

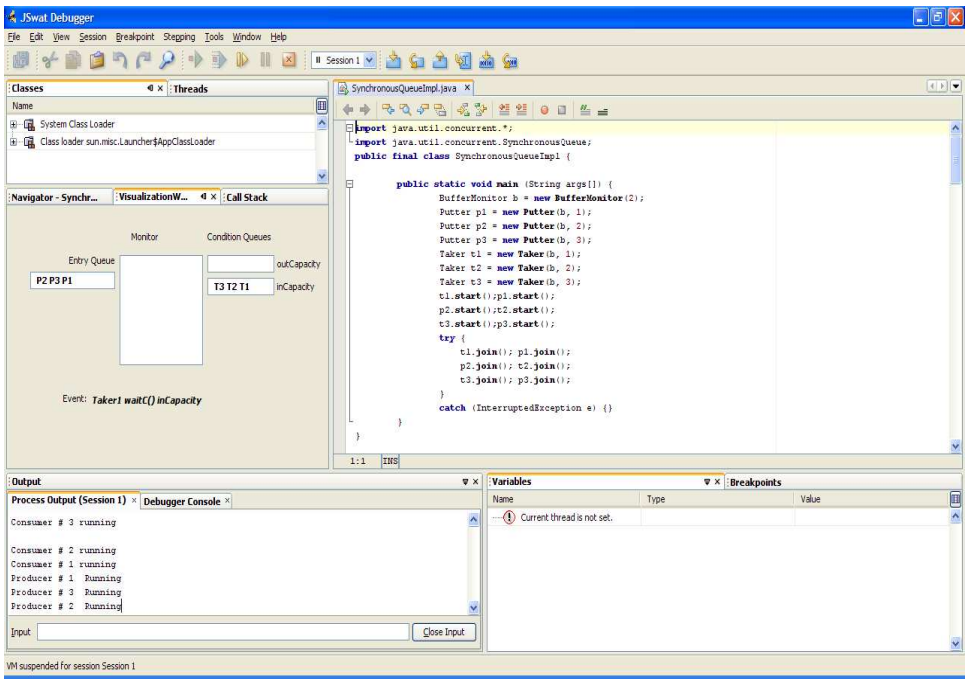

Figure 7.21 T1 waits in the conditional queue with T2 and T3.

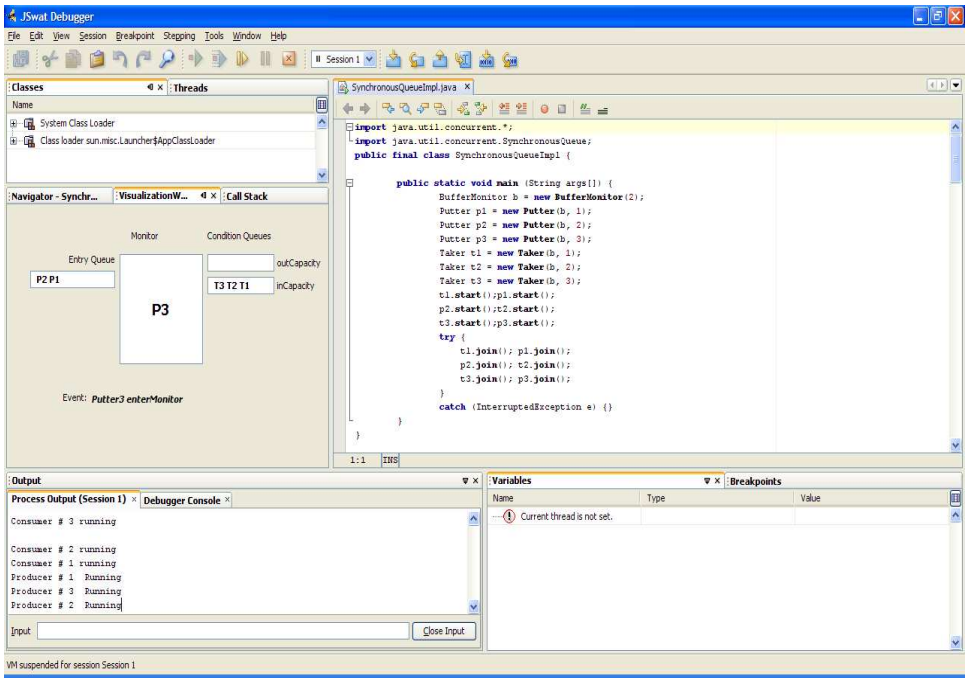

Figure 7.22 P3 enters the monitor.

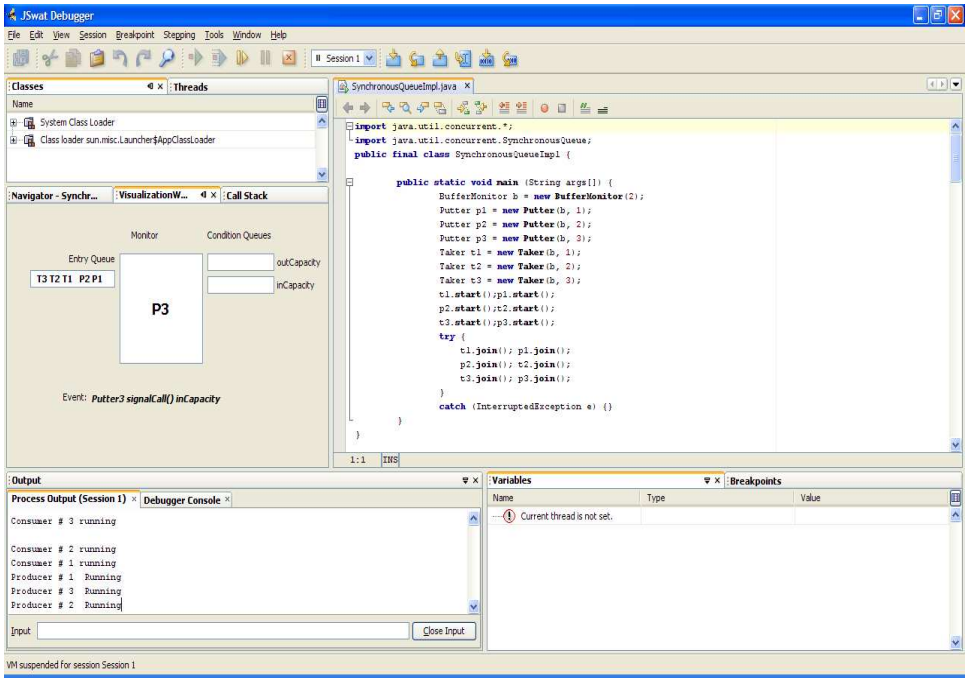

Figure 7.23 P3 awakens the threads from the conditional queue.

In Figure. 7.24, it is shown that P2 is the next thread to enter the monitor.

After depositing, P2 checks the conditional queue and since it is empty, it exits the monitor, as shown in Figure. 7.25.

The next thread in the entry queue, P1 attempts to enter the monitor. Since all the slots in the buffer are full, P2 is made to wait in the outCapacity conditional queue, as in Figure. 7.26.

The next thread T3 enters the monitor in Figure. 7.27.

After consuming, T3 signals the conditional queue in Figure. 7.28.

T3 then proceeds to exit the monitor in Figure. 7.29.

In Figure. 7.30, T1 enters the monitor from the entry queue.

After consuming, T1 signals the conditional queue and exits the monitor, refer Figure. 7.31.

In Figure. 7.32, T2 tries to enter the monitor but since all the slots are empty, it has to wait in the outCapacity conditional queue.

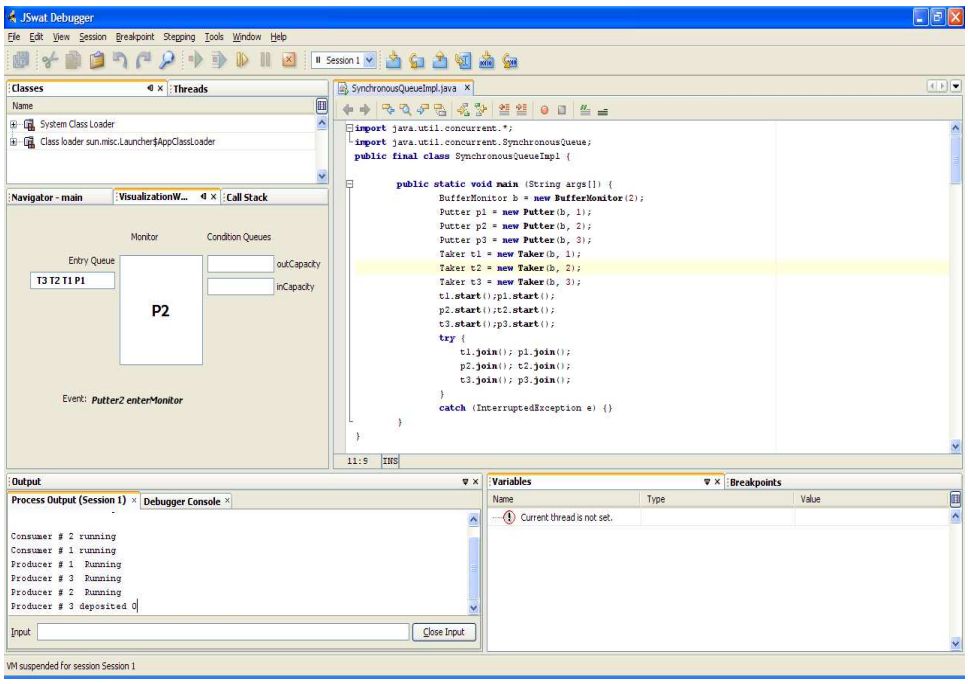

Figure 7.24 P2 enters the monitor.

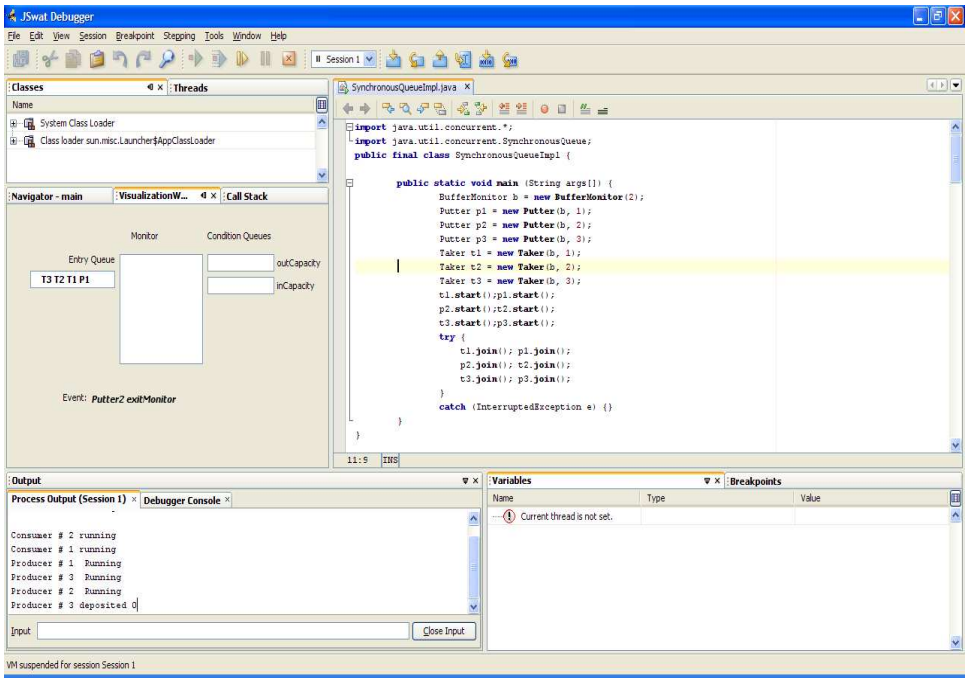

Figure 7.25 P2 exits the monitor.

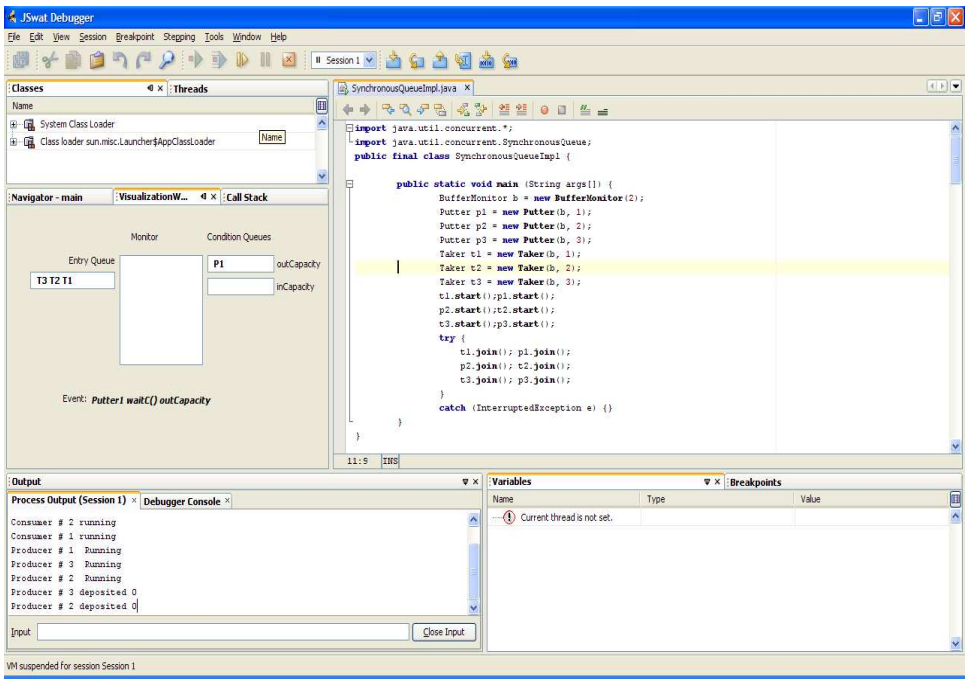

Figure 7.26 P1 waiting in conditional queue.

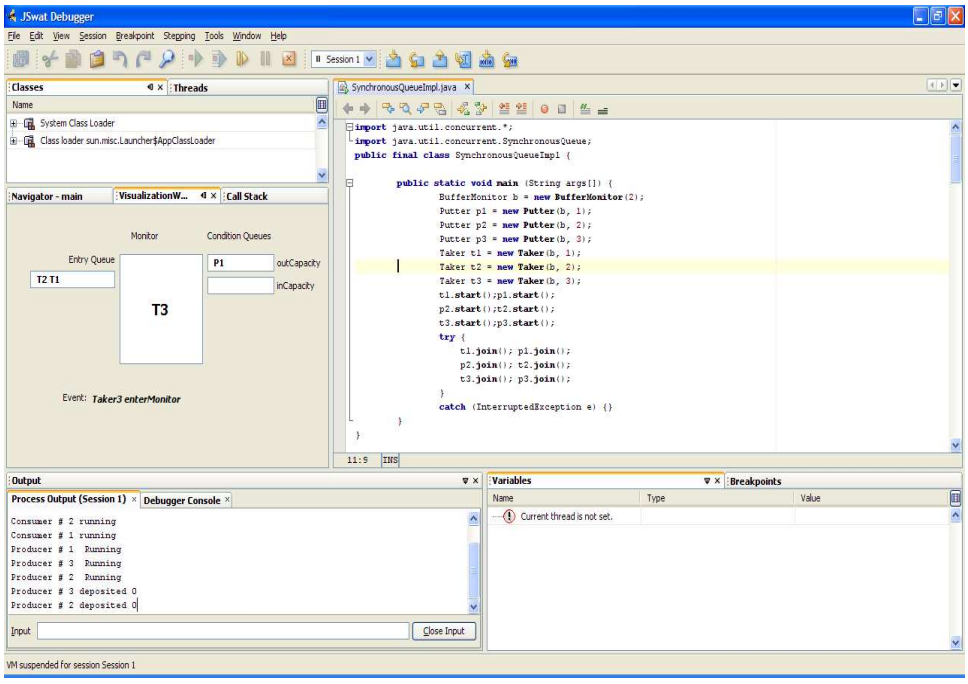

Figure 7.27 T3 enters the monitor.

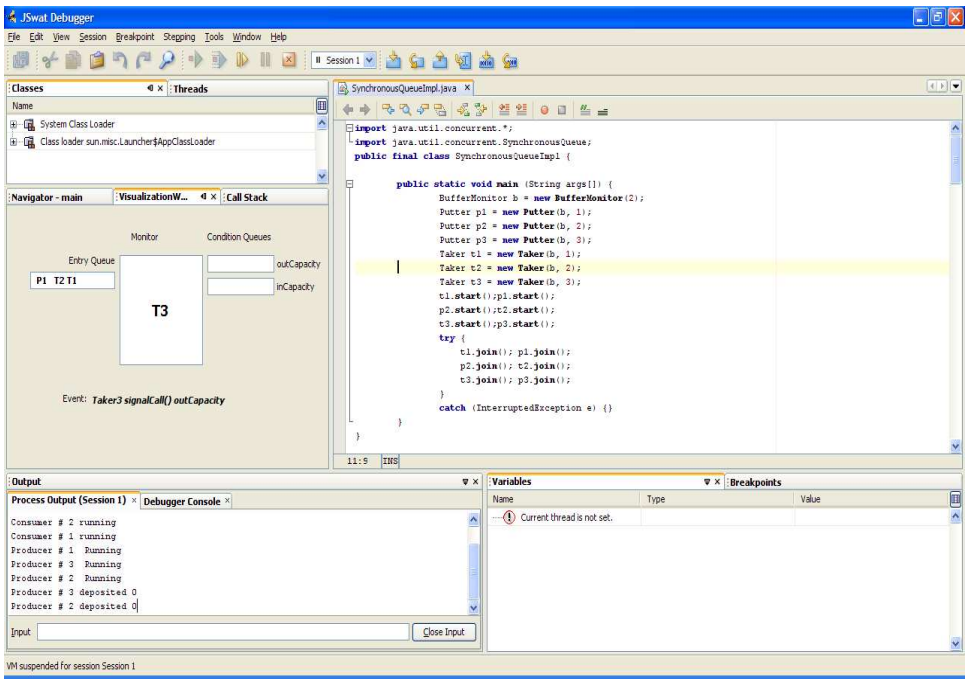

Figure 7.28 P1 awakened from the conditional queue.

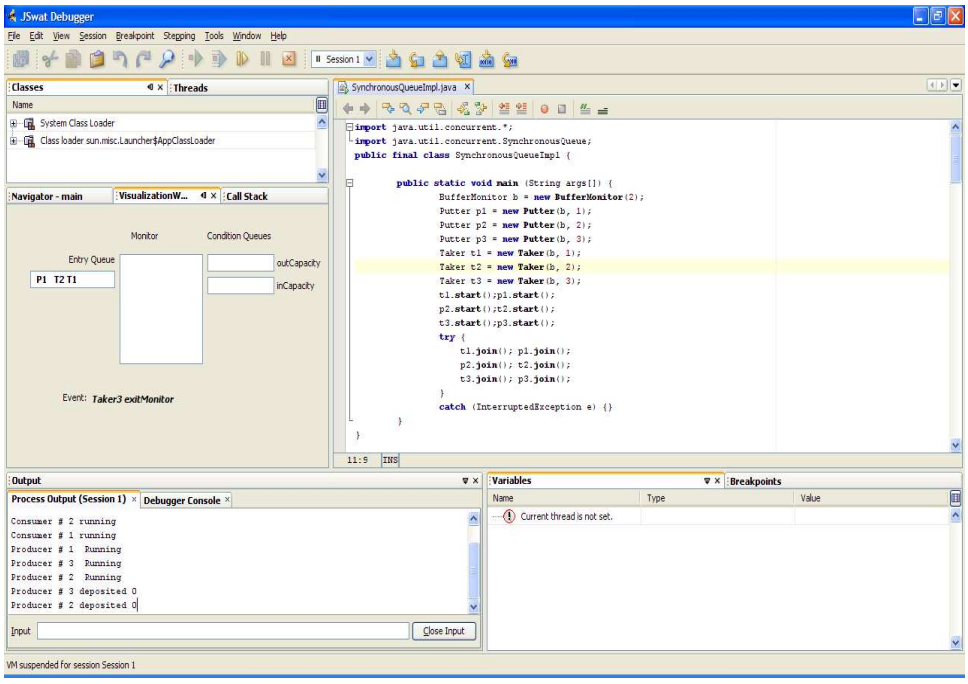

Figure 7.29 T3 exits the monitor.

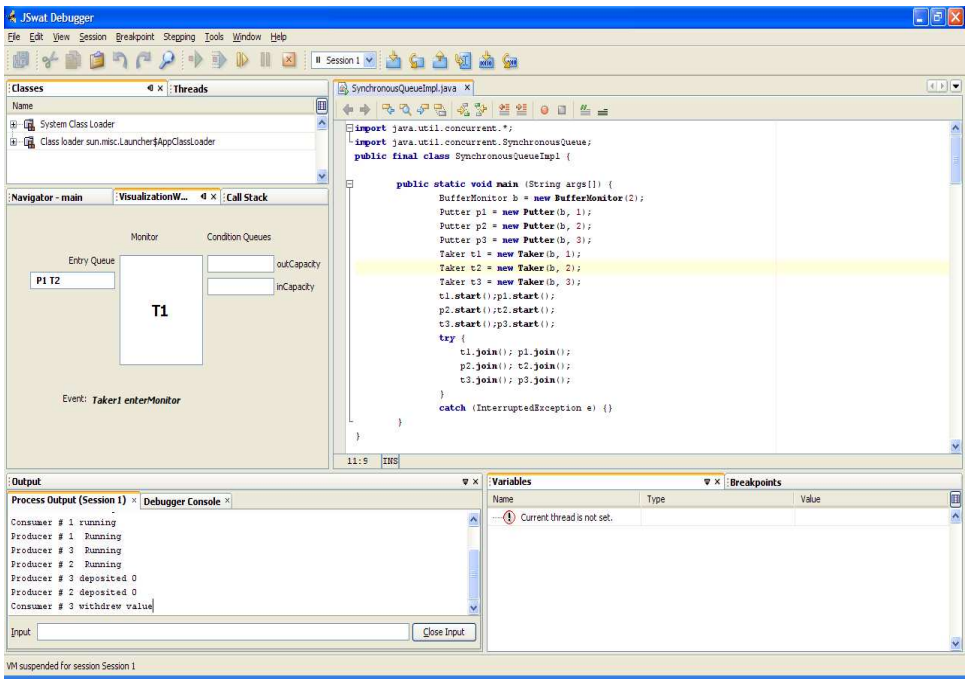

Figure 7.30 T1 enters the monitor.

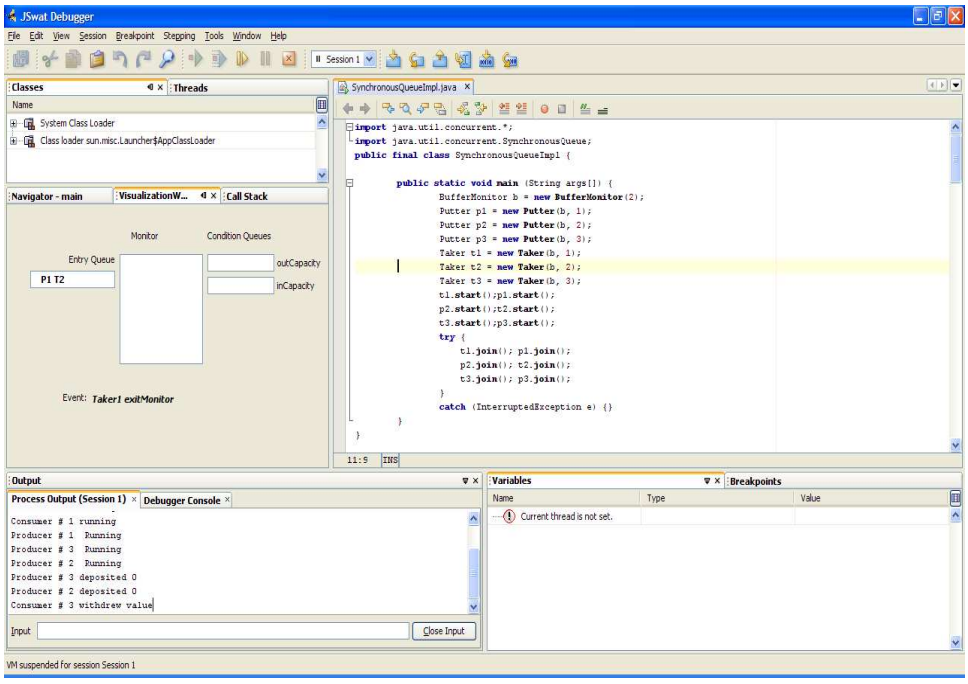

Figure 7.31 T1 exits the monitor.

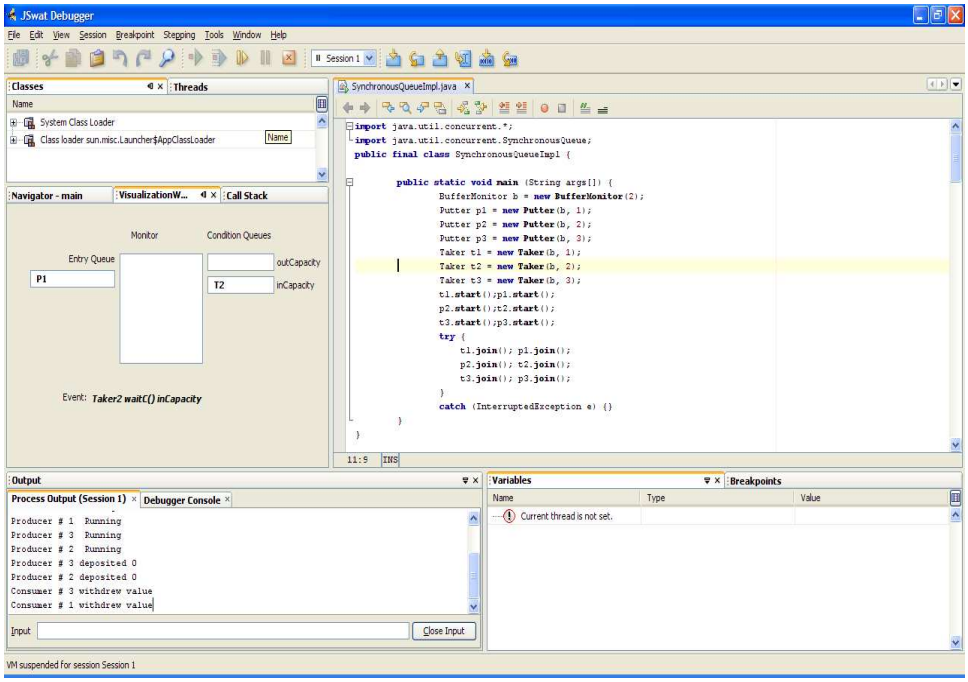

Figure 7.32 T2 waits in the conditional queue.

P1 then enter the monitor and deposits in Figure. 7.33.

After depositing P1 signals the conditional queue and exits in Figure. 7.34.

In Figure. 7.35, P1 exits the monitor.

T2 then enter the monitor and consumes in Figure. 7.36.

T2 signals the conditional queue and exits in Figure. 7.37. This example showcases the visualization process for a bounded buffer problem implemented using a synchronous queue.

7.3.3 Readers Writers Problem

This section deals with another classic multithreaded problem. It consists of two types of threads: Readers and Writers. The actual reading and writing in the shared section occurs outside the monitor. The monitor controls access to the shared section. Multiple readers can read simultaneously but only one writer can

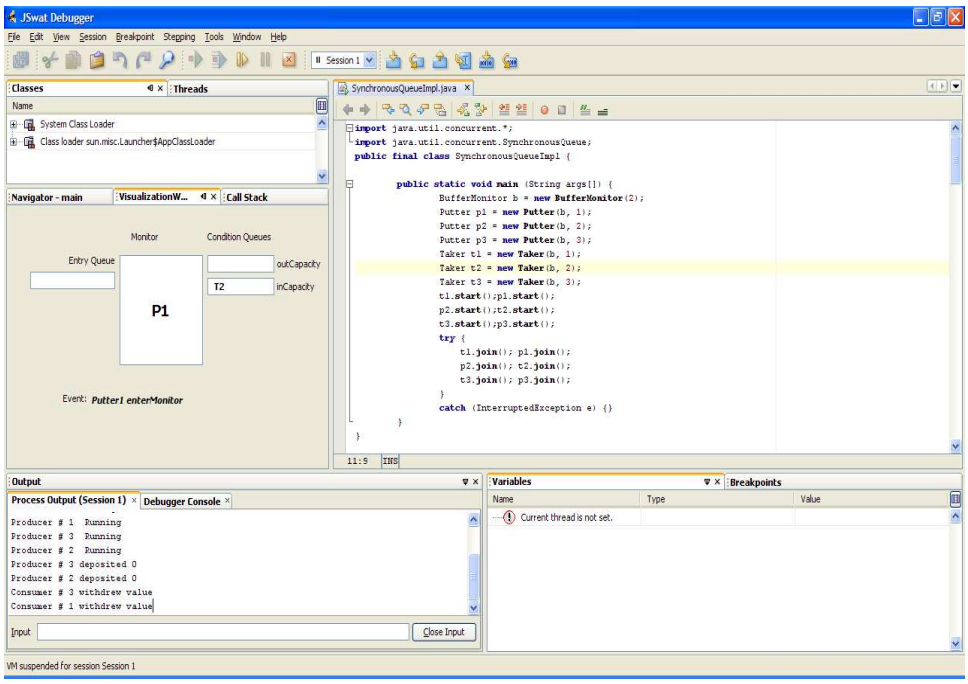

Figure 7.33 P1 enters the monitor.

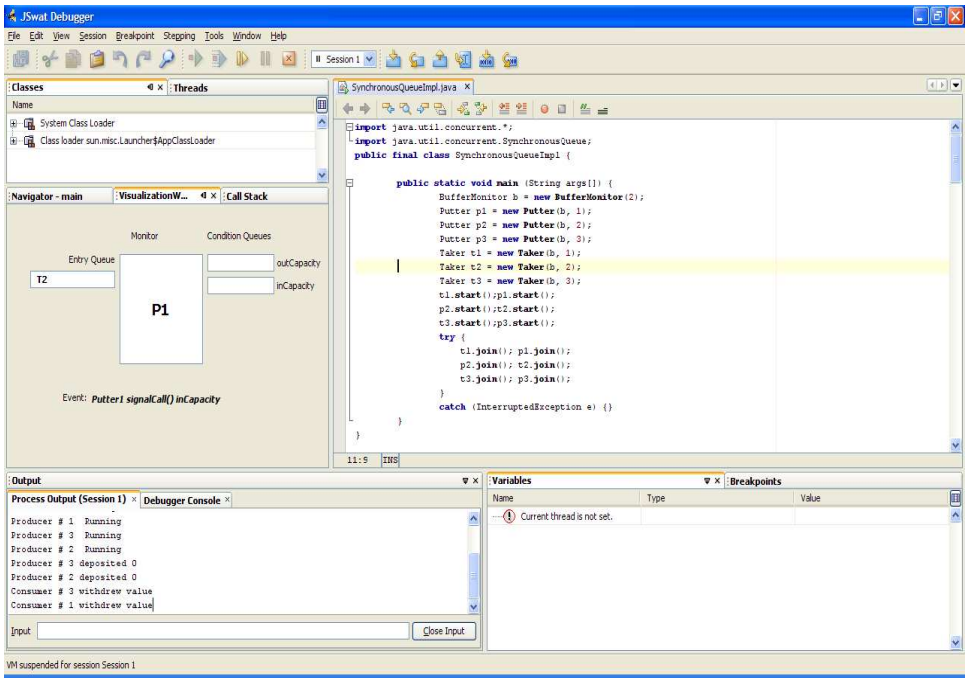

Figure 7.34 T2 awakened by P1 and enters the entry queue again.

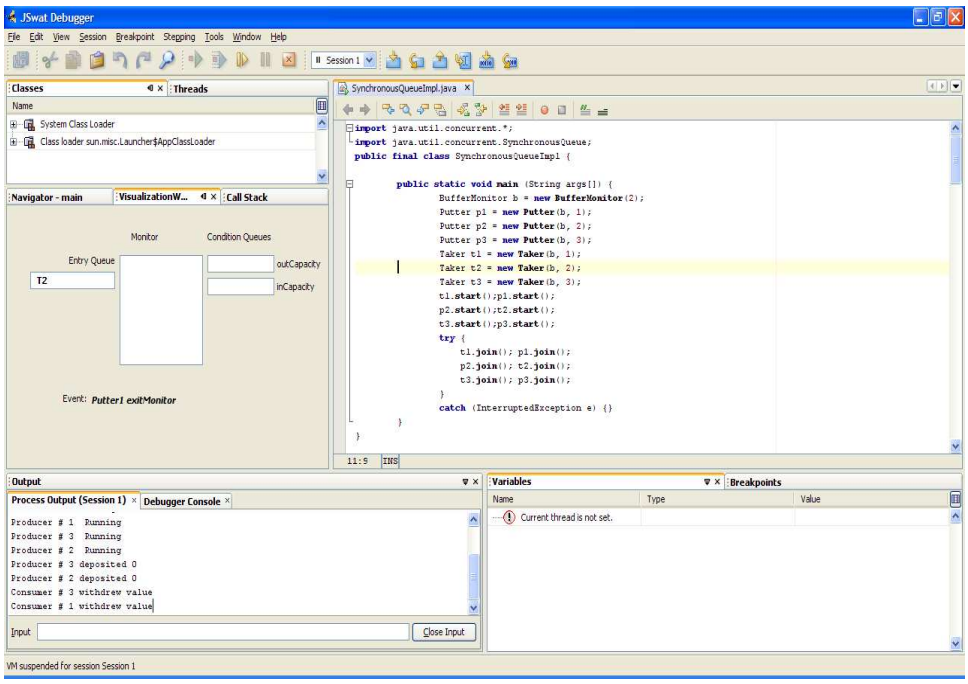

Figure 7.35 P1 exits the monitor.

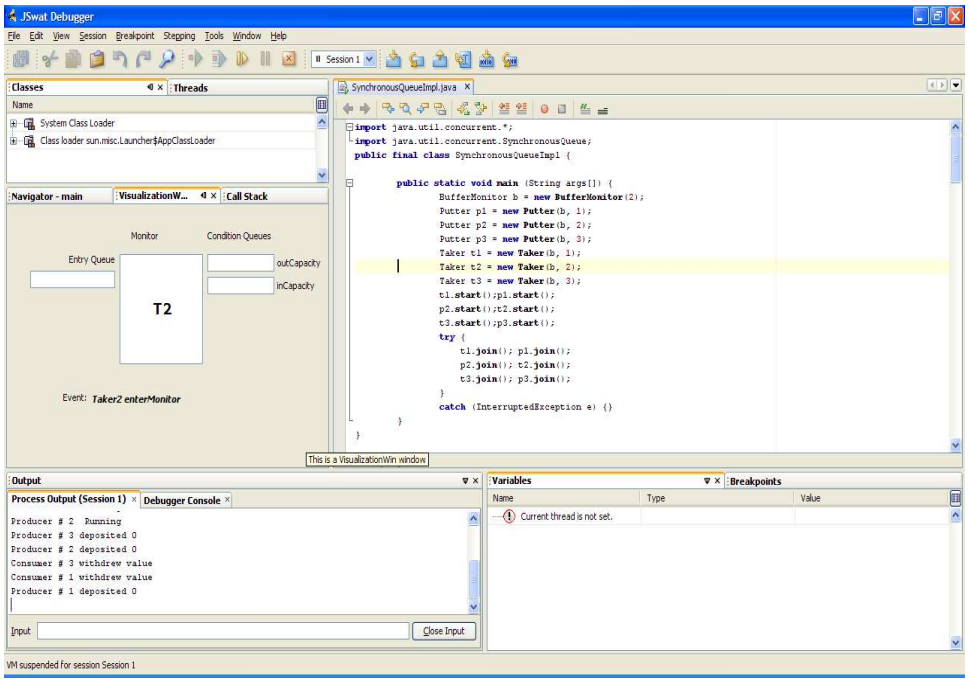

Figure 7.36 T2 enters the monitor.

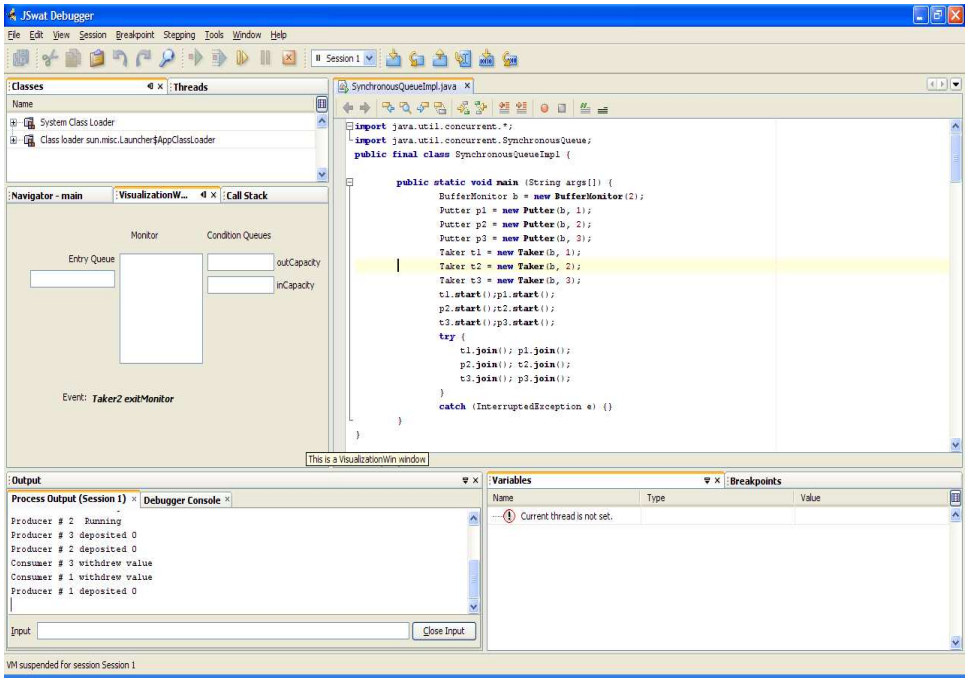

Figure 7.37 T2 exits the monitor.

write at any given time. Therefore, each thread enters and exits the monitor twice once before reading or writing and after reading or writing.

There are three readers and two writers. All the threads wait in the entry queue initially as in Figure. 7.38.

The first thread W1 enters the monitor before writing as in Figure. 7.39.

In Figure. 7.40, W1 then exits the monitor to write in the shared section.

R1 then attempts to enter the monitor but is made to wait in the reader since W1 is currently writing as in Figure. 7.41.

The next thread R2 also tries to enter the monitor and is made to wait in the readerQ as shown in Figure. 7.42.

R3 also waits in the readerQ as shown in Figure. 7.43.

W2 tries to enter the monitor from the entry queue and is made to wait in the writerQ as in Figure. 7.44.

In Figure. 7.45, W1 enters the monitor again after it completes writing.

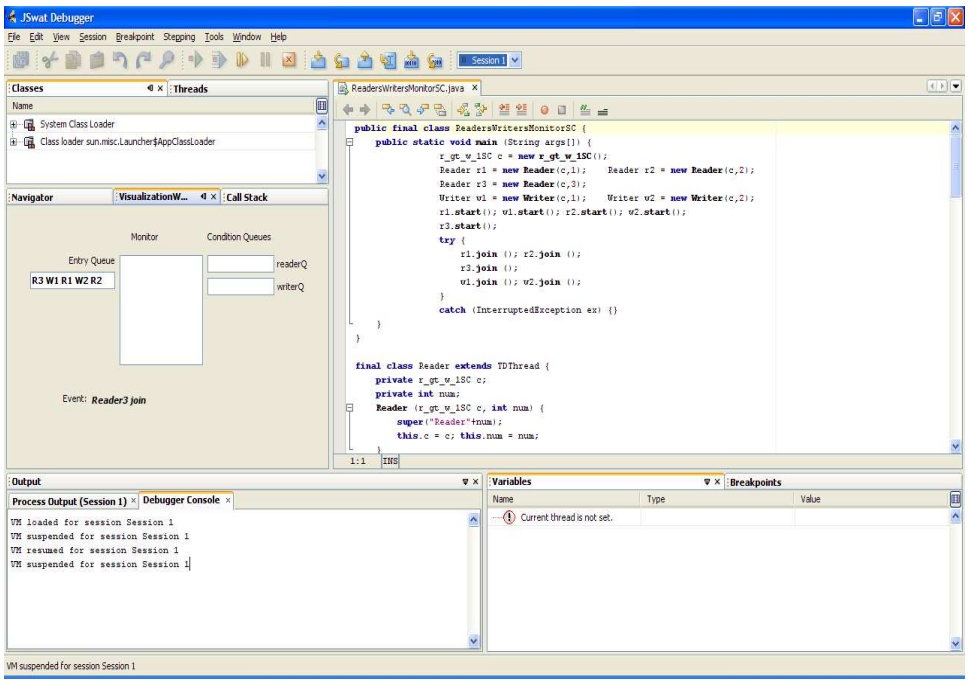

Figure 7.38 All readers and writers waiting in entry queue.

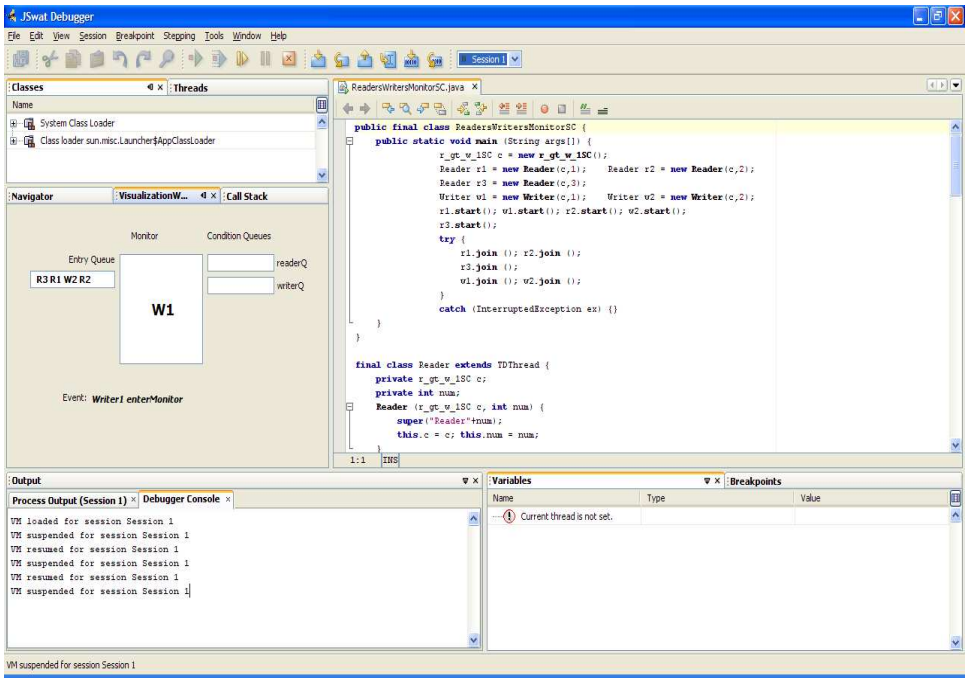

Figure 7.39 W1 enters the monitor before writing.

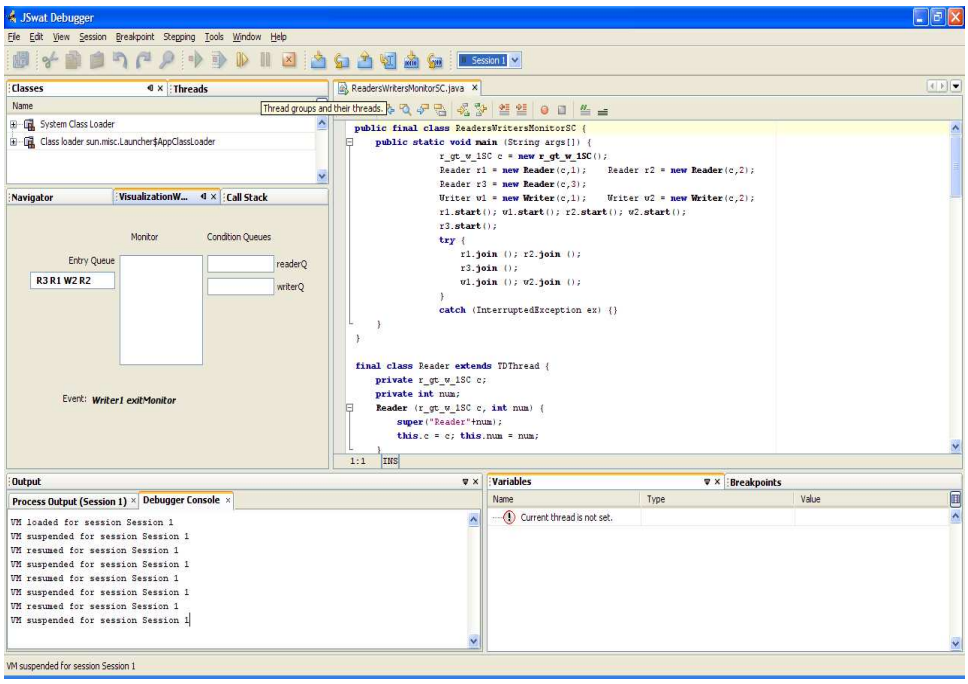

Figure 7.40 W1 exits the monitor.

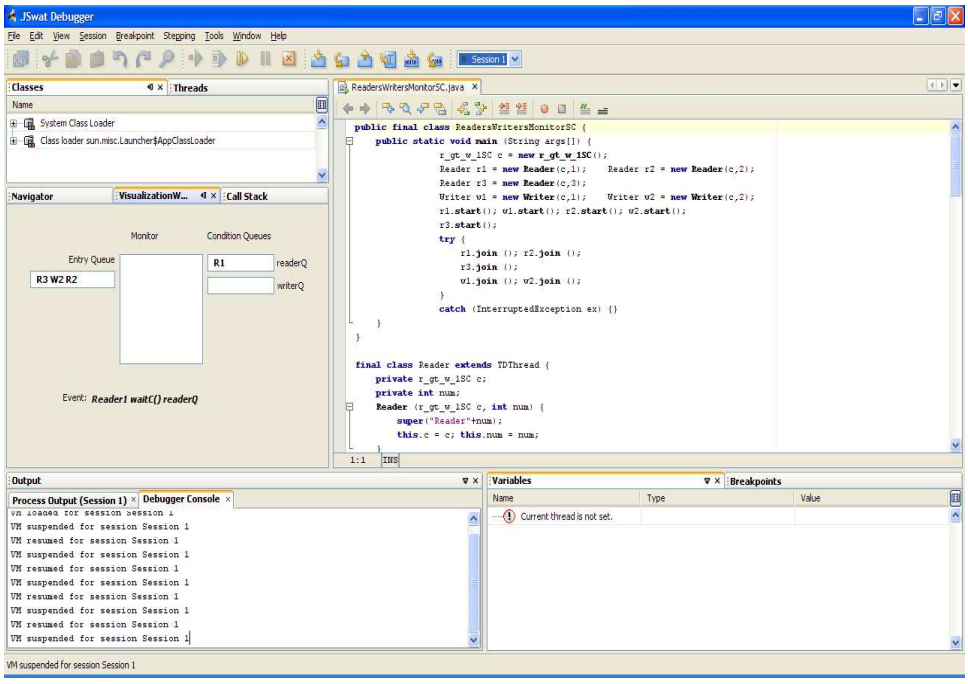

Figure 7.41 R1 waits in readerQ.

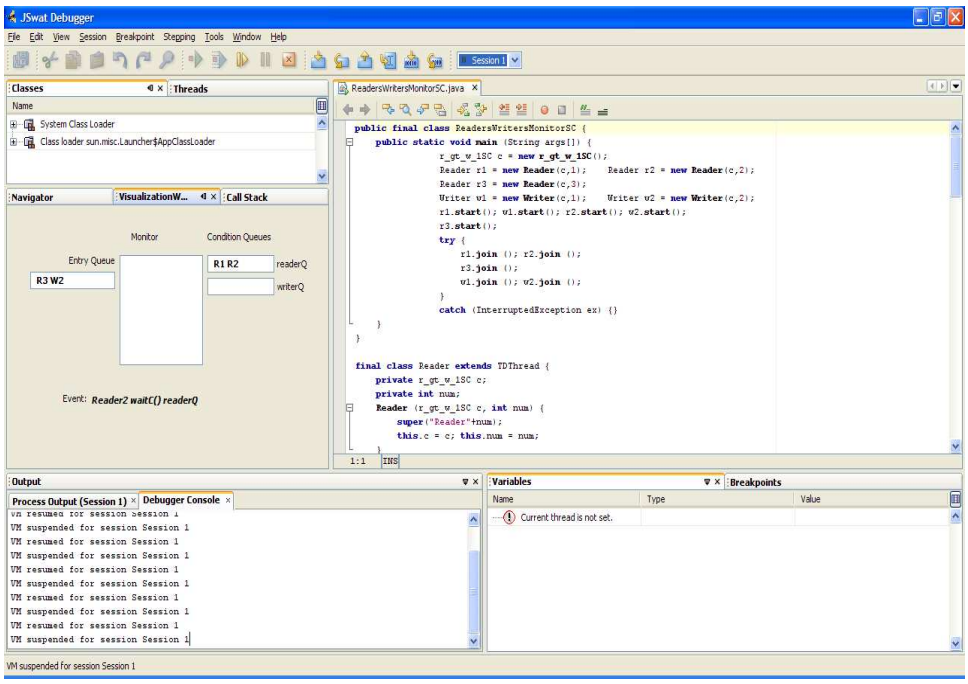

Figure 7.42 R2 waits in readerQ.

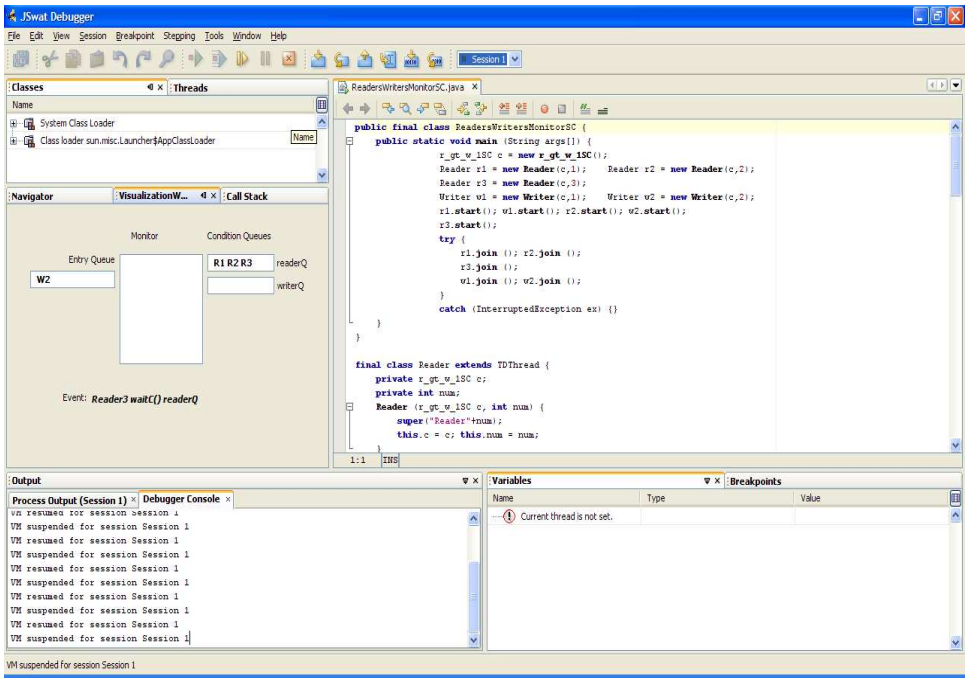

Figure 7.43 R3 waits in readerQ.

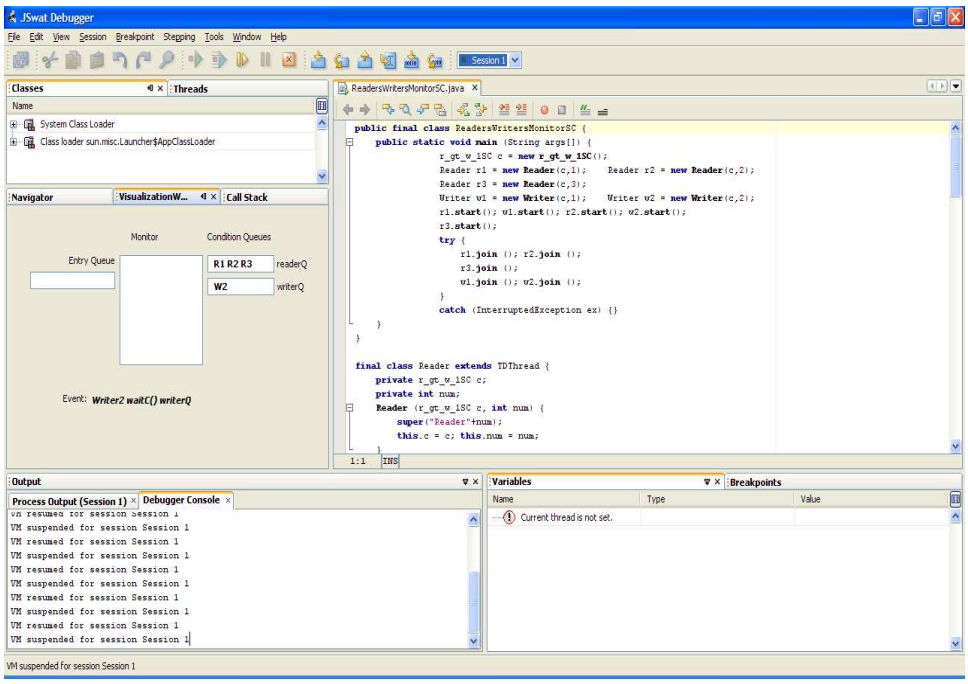

Figure 7.44 W2 waits in writerQ.

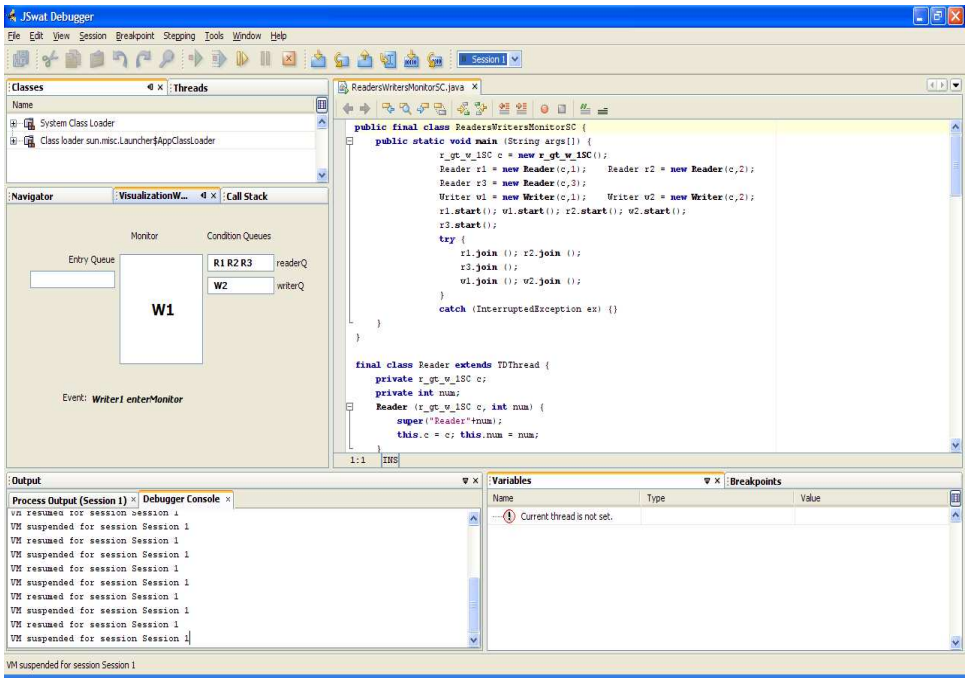

Figure 7.45 W1 enters the monitor after writing.

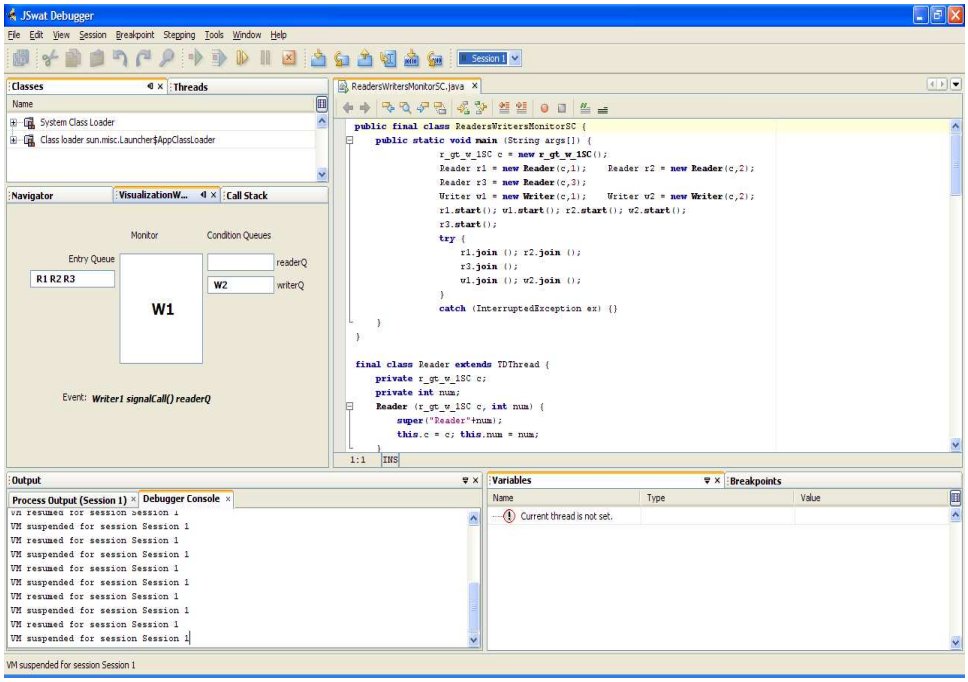

Figure 7.46 W1 awakens all the threads in readerQ.

As shown in Figure. 7.46, W1 awakens all the reader threads in readerQ. All the reader threads wait in the entry queue again.

W1 then exits the monitor as in Figure. 7.47.

R1 enters the monitor from the entry queue before reading, refer Figure. 7.48.

In Figure. 7.49, R1 exits the monitor to read in the shared section.

In Figure. 7.50, R2 enters the monitor before reading. Multiple readers are allowed to read in the shared section at the same time.

In Figure. 7.51, R2 exits the monitor to enter the shared section.

In Figure. 7.52, R3 enters the monitor for the first time before reading.

R3 then exits the monitor to enter the shared section in Figure. 7.53.

R1 then enter the monitor again after it completes reading and then proceeds to exit the monitor in Figure. 7.54.

In Figure. 7.55, R2 enters the monitor again after completing reading.

R2 then exits the monitor since it has completed reading in Figure. 7.56.

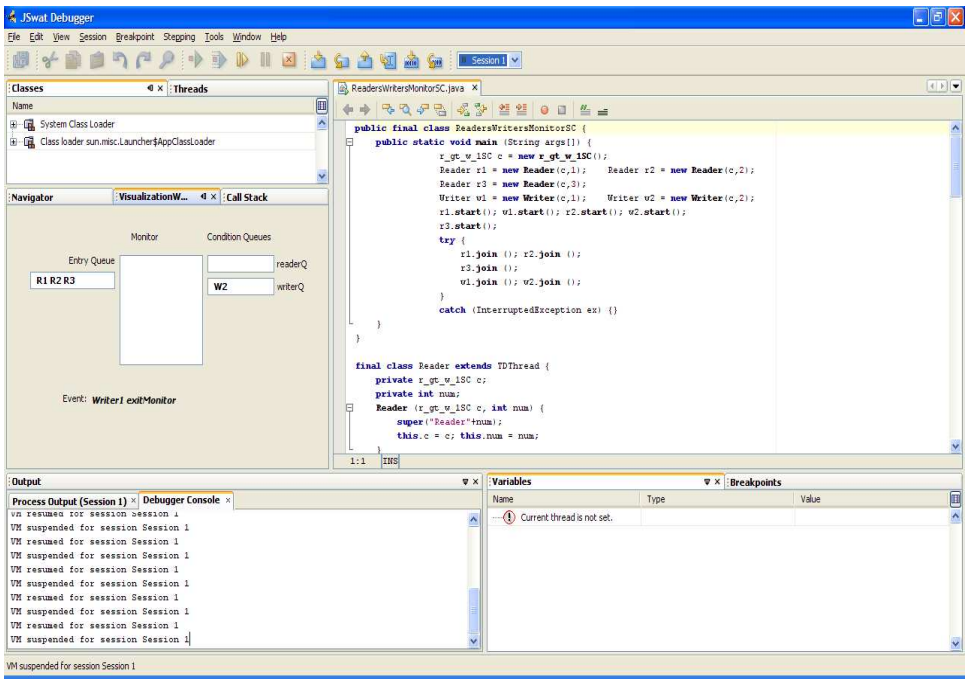

Figure 7.47 W1 exits the monitor.

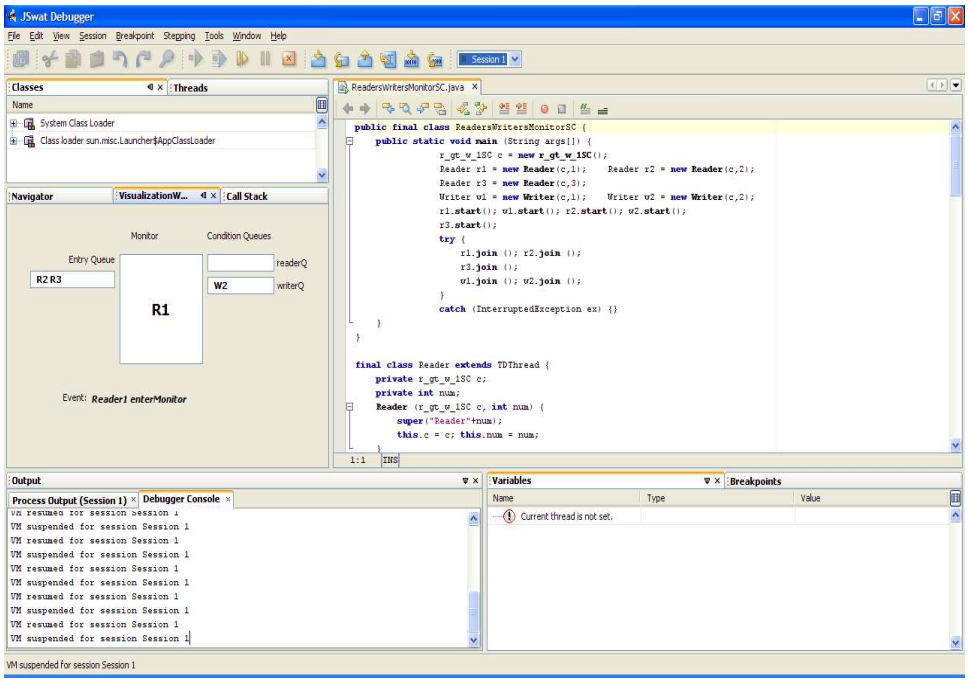

Figure 7.48 R1 enters the monitor before reading.

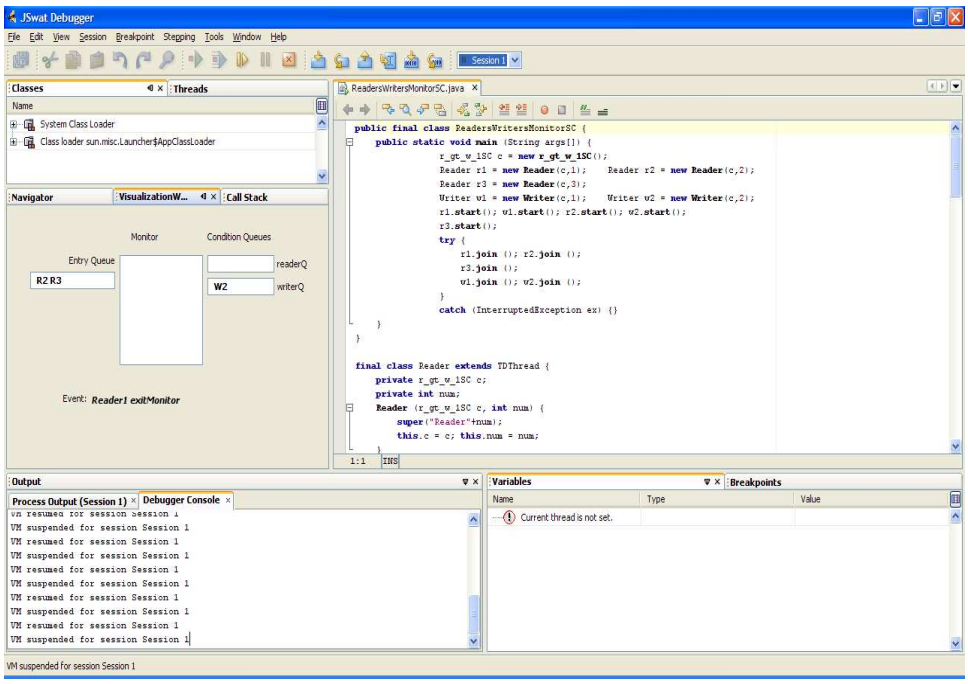

Figure 7.49 R1 exits the monitor.

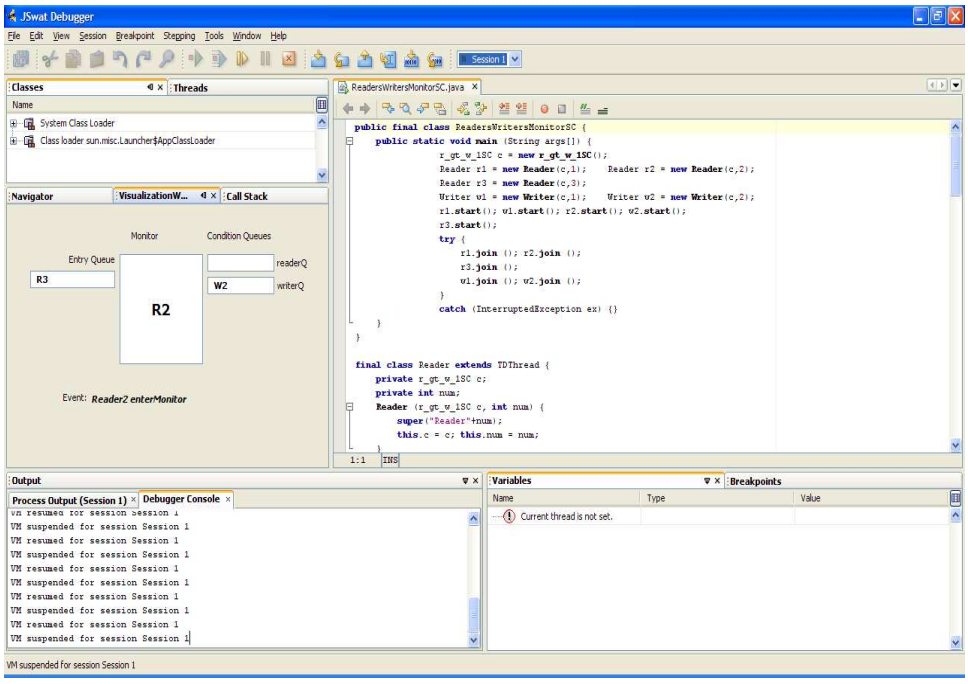

Figure 7.50 R2 enters the monitor before reading.

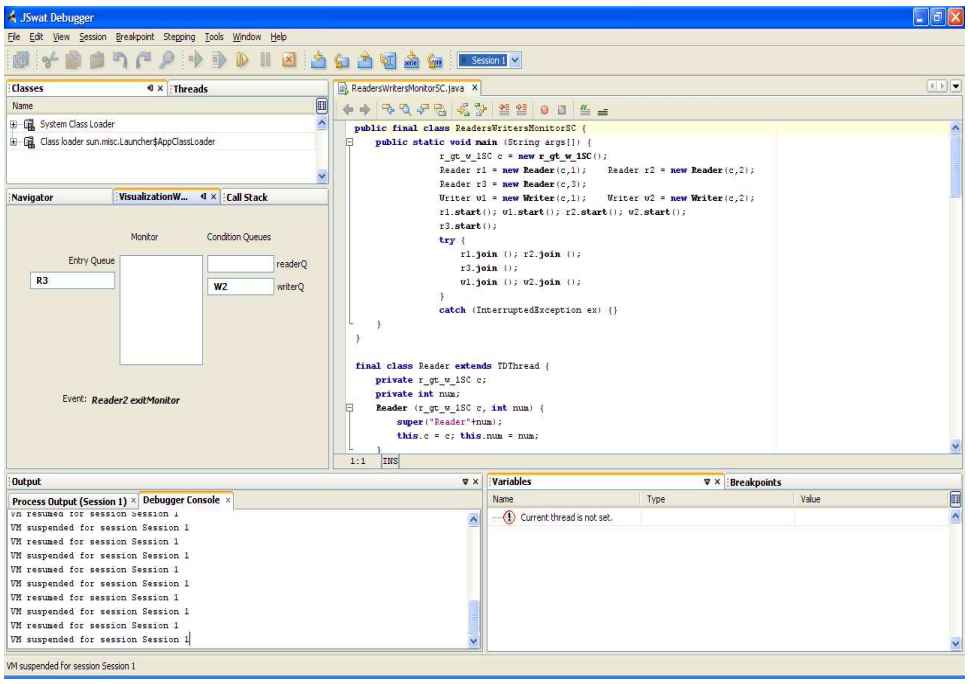

Figure 7.51 R2 exits the monitor.

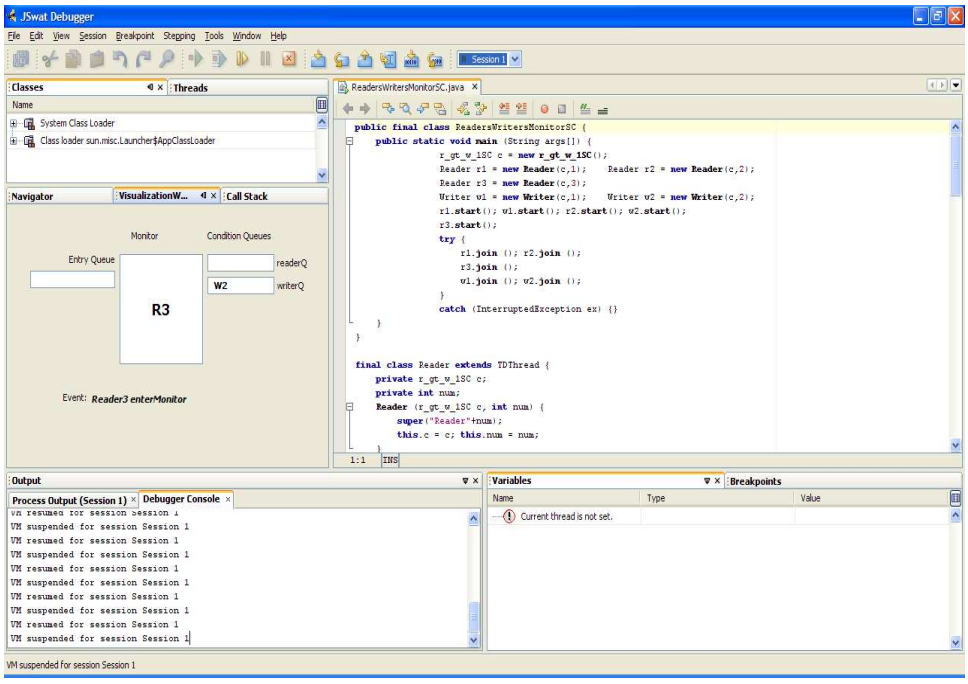

Figure 7.52 R3 enters the monitor before reading.

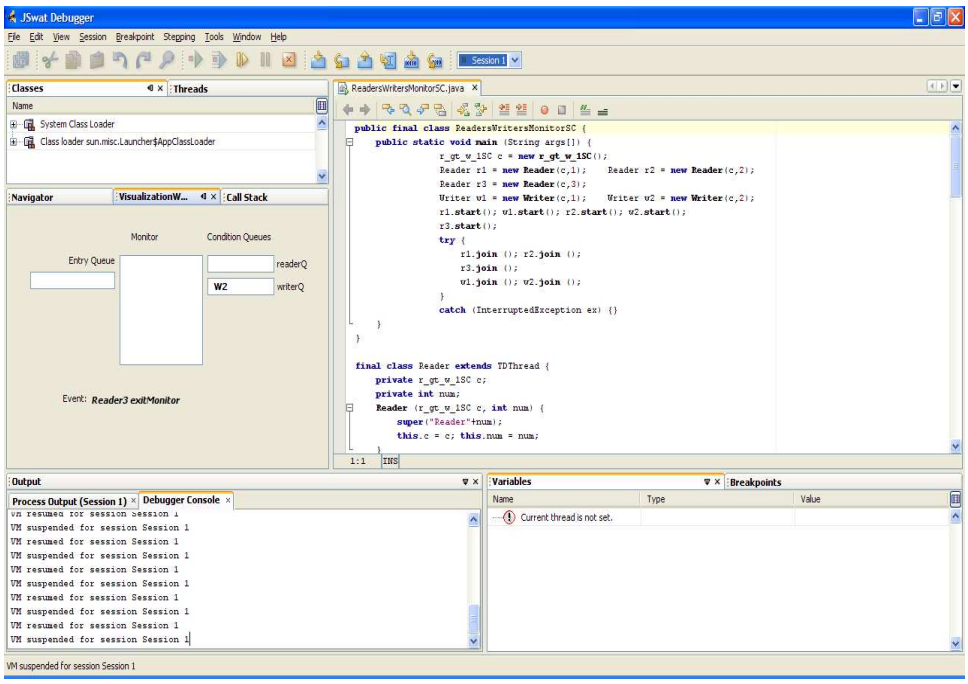

Figure 7.53 R3 exits the monitor.

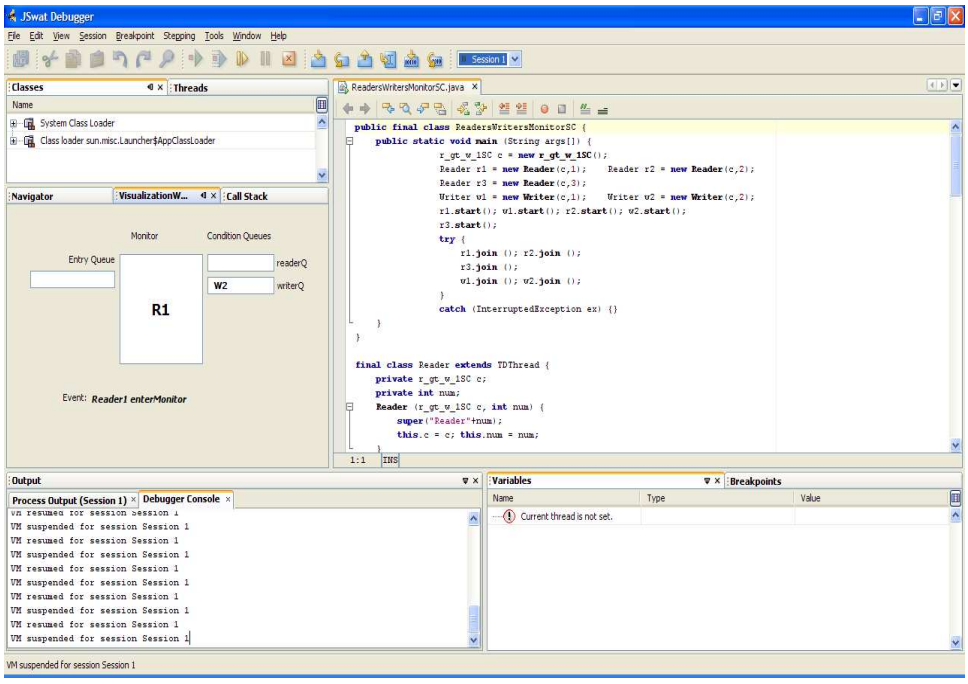

Figure 7.54 R1 enters the monitor after reading.

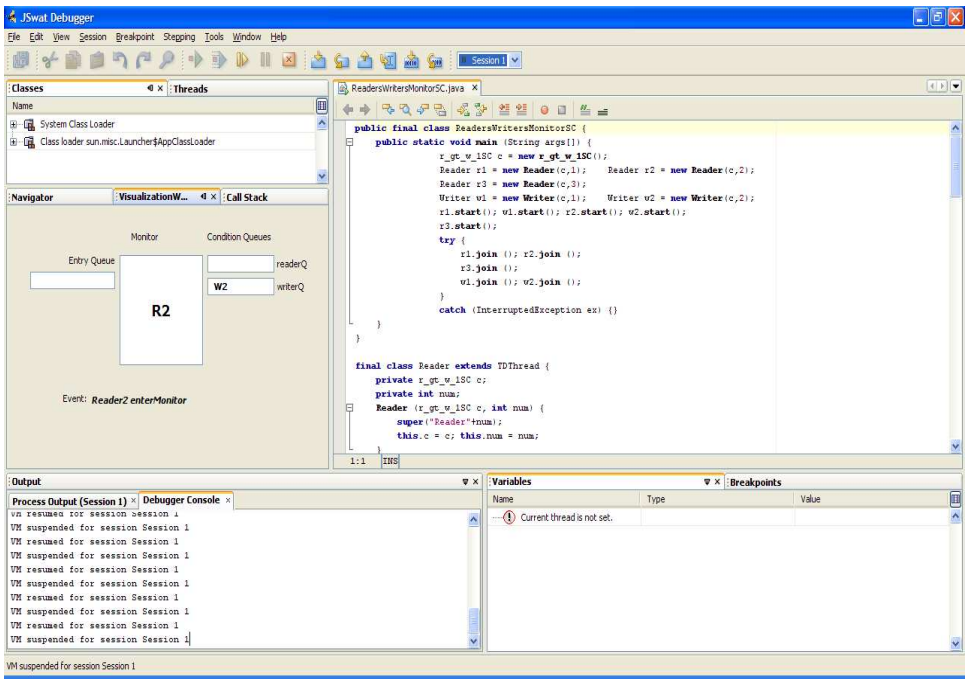

Figure 7.55 R2 enters the monitor after reading.

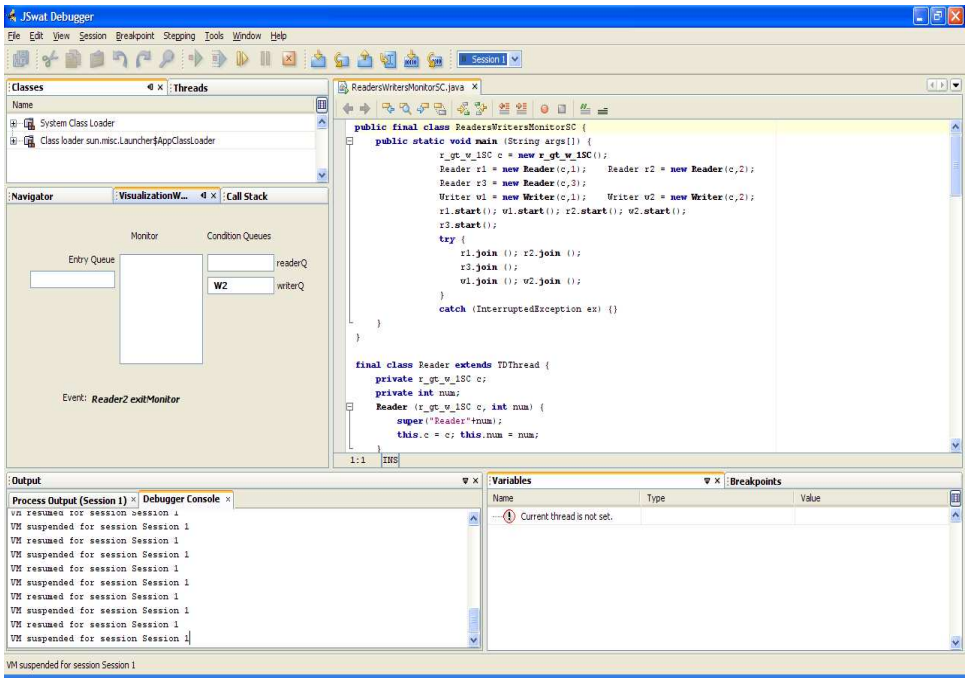

Figure 7.56 R2 exits the monitor after reading.

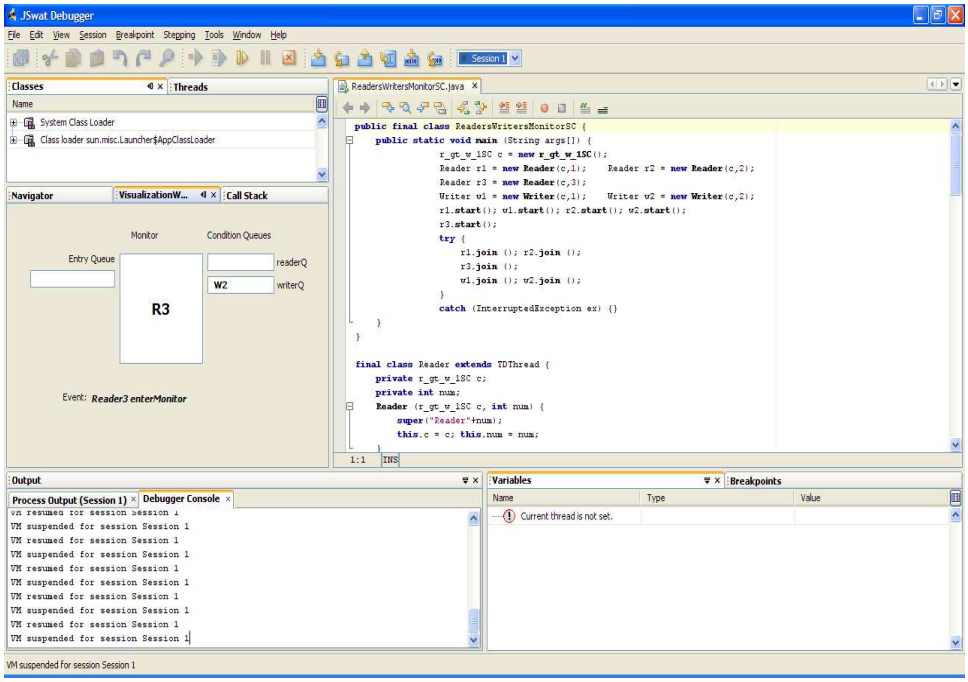

Figure 7.57 R3 enters the monitor after reading.

In Figure. 7.57, R3 enters the monitor again after it completes reading in the shared section.

Since R3 is the last reader to exit the shared section, it signals a writer from the writerQ as shown in Figure. 7.58. W2 then joins the entry queue.

R3 then exits the monitor in Figure. 7.59.

W2 enters the monitor from the entry queue as shown in Figure. 7.60.

In Figure. 7.61, W2 exits the monitor to write in the shared section.

In Figure. 7.62, W2 enters the monitor again after it completes writing.

In Figure. 7.63, W2 signals the readerQ since there are no writers waiting in the writerQ. This has no effect since readerQ is empty.

In Figure. 7.64, W2 exits the monitor since it has completed writing.

7.4 Execution Time Comparison

To determine the overhead of the visualization tool, we compare the execution time taken by the visualization tool with the original JSWAT tool. The concur-

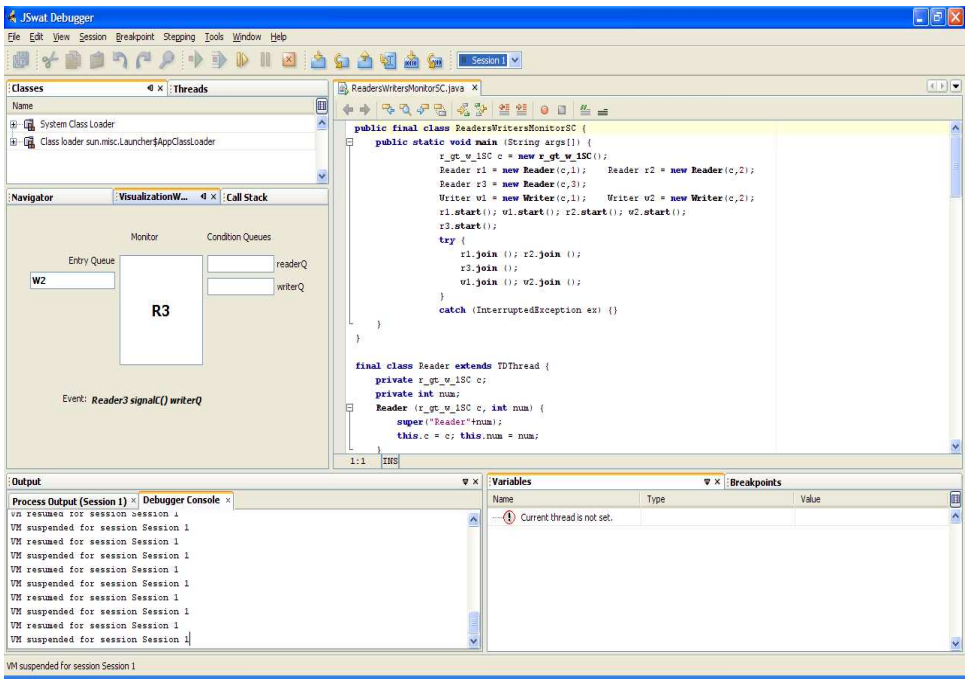

Figure 7.58 R3 awakens writer thread from writerQ.

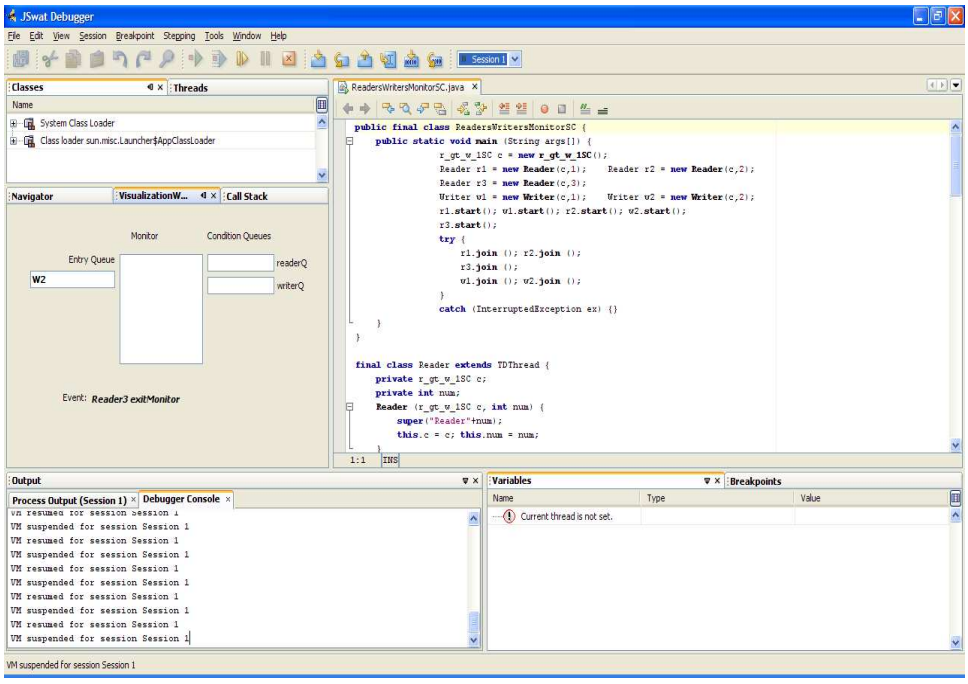

Figure 7.59 R3 exits the monitor after reading.

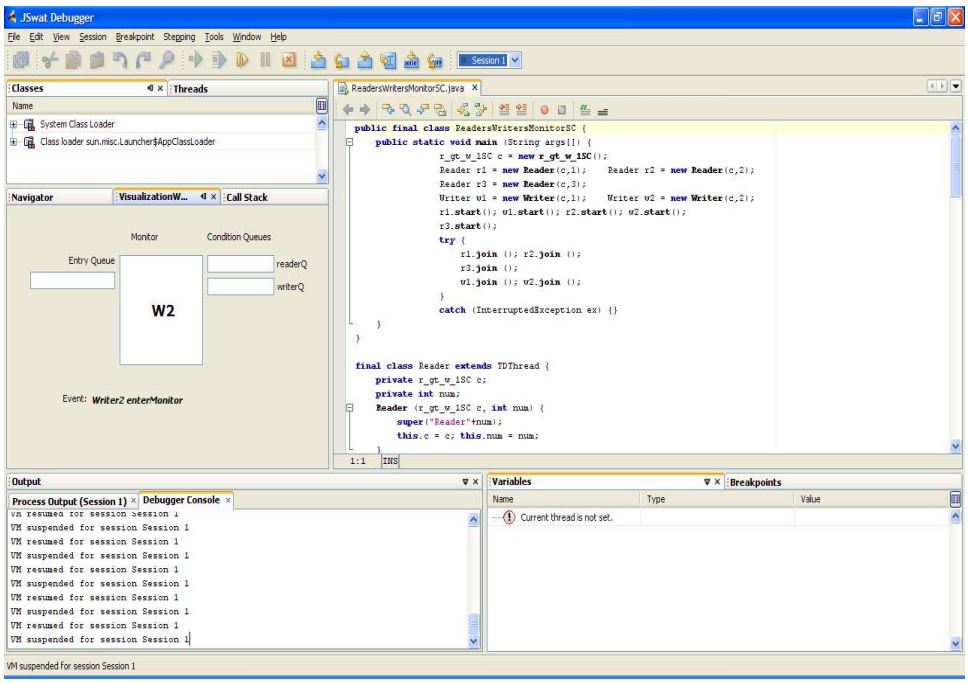

Figure 7.60 W2 enters the monitor before writing.

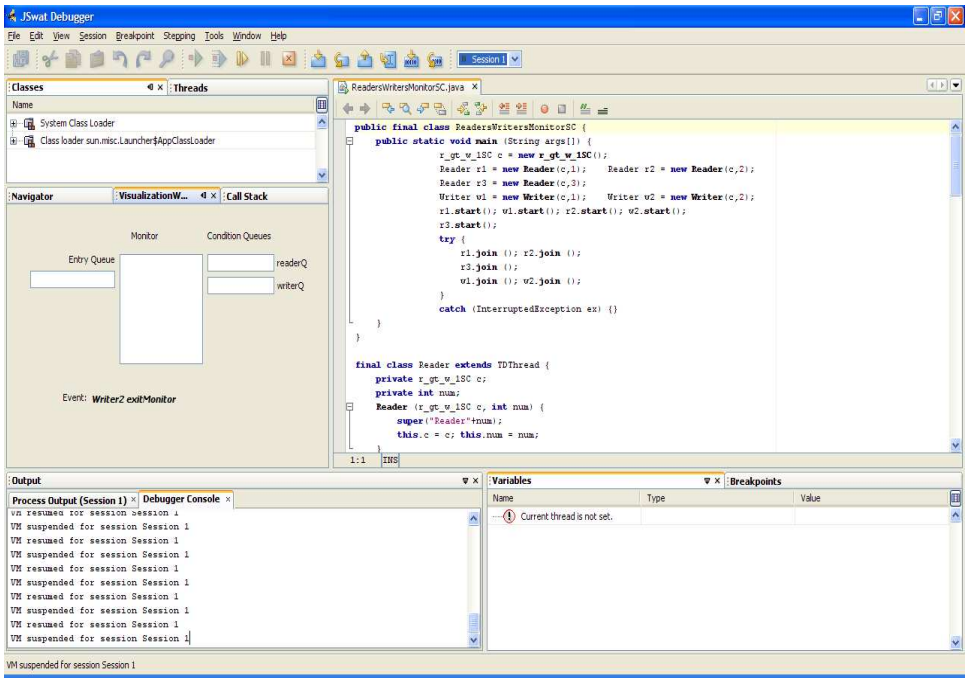

Figure 7.61 W2 exits the monitor before reading.

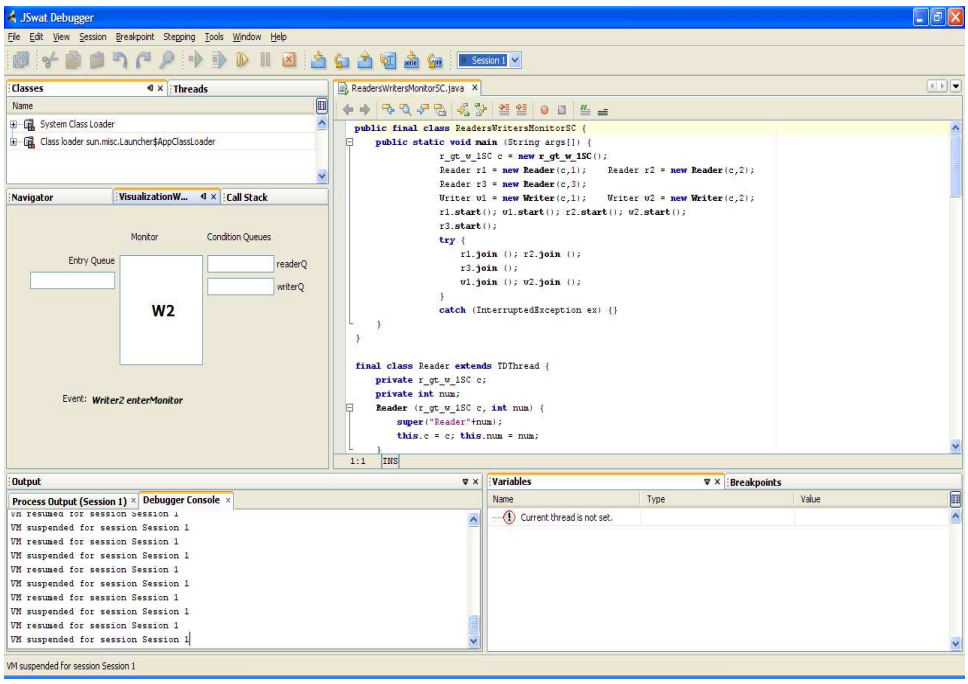

Figure 7.62 W2 enters the monitor after writing.

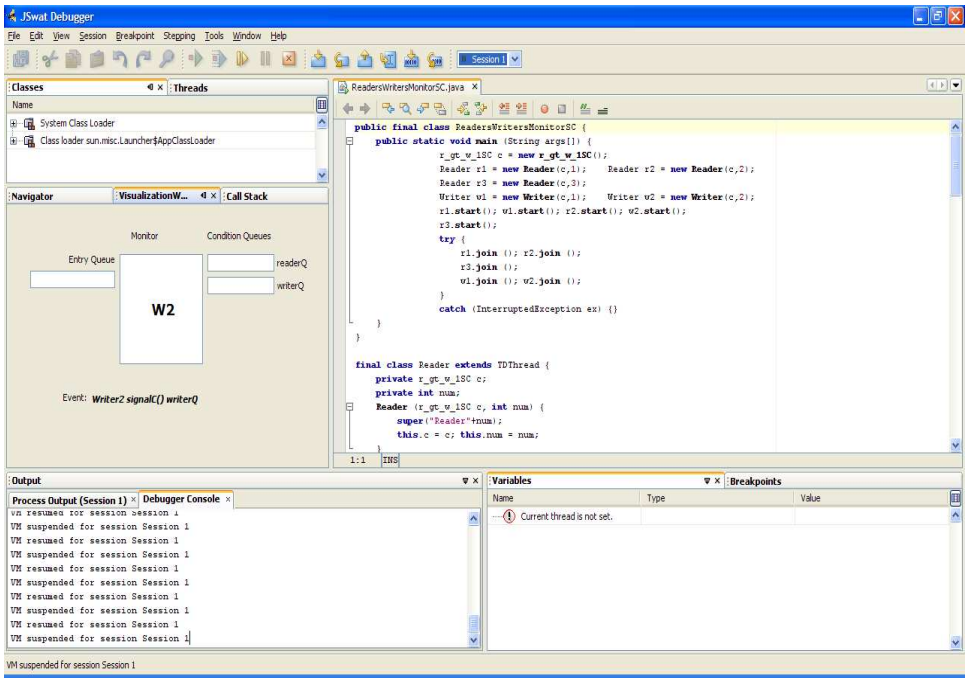

Figure 7.63 W2 signals the readerQ.

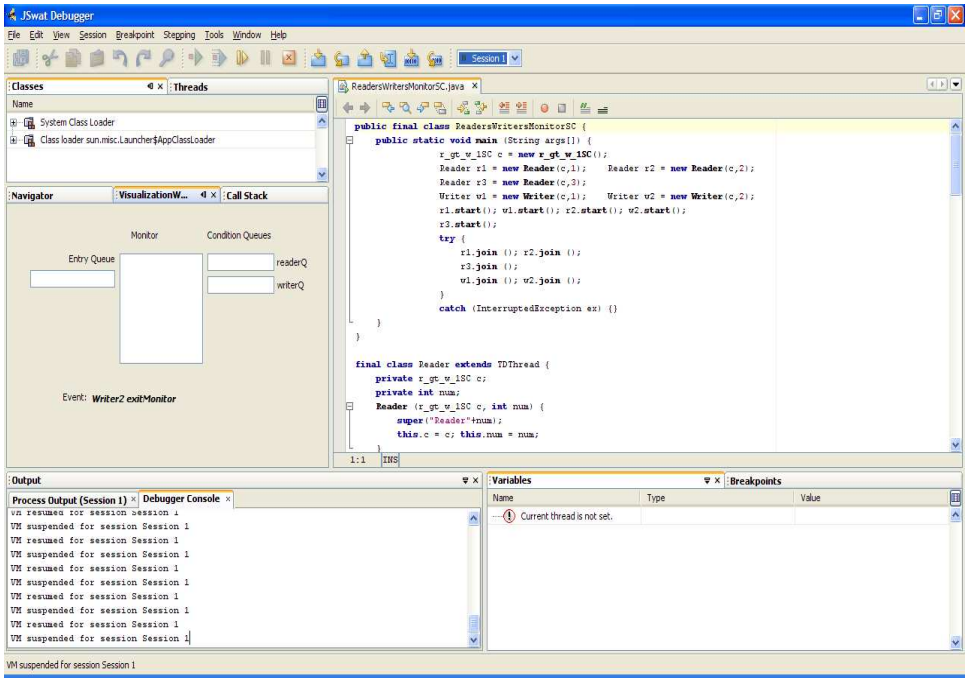

Figure 7.64 W2 exits the monitor after writing.

rent programs taken into consideration for this measurement are BoundedBufferMonitorSC.java, BoundedBufferMonitorSU.java, ReadersWritersMonitorSC.java, ReadersWritersMonitorSU.java and SynchronousQueueImpl.java. The execution time was measured by recording the system time before and after the main execution and calculating the difference. Ten such readings were taken for each program and the average is calculated. This procedure is implemented for both original JSWAT and the visualization tool. The readings are listed in Figure. 7.65. As we can see, the visualization tool, on average, increases the execution time by order of tens of milliseconds.

	Original JSWAT	JSWAT integrated with Visualization Tool
BoundedBufferMonitorSC.java	92.1	148
BoundedBufferMonitorSU.java	73.2	128.5
ReadersWritersMonitorSC.java	98.4	173.5
ReadersWritersMonitorSU.java	98.4	167.8
SynchronousQueueImpl.java	84.3	131.1

Figure 7.65 Comparision of average execution time in milliseconds.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

The goal of this thesis work is to build a testing and debugging environment specially suited for concurrent applications. In this thesis work, we discussed multi-threading concepts and the complexities involved in developing, testing and debugging multi-threaded programs. We also discussed the contributions of other researchers in the field of study related to this thesis work.

We described the concept of monitors in detail, including types of monitors, their structure and their working. We further detailed this topic by exploring the solution for bounded buffer problem using Signal-and-Urgent-Wait monitor. Next, we presented the graphical Java debugger JSWAT. It offers advanced debugging features which would facilitate a programmer to test and debug programs with ease. We also discussed the problems the visualization tool aims to alleviate. The tool identifies important events that have to be visualized in the concurrent application using regular expressions and then visualizes those events. Visualization rules were developed for this purpose, which operates according to the working of monitor constructs and types. We also described the working of parser and visualizer. Furthermore, the visualization tool and JSWAT are synchronized to present an integrated testing and debugging environment for programmers.

My contribution in this thesis work is to design and implement the visualization panel. I implemented the communication interface between the user application and the visualization tool. I also collaborated in the visualization rules that decide what visualization steps are executed based on the monitor type and messages.

8.2 Future Work

The visualization tool can be extended to visualize other multi-threading constructs such as locks and semaphores and constructs from other programming languages. For large and complex applications, the different modules can be extended to a distributed network to achieve higher efficiency. If this tool can be effectively integrated with a Java profiling application, it can be researched upon.

REFERENCES

[1] R. H. Carver and K.C. Tai, *Modern Multithreading Implementing, Testing, and Debugging Multithreaded Java and C++ Pthreads Win32 Programs.* New Jersey, USA: Wiley, 2006.

[2] JSWAT Debugger. [Online]. Available: http://jswat.sourceforge.net/

[3] JSWAT User Guide. [Online]. Available: http://jswat.sourceforge.net/

[4] JSWAT User Guide. [Online]. Available: http://java.sun.com/j2se/1.3/docs/guide/jpda/architecture.html

[5] Synchronous Queues in Java. [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ SynchronousQueue .html

[6] V. Gupta, "Monitorexplorer: A state-space exploration based tool to test Java monitors implementations," Master's thesis, University of Texas at Arlington, Arlington, 2006.

[7] S.P. Reiss, "Visualizing Java in Action," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, San Diego, California, pp. 57–ff.

[8] C. Artho, K. Havelund, S. Honiden, "Visualization of Concurrent Program Executions, " in *COMPSAC 2007: 31st Annual International.* 24-27 July 2007, pp. 541–546.

[9] Netbeans Platform. [Online]. Available: http://www.netbeans.org/

BIOGRAPHICAL STATEMENT

Keerthika Koteeswaran was born in Chennai, India, in 1983. She obtained her B.Tech in Information Technology from Anna University in 2005. Her interest in advanced education in Computer Science brought her to the University of Texas at Arlington, where she obtained her M.S degree in Computer Science and Engineering in 2008. She worked as a Software Engineer Intern with Cummins from June to August of 2007. She is currently working as a QA Engineer with eBay, Inc.