ENHANCING JSWAT FOR MONITOR-BASED EXECUTIONS

by

ARUN RAMANI

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 offer my sincere thanks to Dr. Jeff Yu Lei for giving me an opportunity to work on this project. It has been a memorable experience, to work under his supervision. I would also like to thank Dr. Leonidas Fegaras and Mr. David Levine for agreeing to serve on my committee.

I would like to thank Keerthika Koteeswaran for helping me out with the thesis. I would also like to thank Mr. Nathan Fielder, lead developer of JSWAT for his time and wonderful support for all the questions I had regarding JSWAT. I would like to acknowledge all my friends who encouraged and motivated me throughout my work.

Last but not least, I would like to mention my family, especially my dad, mom and sister, Mr.S. Ramani and Mrs. Radha Ramani and my sister Arthy Ramani. They have always been the source of inspiration and strength for me especially when it is needed the most. I can not thank them enough for giving me such a brilliant opportunity.

March 4, 2008

ABSTRACT

ENHANCING JSWAT FOR MONITOR-BASED EXECUTIONS

ARUN RAMANI, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Jeff Lei

Concurrent programs contain more than one thread that execute concurrently to accomplish a particular task. Since the threads in a concurrent program work together to accomplish a common goal, they share the data, code, resources and address space of their process. This reduces the overhead involved in creating and managing the threads but leads to side-effects like race conditions, critical section problem and deadlocks. There are several operating system constructs to solve these issues such as locks, semaphores, monitors etc. Locks are associated with each object to control access of shared resources. Semaphores can be described as counters used to control access to shared resources. A monitor by definition encapsulates shared data, all the operations on the data and any synchronization required for accessing the data.

JSwat is a stand-alone graphical Java debugger front-end that uses the Java Platform Debugger Architecture. It has features including 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, and Java-like expression evaluation, including method invocation. These advanced features make JSWAT an ideal debugger for a Java concurrent programmer. The main disadvantage of concurrent programs is that they are extremely difficult to test and debug. This is because multiple executions of a concurrent program with the same input may produce different results. This nondeterministic execution behavior creates several problems during the testing and debugging cycle of a concurrent program.

To alleviate this problem to a certain extent this thesis proposes integrating a visualization tool with JSWAT for viewing the status of the monitor and the different threads in a concurrent program during runtime. The visualization will be viewed in a panel in the graphical user interface of JSWAT. When concurrent programmers view the status of the threads with respect to the monitor at runtime, they can understand the working of the program better. Hence, they can easily identify, analyze and rectify bugs in the program.

The tool parses the program to identify data pertinent to visualization such as number of condition variables involved in a concurrent program and their names, sections where threads interact with the monitor and the condition queues. These sections are then mapped to their corresponding visual interpretations or rules. The visualization rules decide how each action of the thread is depicted in the visualization panel. This depiction also depends on the signaling discipline of the monitor. The signaling disciplines are Signal-and-Continue, Signal-and-Urgent-wait and Signal-and-exit. Based on the method's executed by the thread and signaling discipline, the visualization panel decides if a particular thread has to be placed at the entry queue, reentry queue, a condition queue or the monitor. Therefore, the programmer can view the interaction among the threads and the monitor in a particular execution and identify any bugs in the program easily. The programmer can also set breakpoints in JSWAT and view the current status of the threads at that point in the visualization panel.

This user-friendly visualization tool along with the feature-rich JSWAT debugger aims to reduce the time and effort spent by Java programmers in testing and debugging concurrent programs and hence increase their productivity.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Software testing and debugging are an integral part of the software development life cycle. It is a well-known fact that in a project, more than half of time and money is spent on testing and debugging. Even in a moderately sized software development project involving sequential programming, programmers require debuggers with features such as line and method breakpoints, data tool tips, stop on specific events etc.

Concurrent programs are much more complex to develop, test and debug when compared to sequential programs. This is mainly due to the non-deterministic execution behavior of concurrent programs. This nondeterminism is caused by unpredictable rate of progress and sequence of execution of different threads. The conventional testing and debugging practice is to repeatedly execute the program to identify and fix programming errors. But in the case of concurrent programming, repeated execution does not guarantee the same output for the same input. This is because the sequence in which the threads interleave during execution may not be the same.

Also, once an error is encountered, it is very difficult to reproduce the same error again. This necessitates a debugger with special features catering to concurrent program developers. An important feature of such a debugger would be to display graphically the status of threads during runtime.

Since concurrent programs are multithreaded in nature, it is necessary to use software constructs to handle thread synchronization, communication, progress and mutual exclusion. Some of the constructs are semaphores, locks and monitors. A monitor encapsulates shared data, all the operations on the data, and any synchronization required to access the data. A monitor has separate constructs for mutual exclusion and condition synchronization. In fact, mutual exclusion is provided automatically by the monitors implementation, freeing the programmer from the burden of implementing critical sections. Therefore, monitors are popular among concurrent program developers. This thesis focuses on a visualization tool to help developers in viewing how the different threads in their concurrent program interact with the monitor and with each other. This would help the programmers to understand the working of their program and to identify errors. At run time, developers can view the threads present inside the monitor, entry queue, reentry queue and condition queues. This visualization tool is integrated with a graphical user interface Java debugger called as JSWAT [6]. So users can take advantage of the various advanced features of the debugger apart from viewing the threads. Using JSWAT, developers can set colorized breakpoints, watch specific variables and methods, identify syntax errors, stop and start execution using appropriate buttons and view threads, call stacks and visible variables. JSWAT uses Java Platform Debugger Architecture [2] and is based on the Netbeans Platform.

1.2 Structure of the thesis

The thesis is structured as follows: Chapter 1 gives a brief overview of the thesis work. Chapter 2 explains monitors and signaling disciplines. Chapter 3 deals with JSWAT. Chapter 4 explains the Event Recognizer in detail. Chapter 5 deals with the Visualizer and the Visualization Window. Chapter 6 explains about the integrator that integrates the event recognizer and the visualizer. Chapter 7 deal with the sequence of execution of the tool. Chapter 8 deals with the background and related work for concurrent program testing and debugging. Lastly, Chapter 8 concludes with the goals achieved by this work.

My contribution in this thesis work is to identify the conditional queues from the user application. I also implemented a module to identity the events in the user application that changes the state of the thread. These events are interpreted by the visualization rules to visualize the threads. I helped in the design and implementation of the visualization rules. Another important component that I implemented is the synchronous queue [8]. This synchronous queue helps in temporary storage of the events identified from the user program as well as in the synchronization of the threads.

CHAPTER 2

MONITORS AND SIGNALING DISCIPLINES

2.1 Introduction to Monitors

In this chapter, the usage of monitors and signaling disciplines in concurrent applications is explained. Concurrent applications [7] has multiple threads executing in parallel to accomplish a common goal. Since these threads share the CPU and resources to achieve its goal, it becomes necesary to synchronize them.

Monitors act as a layer of abstraction, that encapsulates shared data, all the operations on the data, and any synchronization required for accessing the data. Monitors by default guarantee mutual exclusion, reducing the burden on the programmers to implement critical sections and mutual exclusion.

2.2 Concurrent applications and use of monitors

A monitor is used for mutual exclusion and condition synchronization. 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. Monitors in general are associated with the following to ensure mutual exclusion of critical section:

- Entry queue
- Critical section
- Conditional queues

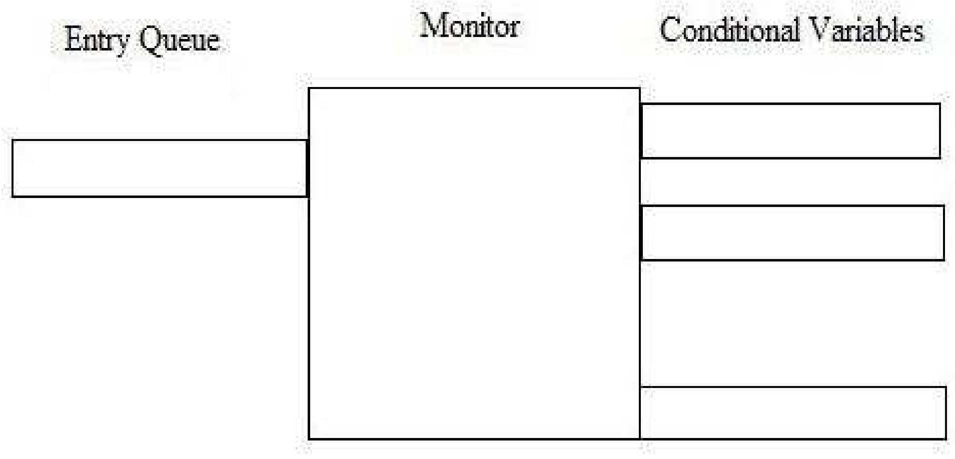


Figure 2.1 Monitor with entry queue and conditional queues

2.3 Mutual Exclusion

At any given time, only one thread can access the critical section. However, the monitor itself provides this mutual exclusion. The entry queue and the conditional queues are outside the critical section, and threads entering these queues do not hold the critical section. When a thread intends to enter the critical section, it first gets queued up in the *entry queue.* In the entry queue, all the threads compete to enter the *critical section.* However, when there is already a thread in the monitor accessing the critical section, other threads that are trying to enter the monitor at that time are queued in a queue called *conditional queue.* The threads in the entry queue execute a *wait()* method to wait on the conditional queue. The threads that are waiting in the conditional queue are outside the critical section. These threads should be woken up from the conditional queue to enter the monitor. This is done by the thread that is currently accessing the critical section by executing the methods *notify()* or *notifyall().*

2.4 Monitor Toolbox for Java

Monitor toolbox [3] is a program that simulates the monitor construct. They are classes that could be extended by regular Java classes. Once a regular java class extends the monitor toolbox classes viz. monitorSC or monitor SU, the regular class could act as a monitor class.

The regular class can act as a monitor class by doing the following;

- Extend class monitorSC or monitorSU
- Use operations enterMonitor() and exitMonitor() at the start and end of each public method
- Declare as many conditionVariables as needed
- Use operations waitC(), signalC(), signalCall(), length(), and empty(), on the conditionVariables

Though the simulated monitors are not easy to use or as efficient as the real monitors, they have the following advantages:

- A monitor toolbox can be used to simulate monitors in languages that do not support monitors directly. For example, as we show below, a monitor toolbox allows monitors to be used in C++/Win32/Pthreads programs
- Different versions of the toolbox can be created for different types of signals. Javas built-in monitors use SC signaling. An SU toolbox can be used to allow SU signaling in Java
- The toolbox can be extended to support testing and debugging

It is because of the above three advantages, monitor toolbox is used to simulate the monitor class. The regular Java class extends monitorSC if the signaling discipline adopted is Signal and Continue while it extends monitorSU if the signaling discipline adopted is Signal and Urgent wait [1]. The signaling disciplines are expained in details in the next section

2.5 Condition Variables

Condition synchronization is achieved by conditional variable and methods wait() and notify(). Condition variables are visualized as queue of threads waiting for a condition to become true. Condition Variables are declared as

ConditionVariable cv;

A thread tries to enter monitor from the entry queue if the monitor is empty. If the monitor is not empty, the thread executes *cv.wait()* to wait in the respective condition queue.

2.6 Signaling Disciplines

Once a thread has been signalled from a conditional queue, the behavior of the signaled thread depends on the signaling discipline chosen for the application. This section explains the following signaling disciplines in detail:

- Signal and Continue
- Signal and Urgent Wait
- Signal and Exit

To understand the signaling disciplines, let us consider a classic example of a multithreaded application Producer Consumer problem. Let us assume the following:

- Buffer capacity is 1.
- Both the producer threads and consumer threads execute *enterMonitor()* [1] method to enter the monitor.
- A producer can execute *deposit()* method and exit the monitor.
- A consumer can execute *withdraw()* method and exit the monitor.
- The condition variable associated with producers is *notFull.*
- The condition variable associated with consumers is *notEmpty.*
- Both the producer threads and consumer thread execute *exitMonitor()* method to exit the monitor

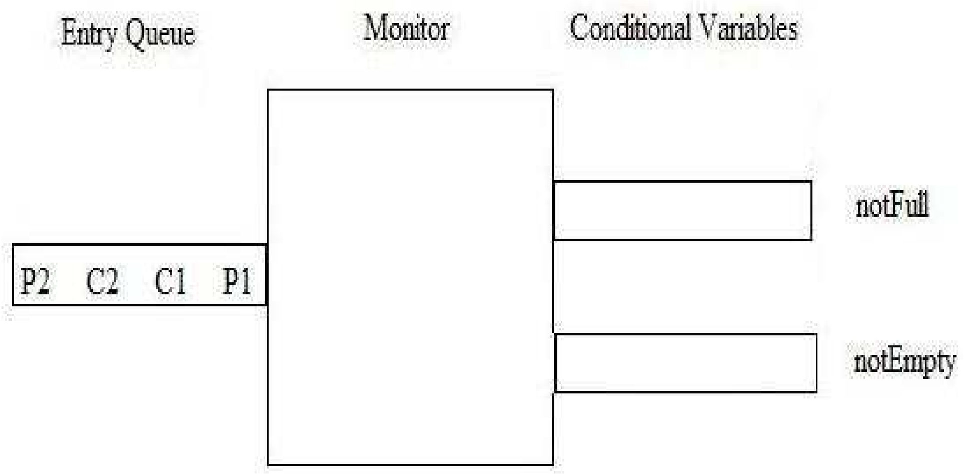


Figure 2.2 Entry queue with all threads populated

Let us assume that there are four threads P1, C1, C2 and P2 in the same order in the entry queue. In the entry queue, these four threads compete to enter the monitor. Since the monitor capacity is only 1, a *deposit()* method and *withdraw()* method should alternate. The execution of the application is shown below:

Now the buffer is empty and a producer is trying to enter the monitor. Since the monitor is empty, a producer can enter the monitor and deposit.

Now P1 has entered the monitor and executed the *deposit()* method to make the buffer value 1. After depositing, the thread P1 checks if the notEmpty condition queue is empty or not. Since at this point, no consumer threads are waiting in the notEmpty condition queue, the *signalCall()* method executed by P1 does not have any effect. P1, then executes *exitMonitor()* method to exit the monitor. Now C1 tries to enter the monitor. Since P1 has just deposited and made the buffer value 1, the thread C1 can thus enter the monitor to execute *withdraw()* method to make the buffer value 0 again.

After withdrawing, C1 checks if the notFull condition queue is empty or not. Since, no producer is waiting at the notFull condition queue, the signal-

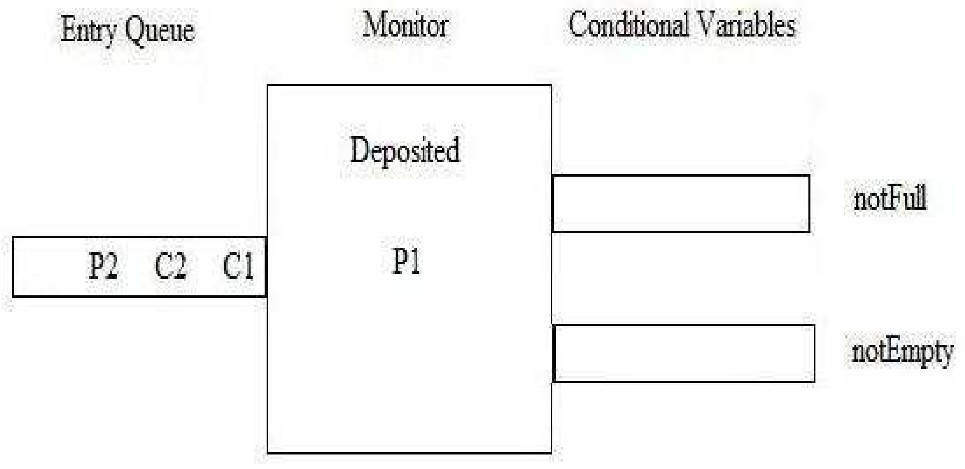


Figure 2.3 Producer 1 deposited.

Call() method executed by C1 does not have any effect. Then C1 finally executes *exitMonitor()* method to exit the monitor.

Now the buffer is empty again. Thread C2 tries to enter the monitor. Since there is nothing to withdraw, C2 blocks itself in the notEmpty conditional queue.

Since the buffer is empty now and the monitor is empty, P2 enters the monitor and deposits. Now P2 checks if the notEmpty condition queue is empty or not. Since at this point, C2 is waiting in the notEmpty condition queue, P2 executes *notFull.signalCall()* method to wake up *C2*.

Here is where signaling discipline comes into picture, the signalled thread can either enter the monitor immediately or go to the entry queue to compete again to enter the monitor. If the signaling discipline is Signal and Continue, the signalled thread goes to the entry queue, or else the signalled thread enters the monitor immediately. Each signaling discipline is explained in detail in the next section.

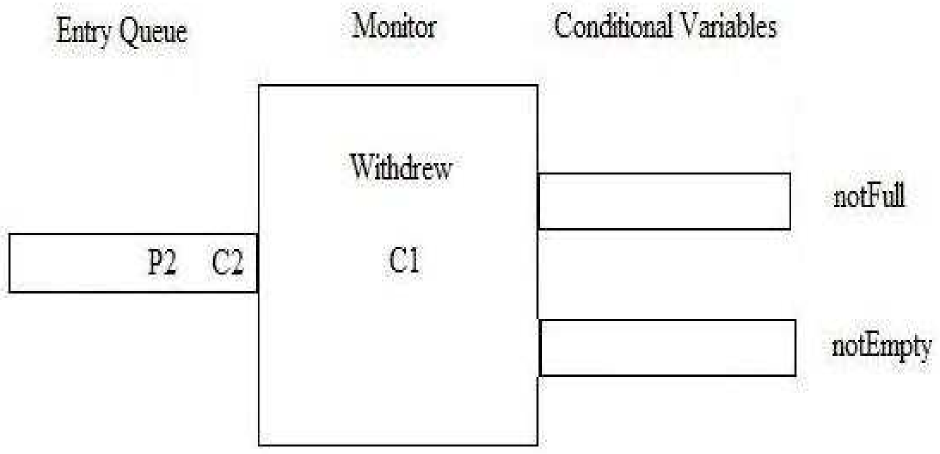


Figure 2.4 Consumer 1 deposited.

For now, let us assume that the signaling discipline is Signal and Continue. So the signalled thread C2 joins the entry queue and then competes in the entry queue to enter the monitor.

In this case, since there is no other thread in the entry queue and since a producer just deposited into the buffer, C2 gets the chance to enter the monitor.

After withdrawing, the buffer will again be empty. Then C2 checks if notFull condition queue is empty or not. Since no producer threads are waiting in the notFull condition queue at this point, the *notFull.signalCall()* method executed by C2 does not have any effect. Hence C2 then executes *exitMonitor()* to exit the monitor.

In the following subsection, the SC, SU and SE signaling disciplines are explained.

2.6.1 Signal and Continue

Signal and Continue is the default signaling discipline adopted by java monitors. In this discipline, the thread waiting in the conditional queue, when awakened, goes to the entry queue and competes in the entry queue to enter the monitor. The above example adopts signal and continue signaling discipline. It is to be noted

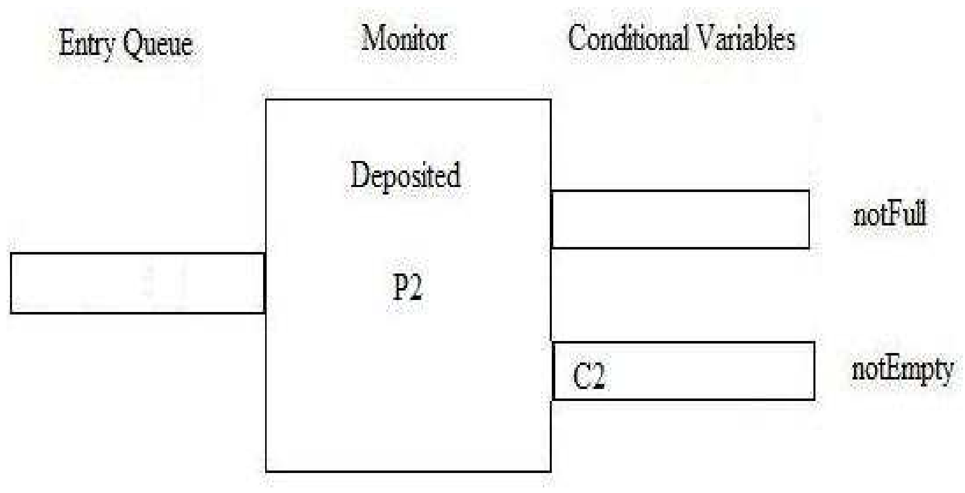


Figure 2.5 Consumer 2 executes wait() to wait in the conditional queue

that in the Fig. 2.6, the signalled thread C2 joins the entry queue and does not immediately enter the monitor.

2.6.2 Signal and Urgent Wait

In this signaling discipline, the thread waiting in the condition queue, when signalled, enters the monitor immediately. The signaler thread that wakes up the thread goes to a special type of queue called *re-entry queue* and the signaled thread enters the monitor immediately. After the signaled thread has completed its operation, then it checks if the re-entry queue is empty or not. If the re-entry queue is not empty then the thread in the re-entry queue gets more priority than the threads competing in the entry queue and enters the monitor. When the re-entry queue is empty, then the threads competing in the entry queue get the chance to enter the monitor. This approach basically prevents thread barging.

Thread barging denotes the act of a thread trying to enter the monitor ahead of the threads waiting before this thread to enter the monitor.

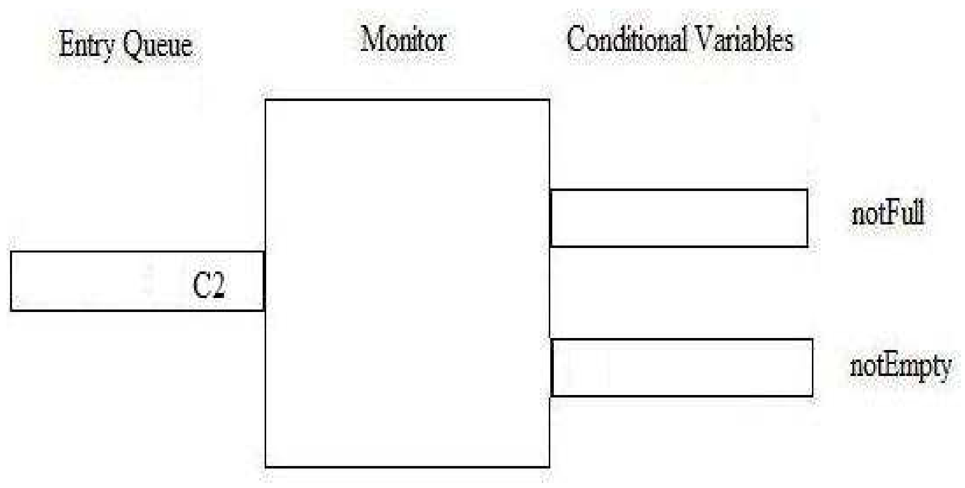


Figure 2.6 Consumer 2 is signalled to enter the entry queue again

2.6.3 Signal and Exit

Signal and Exit is similar to that of Signal and Urgent Wait but for the fact that when a thread executes the *signal_and_exit()* method, it does not go to the re-entry queue. Instead, the signaling thread executing *signal_and_exit()* method just exits the monitor. Hence the signal_and_exit() method is normally the last line of code in the method. This Signal and Exit discipline could be imagined as a special case of Signal and urgent wait.

2.7 Testing and debugging concurrent applications

Though multithreaded applications have considerable advantages, the most important concern is testing and debugging them. Synchronization is the key aspect of concurrent programming. Since monitors are used as one of the synchronization constructs, monitors play an important role in concurrent programming.

Since a concurrent application is multithreaded, the output obtained during one round of execution is not guaranteed for another round of execution with the same set of inputs. Hence this makes testing concurrent applications even more difficult. The basic approach of testing by running the application many times with

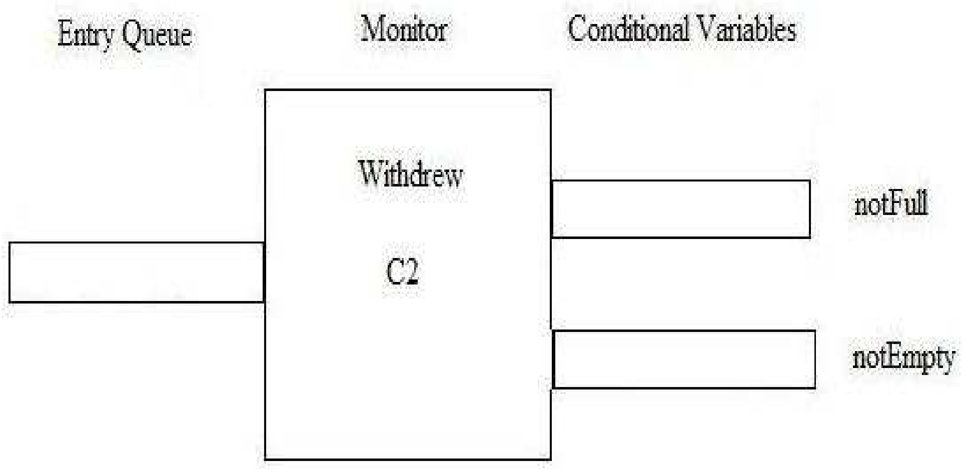


Figure 2.7 Consumer 2 enters monitor

the same set of inputs to analyze a discrepancy does not work with multithreaded applications.

Hence it becomes the responsibility of the programmers to write bug less code when it comes to concurrent programming. Testing multithreaded applications can thus becomes easier to an extent with the presence of a good debugger, which can visualize the threads so that the user can understand the value and state of the thread variables. JSWAT is one such Java debugger with lots of features explained in detail in the next chapter.

CHAPTER 3

JSWAT

3.1 JPDA Architecture

JSwat [6] is a graphical Java debugger front-end, written to use the Java Platform Debugger Architecture. JPDA [2] is a multi-tiered debugging architecture that allows tools developers to easily create debugger applications. JPDA consists of three layers:

- JVMDI - Java VM Debug Interface
- JDWP - Java Debug Wire Protocol
- JDI - Java Debug Interface

3.2 Components of JPDA Architecture

3.2.1 Debugee

The debuggee is the process being debugged, it consists of the application being debugged.

3.2.2 Java Virtual Machine (VM)

This refers to the VM running the application being debugged. The VM implements the Java Virtual Machine Debug Interface (JVMDI).

3.2.3 Back-end

The back-end of the debugger is responsible for communicating requests from the debugger front-end to the debuggee VM and getting the response back. The back-end communicates with the debuggee VM using the Java Virtual Machine

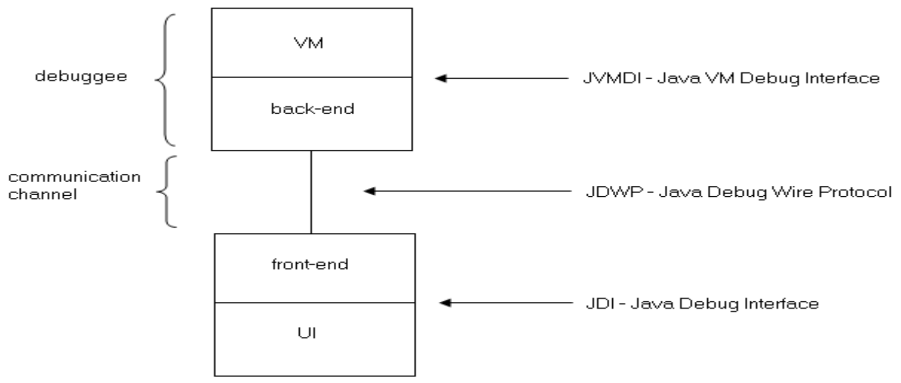

debuggee
communication channel
VM
back-end
front-end
UI
JVMDI - Java VM Debug Interface
JDWP - Java Debug Wire Protocol
JDI - Java Debug Interface

Figure 3.1 JPDA Architecture

Debug Interface (JVMDI) and communicates with the front-end over a communications channel using the Java Debug Wire Protocol (JDWP).

3.2.4 Communications channel

The communications channel is the link between the front and back ends of the debugger. The format and semantics of the serialized bit-stream flowing over the channel is specified by the Java Debug Wire Protocol (JDWP).

3.2.5 Front-end

The debugger front-end implements the high-level Java Debug Interface (JDI). The front-end uses the information from the low-level Java Debug Wire Protocol (JDWP).

3.2.6 User Interface (UI)

The graphical user interface (GUI) provided serves as test harness and as a starting point for the development of more complex GUIs. The example UIs are clients of the Java Debug Interface (JDI).

3.3 Debugger Interfaces

3.3.1 Java Virtual Machine Debugger Interface (JVMDI)

The JVMDI is a native interface implemented by the VM. It defines the services a VM must provide for debugging including requests for information (for example, current stack frame), actions (for example, set a breakpoint), and notification (for example, when a breakpoint has been hit). It also allows alternate communication channel implementations.

3.3.2 Java Debug Wire Protocol (JDWP)

The JDWP defines the format of information and requests transferred between the debuggee process and the debugger front-end. It does not define the transport mechanism. It allows the front-end to be written in a language other than Java, or the debuggee to be non-native (e.g. Java).

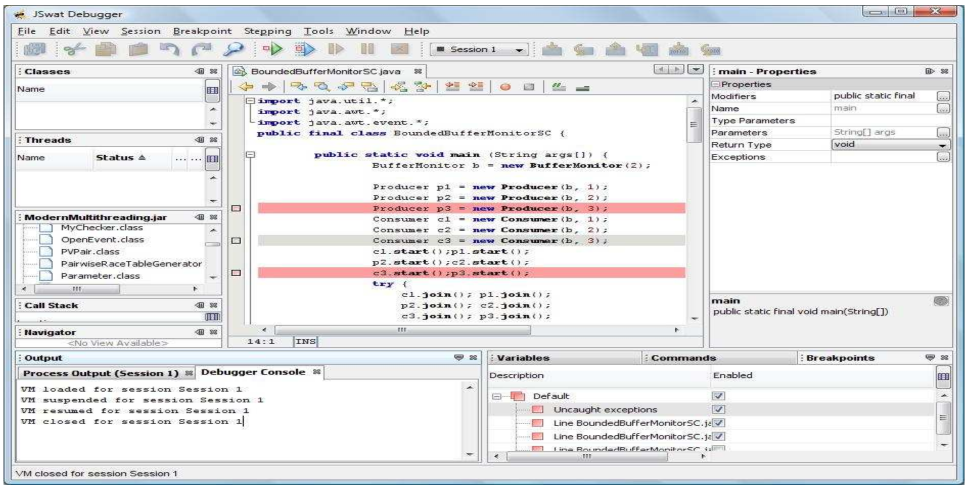
3.3.3 Java Debug Interface (JDI)

The JDI is a java interface implemented by the front-end, that defines information and requests at a user code level. This interface greatly facilitates the integration of debugging capabilities into development environments.

In this section, JSWAT debugger, its usage and its features are explained in detail. Jswat is an open source graphical java debugger. It is based on the Java Platform Debugger Architecture (JPDA). It is built in Netbeans Platform. Since the Jswat debugger is a stand-alone debugger, it can be used without Netbeans as well.

Some of the key features of JSWAT [6] are

- Sophisticated breakpoints
- Colorized source code display with code navigator
- Byte code viewer
- Movable display panels showing threads



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
```

```
VM loaded for session Session 1
VM suspended for session Session 1
VM resumed for session Session 1
VM closed for session Session 1
```

VM closed for session Session 1

Figure 3.2 JSWAT GUI

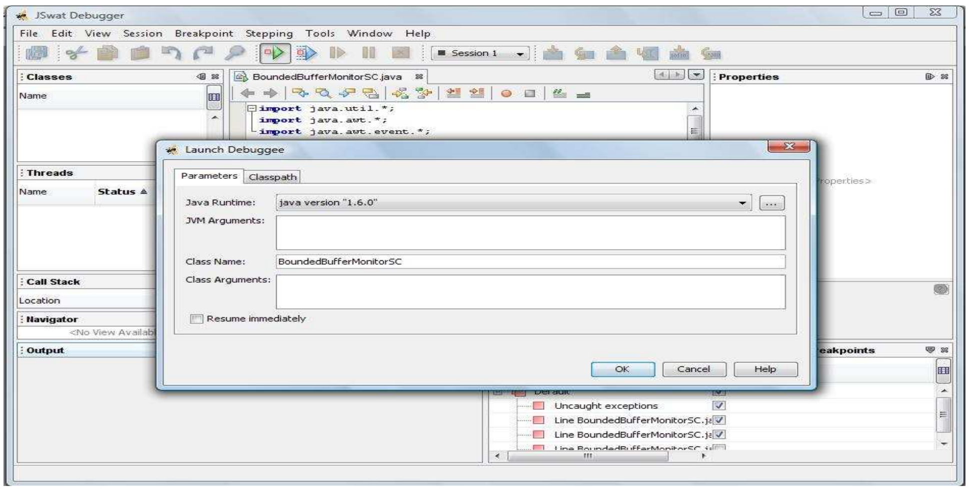
- Call stack
- Visible variable and loaded classes
- Java-like expression evaluation, including method invocation.

The actual display areas within the main window are called as windows themselves. These windows display the variables, threads, classes, breakpoints, sessions, and so on. Most of the windows are not present by default. If we need to see a specific display, then we need to drop down the *Window* menu and then choose the window that we need to be displayed. Also, these windows are movable. The user can have a window at any location within the main JSWAT window.

3.4 Usage and features of JSWAT

JSWAT could be used for any application with the following simple steps:

- Starting the Debuggee
- Setting the classpath and the sourcepath
- Setting breakpoints
- Stepping through code
- Display Variable values



BoundedBufferMonitorSC.java

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
```

Java Runtime: java version "1.6.0"
JVM Arguments:
Class Name: BoundedBufferMonitorSC
Class Arguments:
☐ Resume immediately

Uncaught exceptions	✓
Line BoundedBufferMonitorSC.ja	✓
Line BoundedBufferMonitorSC.ja	✓

Figure 3.3 Launching JSWAT

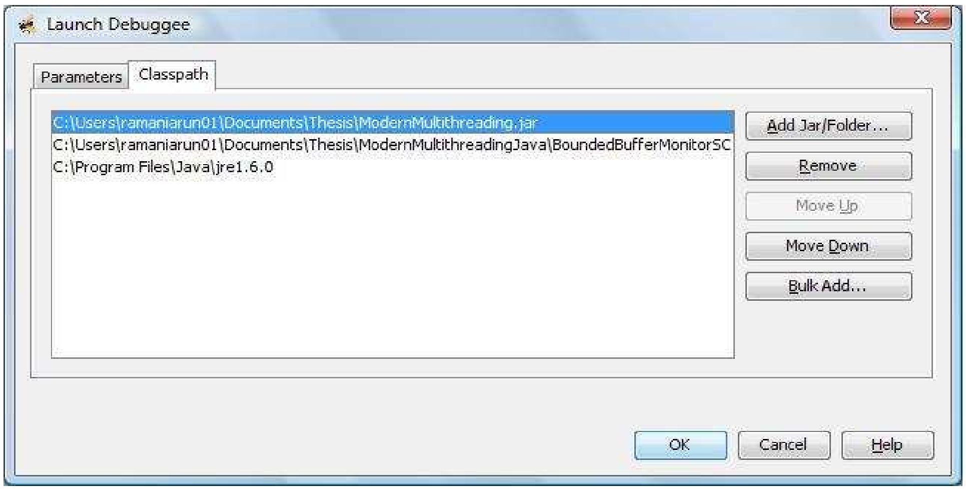
3.4.1 Starting the Debuggee

To launch the debuggee, select *Start* from the *Session* menu (or click on the corresponding toolbar button). The name of application's main class and the classpath that is normally used to launch your application should be provided to launch the application. As soon as the debuggee is started, it goes to the pause state for the user to signal it to start. To start the debuggee, the user should select *continue* from the *sessions menu.* The debuggee will continue to run until it hits a breakpoint or it exits normally.

3.4.2 Setting the classpath and the sourcepath

The classpath has to be set to run any application in JSWAT. This classpath can either be set while launching the application or by selecting *Settings* from the *Settings* menu.

While launching the application by clicking on the *start* button from the *Sessions* menu or by clicking the start icon from the toolbar, it opens a dialog to specify the main class name. There is also a tab in the dialog called *classpath.* We can click that tab and load all the classes that are required to run the application.



```
C:\Users\ramaniarun01\Documents\Thesis\ModernMultithreading.jar
C:\Users\ramaniarun01\Documents\Thesis\ModernMultithreadingJava\BoundedBufferMonitorSC
C:\Program Files\Java\jre1.6.0
```

Figure 3.4 Setting classpath from the JSWAT GUI

The classpath can also be set by selecting *Settings* from the *Sessions* menu. Doing so, will also result in the dialog to let the user to add all the classes required.

3.4.3 Setting Breakpoints

To set breakpoints, use the debugger to open the source file containing the code in which you want to stop. Scroll the editor to the desired line, then click the mouse 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.

3.4.4 Stepping Through Code

Once the debuggee has been launched and it hits a breakpoint, we can begin stepping through the code. We can do this 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 execute the current method and stop at the calling method. *Run to*

Cursor will set a breakpoint at the current cursor location, resume the debuggee so that it will hit the breakpoint, and then automatically delete the breakpoint.

3.4.5 Displaying Variable Values

There are three ways in which to view the values of variables. 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 you step through the code.

In addition to that view, there is the editor tooltip, whereby the value of the variable under the mouse pointer will be displayed as a 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 is 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. The user can type any Java-like expression and it will be evaluated and the result displayed in the window. This requires that the debuggee be stopped at a breakpoint in order to evaluate any variable references.

3.5 Enhancement in JSWAT

From the previous section, it could be inferred that there are lot of features available in JSWAT, which makes it not only fully functional but also user friendly. Inspite of all these features, there is one missing aspect in JSWAT, which this thesis would like to add on to it.

There is no special feature in JSWAT to visually display Java threads at run time. If there was a separate window, called as *Visualization Window* to visualize java threads at run time depicting the threads state, it will be really useful for concurrent programmers. It will help the concurrent programmers to better understand a multithreaded program thereby making the debugging phase of software development easier.

3.5.1 Current modules of JSWAT

The JSWAT debugger has 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 of these modules implements a feature. The visualization feature is added to the debugger by adding a file called *VisualizationWinTopComponent.java* to the *JSWAT Interface* module. The first step in adding the visualization feature to JSWAT is to create a *visualization window.* This visualization window is a user interface built using java swing. It contains a list of text boxes to visualize threads.

This visualization window is created in the package *com.bluemarsh.jswat.ui.components.* Then the next task is to pass the threads to be visualized to the visualization window. This code is added to the action event of the *OK* button of the *Launch Debugee Panel.* It gets all the useful information from the user program and passes it on to the visualization window.

This thesis work adds value to JSWAT by adding the visualization window to JSWAT and integrating it in a synchronous manner along with JSWAT. The process of integration of the visualization window and the JSWAT in a synchronous manner is explained in the forthcoming chapters.

CHAPTER 4

EVENT RECOGNIZER

The Event recognizer collects the data required to visualize the java threads involved in the user program. The visualization as such should capture the following in order to be useful to the programmers:

- The threads competing in the entry queue
- The thread entering the monitor
- The thread that waits on the condition queue due to a failed condition
- The signaler thread that signals thread(s) in the conditional queue based on the signaling discipline chosen
- The signaled threads behavior based on the signaling discipline
- The thread that exits the monitor

It comprises of three components and acts in a synchronized manner to collect these data. This chapter intends to explain all the three layers in detail.

4.1 Components of the Event Recognizer

The Event Recognizer comprises of three components. They are *Component to intiate visualization*, *Component to capture the events from the user program* and *Component to Synchronize the threads*. The user application is a multithreaded program which is loaded to the JSWAT debugger. It contains multiple threads that are created on the fly to share a common resource and achieve the applications requirement. The event recognizer captures the data required to visualize the threads in the user application. To visualize the threads, we need to capture the following:

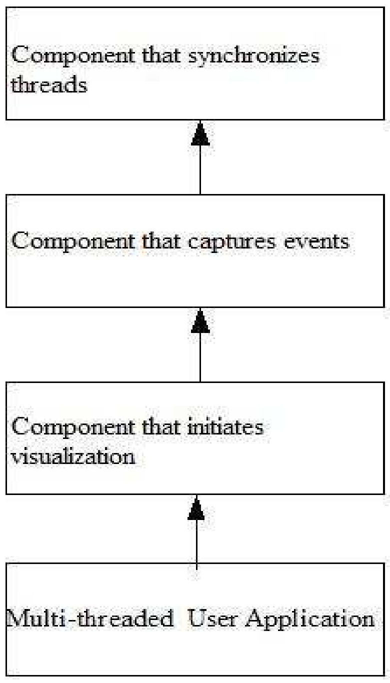
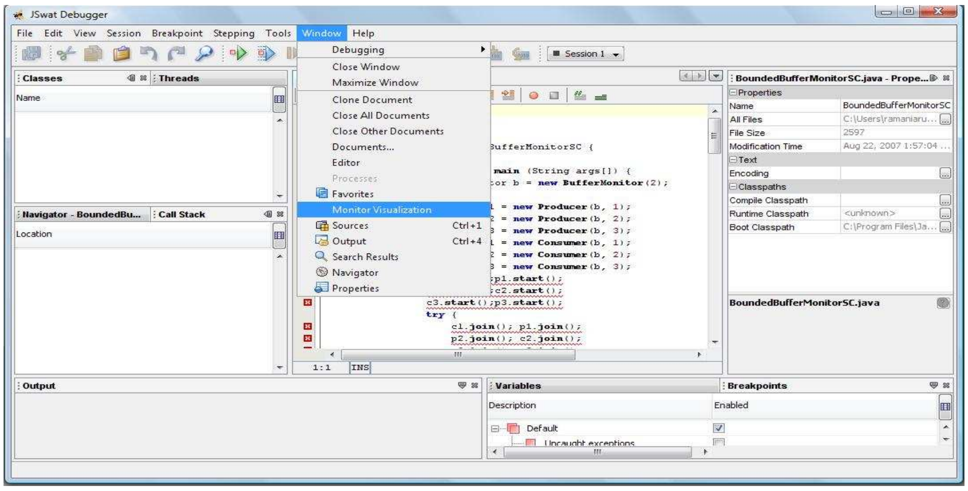

Component that synchronizes threads
Component that captures events
Component that initiates visualization
Multi-threaded User Application

Figure 4.1 Components of the Visualization Tool

- To find the threads competing in the entry queue, we need to capture the instance at which each thread executes its run() method.
- To find the thread that enters the monitor, we need to capture the instance at which a thread executes the enterMonitor() method.
- To find the threads that enter the conditional queues, we need to capture the instance at which a thread executes the wait() method.
- To find the threads that signal the thread in the conditional queue, we need to capture the instance at which the thread executes the signalCall() method depending on the user application.
- To find the threads that exits the monitor, we need to capture the instance at which a thread executes the exit() method.

Though the event recognizer collects the information, it has to be invoked by the component that initiates visualization. The next section explains about the component that initiates visualization.



```
BufferMonitorSC {
 main (String args[]) {
:or b = new BufferMonitor(2);
1 = new Producer(b, 1);
2 = new Producer(b, 2);
3 = new Producer(b, 3);
1 = new Consumer(b, 1);
2 = new Consumer(b, 2);
3 = new Consumer(b, 3);
;p1.start();
;c2.start();
c3.start();p3.start();
try {
    c1.join(); p1.join();
    p2.join(); c2.join();
```

BoundedBufferMonitorSC.java - Prope...

Properties	
Name	BoundedBufferMonitorSC
All Files	C:\Users\ramaniaru...
File Size	2597
Modification Time	Aug 22, 2007 1:57:04 ...
Text	
Encoding	
Classpaths	
Compile Classpath	
Runtime Classpath	<unknown>
Boot Classpath	C:\Program Files\Ja...

BoundedBufferMonitorSC.java

Figure 4.2 Monitor Specification

4.2 Component to intiate visualization

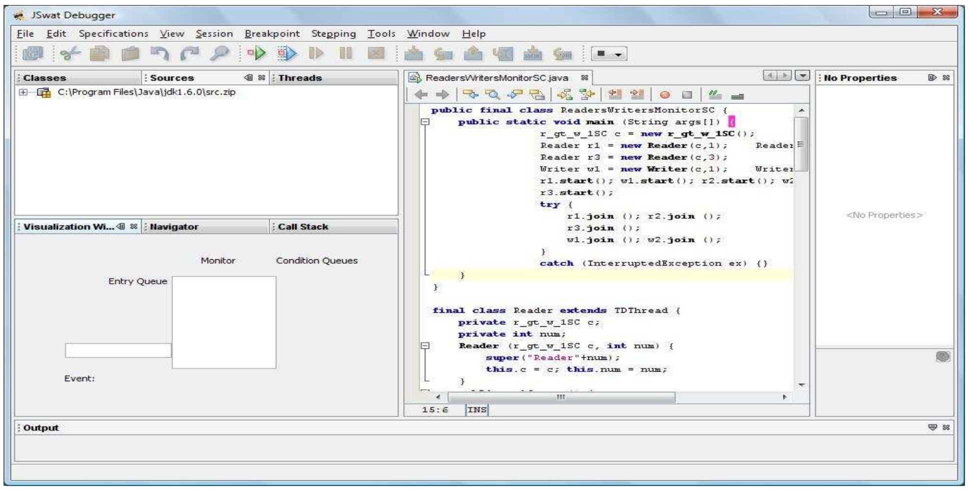
In this section, we explain the role of the Component that intiates visualization in the data collection process. It actually acts as an on / off switch for the visualization. The Component thus has the following roles:

- Act as an on/off switch for the visualization
- Invoke the module to capture the data required for visualization

4.2.1 Initiating the visualization

This Component acts as a gateway to the monitor visualization. This is controlled by a drop down menu in the JSWAT window. JSWAT has a menu called *Window*. We need to drop down the window menu from JSWAT and select *Monitor Visualization* to have the visualization.

If the user does not select this option, the user will not be able to see the visualization of threads. The application is designed this way so that there is no real compulsion on the users to use our enhancement along with the JSWAT debugger. However, if they choose to use the visualization, they can by selecting Monitor Visualization from the Window menu. Once the user selects the monitor



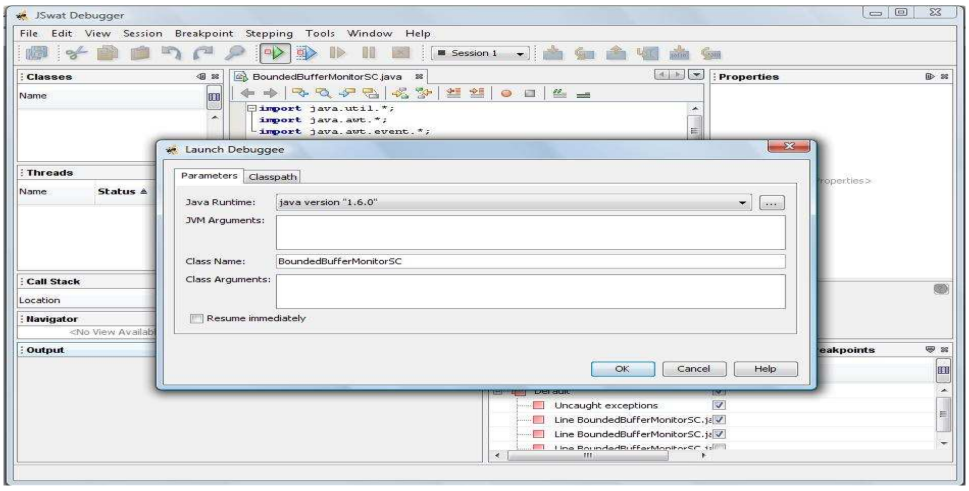
```
public final class ReadersWritersMonitorSC {
    public static void main (String args[]) {
                r_gt_w_1SC c = new r_gt_w_1SC();
                Reader r1 = new Reader(c,1);    Reader
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer
                r1.start(); w1.start(); r2.start(); w2
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}
    }
}

final class Reader extends TDThread {
    private r_gt_w_1SC c;
    private int num;
    Reader (r_gt_w_1SC c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 4.3 Visualization Window

visualization, he or she can see the visualization window being docked to one of the windows in the JSWAT user interface.

The visualization window is so designed that it will take a form based on the user program. For example; the basic framework of the visualization window consists of an entry queue, a monitor and the conditional queues. However, the number of conditional queues and the presence or absence of a re-entry queue depends on the user application and the signaling discipline adopted.

As you might see in the above figure, the visualization window may or may not have the re-entry queue based on the signaling discipline adopted in the user program. If the user application adopts *Signal and Continue*, there is no re-entry queue. If the user application uses *Signal and Urgent Wait* or *Signal and Exit* signaling discipline, the visualization window will have a re-entry queue. Also, the conditional queues are not shown at this point. The condition variables are computed in the next subsection after the component to capture the events from the user application. Hence the Visualization window is dynamic when it comes to choosing its components.



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
```

Java Runtime: java version "1.6.0"
JVM Arguments:
Class Name: BoundedBufferMonitorSC
Class Arguments:
☐ Resume immediately

Figure 4.4 Launch Debugee Panel

4.2.2 Invoking the Event Recognizer

The *OK* button in the Launch Debugee dialog acts as an Component that intiates visualization. When the user clicks on the Ok button, the visualization window is first altered based on the signaling discipline chosen and then the component to capture the events from the user application is called to get the data required to visualize the threads in the user application.

4.3 Component to capture the events from the user program

The Component that captures the events from the user program is the most important component of the event recognizer. It is responsible for collecting the data required to visualize the threads. In this application, this component is totally based on the concepts of *regular expressions.* Regular expressions are so powerful part of Java language that the component in this application has made the best use of it.

Regular expressions are also very flexible. Since the user program can be any multithreaded application, such a flexible base was required to rely upon to collect the data required. To use the regular expressions in java, *java.util.regex.Pattern* and *java.util.Matcher* class are imported.

4.3.1 Pattern

The pattern class enables us to specify the pattern that we are looking for in the user application. The steps involved in using the Pattern class are as follows:

- A regular expression is specified as a string
- This string is compiled as an instance of the Pattern class

Now the mentioned regular expression is compiled as an instance of the Pattern class. Then the matcher class is used to match the pattern in the user program. The matcher class is explained in the next section

4.3.2 Matcher

The matcher class enables us to check for a match of the pattern in the user program. The pattern instance mentioned in the above section is used to create a Matcher object that matches user program against the regular expression. The *matches* method is then used to find if there is a match of the user program against the regular expression. Here is where the flexibility of the regex class comes into picture. The regular expressions can be a combination of characters, numbers, special characters etc. The number of occurrences of a character or a sequence can also be specified. All these features of the regex class made it an ideal base for this thesis work.

```
String teststr = [a-z][0-9];
Pattern p = Pattern.compile(teststr);
Matcher m = p.matcher(String to match);
boolean b = m.matches();
```

Figure 4.5 Code Snippet to explain regular expressions

```
Public void deposit(int value) {
         enterMonitor("deposit");
while (fullSlots == capacity)  {
         notFull.waitC();
}
buffer[in] = value;
in = (in + 1)% capacity;
++fullSlots;
exerciseEvent("deposit");
notEmpty.signalCall();
exitMonitor();
}
```

Figure 4.6 Code Snippet to deposit using monitor

4.3.3 Capturing

Another important feature of the regex class is the capturing of matched messages. The messages matched are captured as *groups*. This was very important for this thesis work. For instance, a same word or sequence of characters can be broken down into different groups of messages. This feature enables us to obtain the needed information from the matched string.

Let us consider the following sample code snippet. This code snippet is a method that is used to deposit some value into the monitor.

Several threads execute the above code snippet [1]. The following messages are of importance from the above listing:

- notFull.waitC();
- enterMonitor
- exerciseEvent(deposit);
- notEmpty.signalCall();
- exitMonitor();

These messages are the events that cause the threads to navigate between its different states. Hence it is not only important to find these messages in the code but it is also important to know which thread executes this method at which

```
String patterncondQueues = "([A-Za-z]+).(waitC\\(\\))";
String patternSigQueuesall = "\\b([A-za-z]+[0-9]?).
                             (signalCall\\(\\))";
Pattern pattern1 = Pattern.compile(patterncondQueues);
Pattern pattern2 = Pattern.compile(patternSigQueuesall);
Matcher matcher1 = pattern4.matcher(nextToken);
Matcher matcher2 = pattern5.matcher(nextToken);
boolean matchFoundCondQueues = matcher1.find();
boolean matchFoundSigQueuesall = matcher2.find();
```

Figure 4.7 Code Snippet to explain the use of regex in data gathering

instance. This is important to synchronize the visualization with the JSWAT debugger.

Now, we let you consider that the code snippet mentioned above as a String say *InputString*. The Inputstring is tokenized as individual lines and the following code could be used to find the occurrence of the message noFull.waitC() and notEmpty.SignalCall()

Since we do not know the conditional queue name before, we have a regular expression for the conditional queue name and the conditional queue name from which the thread has to be signaled and the conditional queue name can be captured using a method called *group* of the *matcher* class. For instance, *matcher1.group(1)* would give the conditional queue name for the method *waitC()*.

The next step is to have the captured messages in a temporary buffer storage so that the Visualizer thread can use these messages to visualize the threads. If we have this temporary buffer, then it should be synchronized as well. One such synchronized storage is *Synchronous Queue*. The synchronous queue is used as a temporary buffer and the captured messages are put in the synchronous queue as and when the component finds the messages of interest. The usage of Synchronous queue is explained in the next section.

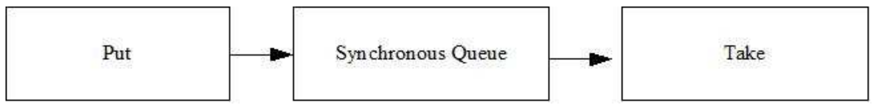


Figure 4.8 Synchronous Queue

4.4 Component to Synchronize the threads

In this section, the use of Synchronous queue and its importance in synchronization is explained in detail. Synchronous queue is used as an intermediate layer between the event recognizer and the visualizer.

4.4.1 Why do we need Synchronous queue?

Synchronous queue [8] is a blocking queue in which each take must wait for a put, and vice versa. Hence, Synchronous queue does not only act as a temporary storage between the event recognizer and the visualizer, but it also helps in synchronizing the threads with respect to the messages inserted into the queue i.e, Using Synchronous queue, we can guarantee that the message that is inserted first into the queue is read first out of the queue. This feature enables us to successfully integrate the visualization tool with the JSWAT debugger in synchronous with each other as well.

4.4.2 Put()

Once the desired messages are captured from the user program, the next step as mentioned would be to insert those messages in the synchronous queue. At this point, care is taken to ensure the discovered statements and the inserts into the synchronous queue as atomic statements. This will ensure the order of occurrence of the events of the thread and the order of events pushed in the queue is always same.

4.4.3 Take()

As discussed earlier, the synchronous queue is a blocking queue in which the put and take methods to and from the queue should alternate. Hence as soon as a message is inserted into the queue, there has to be a *take()* method from the queue to retrieve the inserted message and send it to the visualizer. The relationship between the take method from the synchronous queue and the visualizer is explained later.

CHAPTER 5

VISUALIZER

In this chapter, the visualization window and the visualizing rules are explained in detail. Once the event recognizer puts all the messages in the synchronous queue, it is the responsibility of the visualizer to figure out some meaning out of the messages and visualize them accordingly. This chapter deals with two sections, viz. the visualizer and the visualization window.

5.1 Visualizer

The visualizer is the component that is responsible for deciding the following from the messages obtained from the synchronous queue:

- Which thread to be focused on
- Where the thread should be displayed at this instance.
- What should be the state of other threads.

All the information required to decide on the above three criteria is obtained from the synchronous queue. The event recognizer pushes the information to the synchronous queue in a specific format which the visualizer understands. The visualizer then reads from the queue and interprets the action to be performed based on the messages. The visualizer has a set of actions to perform based on the messages read. For instance, if the message read from the synchronous queue is *P1 notFull wait*, it knows that the thread *Producer 1* has executed the *wait()* on the conditional queue *notFull*.

```
public void deposit(int value)
{
  enterMonitor("deposit");
  while (fullSlots == capacity)
  {
            notFull.waitC();
  }
  buffer[in] = value;
  in = (in + 1) % capacity;
  ++fullSlots;
  exerciseEvent("deposit");
  notEmpty.signalCall();
  exitMonitor();
}
```

Figure 5.1 Code Snippet to show the visualizers role

The following would be the messages pushed into the synchronous queue when a thread executes the above code snippet [1]:

- Thread Name waitC notFull MonitorSC
- Thread Name enterMonitor MonitorSC
- Thread Name signalCall notEmpty MonitorSC
- Thread Name exitMonitor MonitorSC

When the visualizer reads these messages from the Synchronous queue, it will interpret these messages and will act as following:

- *Thread Name waitC notFull MonitorSC* – The visualizer will move the thread that executed this method to the notFull conditional queue.
- *Thread Name enterMonitor MonitorSC* – The visualizer will move the thread to the monitor.
- *Thread Name signalCall notEmpty MonitorSC* – The visualizer will move all the threads in the notEmpty conditional queue to the entry queue. Note that the signaling discipline is also sent to the queue. Since the visualizer knows the signaling discipline, it knows what to do next. For instance, if the signaling discipline was Signal and Urgent wait, then a signal message

will make the signaled thread to enter the monitor instantly and the signaler thread waits in the re-entry queue whereas if the signaling discipline is Signal and continue, when a thread executes the signalC, the signaler thread signals a thread in the conditional queue and continues to remain in the monitor till it exits. The signaled thread moves to the entry queue and competes to enter the monitor after the signaler thread exits the monitor.

- *Thread Name exitMonitor MonitorSC* – The visualizer knows to remove the thread from the monitor.

Similarly, the visualizer is coded to handle all other events that the thread executes. Hence the visualizer can be visualized as a driver consists of set of rules that assists in the actual visualization of thread.

5.2 Visualization Window

In this section, the actual display components are explained in detail. The Visualization window is built using Java Swing. The visualization window is docked in within the JSWAT debugger user interface and it has text boxes for the following:

- Entry Queue
- Monitor
- Conditional Queues (Number depends on the user application)
- Re-entry Queue (if the signaling discipline is Signal and Urgent Wait or Signal and Exit.)

The visualization window displays the threads in the respective textboxes as instructed by the Visualizer. Hence, this visualization window has to be integrated with the JSWAT user interface in a synchronous manner so that when a user pauses the execution or sets break points in the debugger, the visualization should also halt at that point and show the current status of the threads in the visualization window. When the user clicks on resume, the visualization should carry on from that point.

This aspect of integrating the visualization window with the JSWAT synchronously requires the thread to be synchronized. This is where synchronous queue comes into picture. Since the synchronous queue allows synchronized access to it and also ensures that the data pushed in the queue is read first before the next message is pushed in, the order of messages is maintained while reading the messages from the queue and hence the display of the threads in the respective text boxes in the visualization window is synchronized with the user application.

This approach of integrating the visualization window with the JSWAT in a synchronized manner has a lot of advantages. The main advantage of this approach is that the visualization window can be controlled in many different ways.

The user can set break points on those lines of code which he or she desires to analyze in the user application before starting the execution and can start the debugger. When the debugger encounters the line of code where the break point is set, it stops the visualization too and the user can thus see the current status of the threads in the respective text boxes. Then when the user clicks on resume in the JSWAT debugger, the visualization and execution continues from that point. Hence by this approach, the chance of finding a bug is more for a user and it also enables the user to better understand the user application.

If the user does not set break points in the user application, the visualization goes hand in hand with the user program such that the user can actually see the application being executed by the debugger and also the visualized output of the threads in the visualization window in parallel. This way, the visualization window is dynamic and it is only because of this approach of synchronous integration, that the visualization guarantees the order of execution of the threads in the user program.

Another advantage of the synchronous integration of the visualization window with the JSWAT user interface is that it gives the user an option to pause the execution at any point in time during the execution, where the threads are not visualized as expected. Doing so, will cause the visualization also to halt at that point. Hence it will be easier for the user to trace till that point and fix any bug that might persist or synchronize the threads in the user program.

Hence, the visualization window helps in debugging applications in the above-mentioned three ways. Since multithreaded applications are difficult to debug, the visualization window helps the user to better understand the program and debug them.

CHAPTER 6

INTEGRATOR

In this chapter, the actual integration of the visualization window with the JSWAT is explained. As explained in the previous chapter, the first task involved in visualizing the threads is to collect the data required to visualize them from the user program. Once this is done, the collected data are put in the synchronous queue. The synchronous queue is used both for communication and synchronization. The next step is to take the data from the synchronous queue and send it to the visualizer which processes the data and visualizes accordingly.

Now, the problem involved in sending the data from the synchronous queue to the visualizer is that they both are in different package altogether but still they have to communicate with each other. The user application is a separate package of its own and the visualizer resides in the package of JSWAT. Hence it was a tough task in making them communicate with each other.

Since there was not a direct means to make those two packages communicate with each other, a common medium was necessary through which the messages can be exchanged. This is where *sockets* helped. A socket acts as a communication medium through which the data taken from the *synchronous queue* [8] can be sent to the visualizer so that the visualization could happen from the package inside JSWAT. Hence the data flow can be visualized as follows:

In the figure 6.2, the data flows from user program to the Visualization window. The relevant data required to visualize the threads are collected from the user program and is put in a synchronous queue. Then the data is taken out from the synchronous queue and sent to the visualizer through sockets. The visualizer drives the visualization and the threads are visualized in the visualization window.

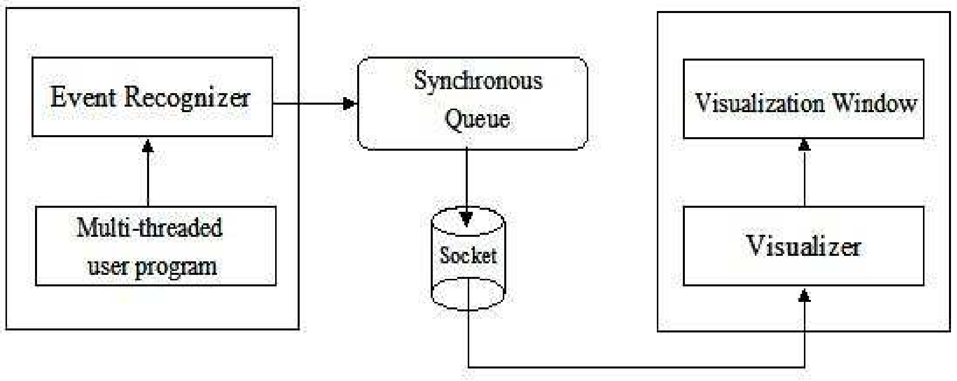


Figure 6.1 Integrator

The point to be noted here is that the user program is in a different package of its own and the visualizer and the visualization window are in package with JSWAT. The presence of a synchronous queue helps in synchronization and the sockets help in actual data transfer from synchronous queue to the visualizer.

The next aspect to consider is that the sockets should be capable of handling messages sent my multiple threads. Hence socket should be of *asynchronous* type to help in this situation. Asynchronous socket programming has enabled the visualizer to keep listening to the messages sent from the synchronous queue.

Asynchronous socket, by nature lets the server to keep listening to requests sent by the client from the other end of the socket. To achieve this, the server has to be multithreaded. Basically, there has to be a listener thread dedicated to listening to requests from the client and *worker threads* or *handler threads* to process the requests from the client. In this case, the handler threads are created as and when the listener thread receives a request from the client. This concept is incorporated in this thesis work as well.

The Visualizer here has a listener thread which keeps listening to the client request. As soon as it receives a request, it creates a handler thread and goes back to the listening mode to listen to the next request and this process goes on. It is

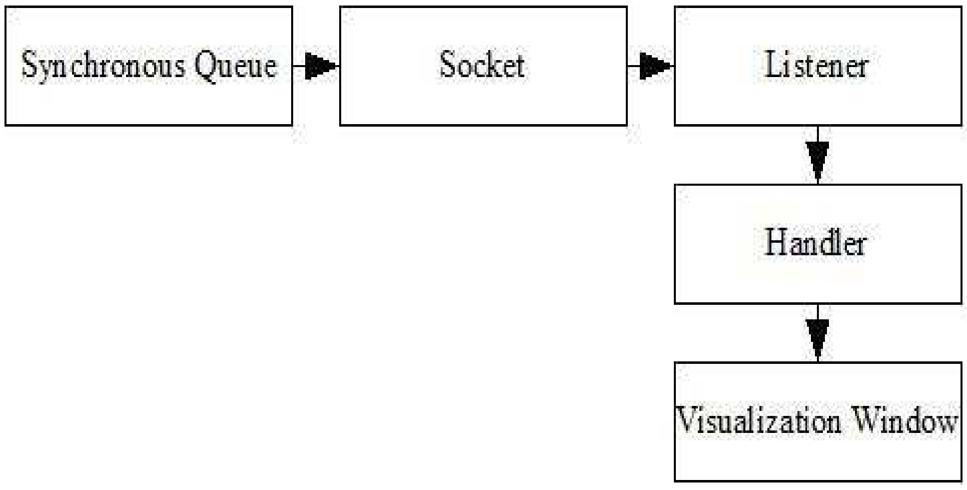


Figure 6.2 Asynchronous Socket

because of this multithreaded socket programming approach, the visualization can happen seamlessly as and when the events are executed.

As shown in the 6.2, the Visualizer actually consists of a listener thread that keeps listening to the messages sent from the synchronous queue. As soon as the listener thread receives a message, it creates a handler thread to process the message and display the threads in the respective text boxes in the visualization window.

One of the main advantages of having this asynchronous socket programming approach is that since a thread is dedicated to listening to the messages from the synchronous queue, it could be guaranteed that no message would be dropped. Also, since the socket is of *non-blocking* type, the visualization is not blocked at any point, waiting for messages from the client. Also, because a handler thread is created whenever a message is received by the listener thread and the handler thread handles the visualization of the threads, the messages are sent to the listener thread immediately after the threads in the user program executes an event, the visualization as such occurs seamlessly with respect to the execution of events in the user application.

Though the multithreaded socket programming approach has the above mentioned advantages, it leaves an overhead of synchronizing the threads. However, since Java has very good support for multithreaded applications, it is not a considerable overhead and hence can be managed well. Hence, this asynchronous socket is an apt integrator for this situation. The messages could be transferred through the socket as long as the messages are being pushed in the synchronous queue and finally when the JSWAT window is closed, the connection also terminates.

This approach also provides a level of abstraction to the users. JSWAT users can use the debugger in the normal way without having to do any special operations to have the visualization. The user has to just choose to open the monitor visualization window and run the debugger just the way he or she would in the absence of the visualization window.

CHAPTER 7

USE CASES AND SEQUENCE DIAGRAM

7.1 Use Cases

In this section, the use cases involved with the application are explained. The use case diagram for the application is shown in fig. 7.1.

7.2 Sequence Diagram

In this section, the sequence of execution is shown along with the screen shots at different instances.

As seen in the Figure 7.2, the user will perform the following steps in order:

- Open the file using the File $->$ Open menu
- Optionally set or unset Break points
- Open Visualization Window
- Select Run to specify the main class name, classpath to the application and the source path
- Optionally chooses pause, continue or stop while the application is running or stopped.
- See the visualization in the Visualization Window.

7.3 Experimental Results

7.3.1 Producer-Consumer problem

Let us consider a producer - consumer problem. Producer - Consumer problem is a classic problem involving a list of producer threads trying to deposit on to a bounded buffer and a list of consumer threads trying to withdraw from the bounded buffer. In this experiment, let us visualize these producer and consumer

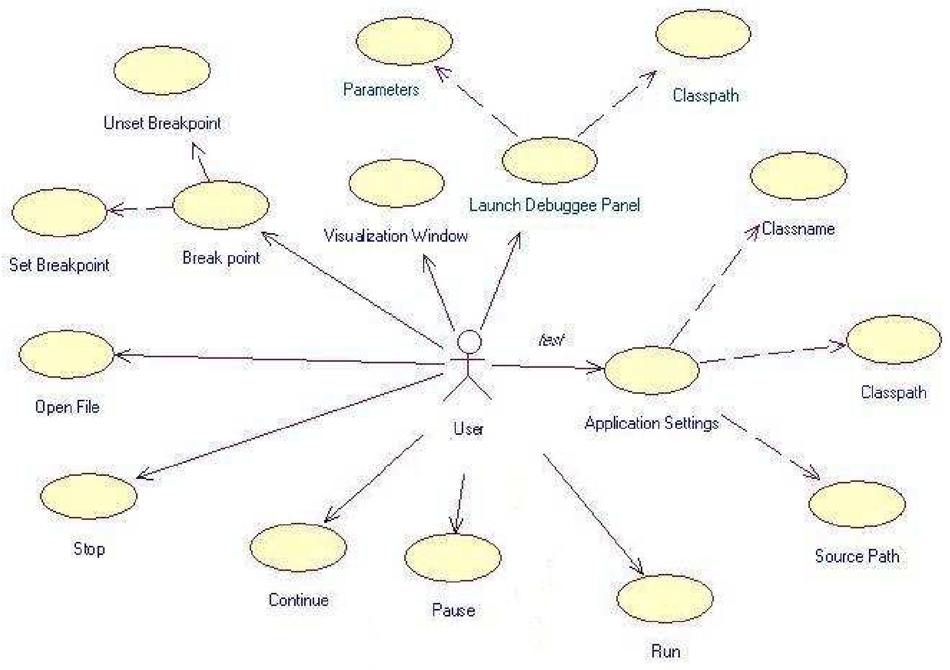


Figure 7.1 Use Case diagram

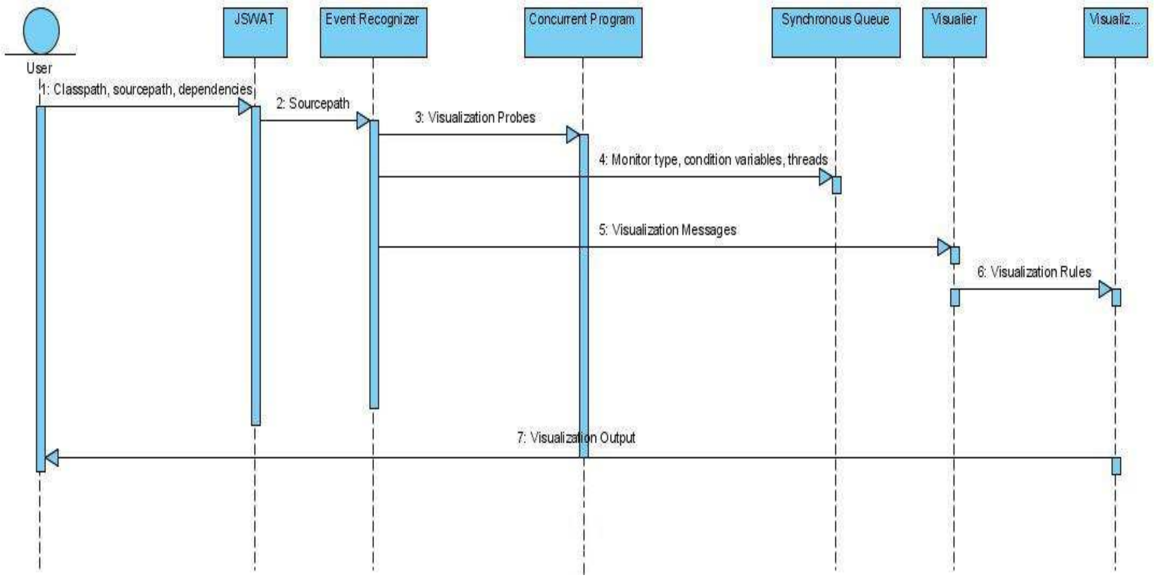
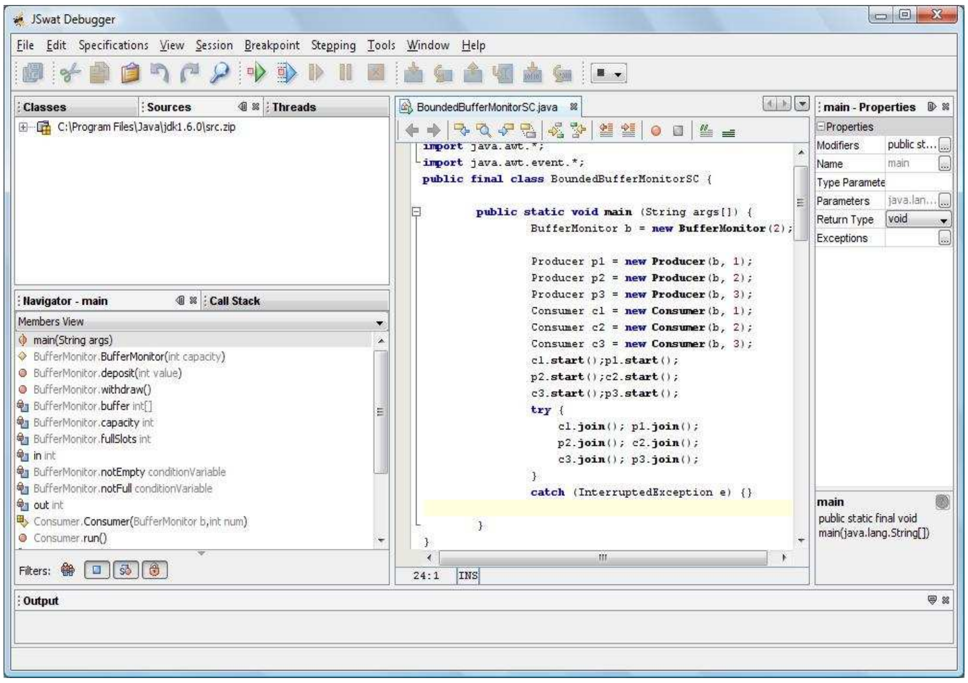


Figure 7.2 Sequence diagram


JSwat Debugger
File Edit Specifications View Session Breakpoint Stepping Tools Window Help
Classes
Sources
Threads
C:\Program Files\Java\jdk1.6.0\src.zip
Navigator - main
Call Stack
Members View
main(String args)
BufferMonitor.BufferMonitor(int capacity)
BufferMonitor.deposit(int value)
BufferMonitor.withdraw()
BufferMonitor.buffer int[]
BufferMonitor.capacity int
BufferMonitor.fullSlots int
in int
BufferMonitor.notEmpty conditionVariable
BufferMonitor.notFull conditionVariable
out int
Consumer.Consumer(BufferMonitor b,int num)
Consumer.run()
Filters:
BoundedBufferMonitorSC.java
main - Properties
Properties
Modifiers
public st...
Name
main
Type Paramete
Parameters
java.lan...
Return Type
void
Exceptions
main
public static final void main(java.lang.String[])
Output
24:1
INS

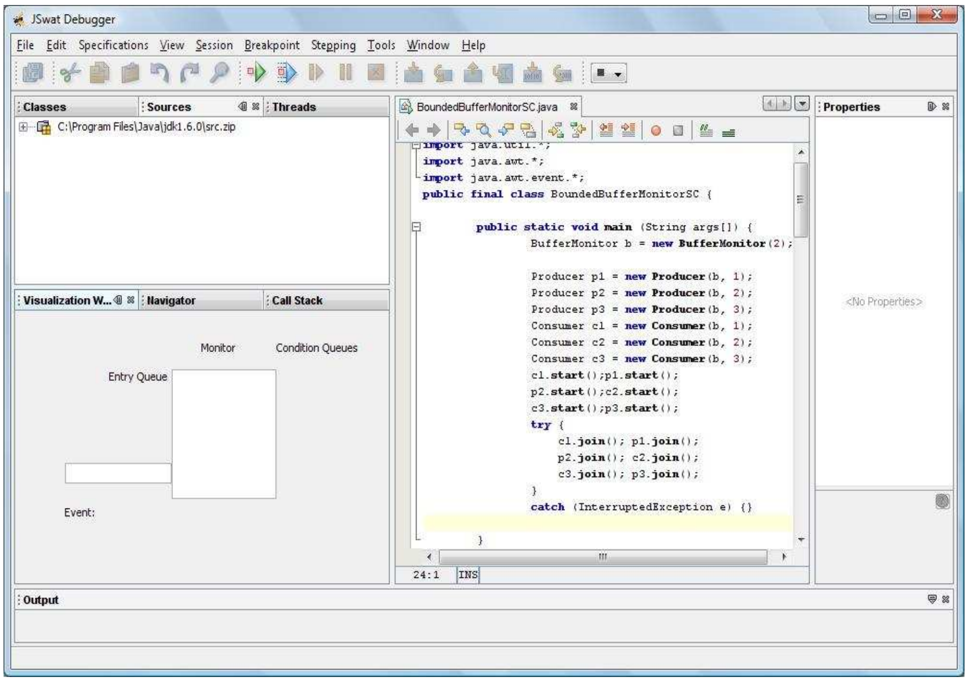
```
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.3 Producer - Consumer problem with SC Signaling discipline

threads. There are three Producer threads trying to deposit and three consumer threads created as showin in Figure. 7.3.

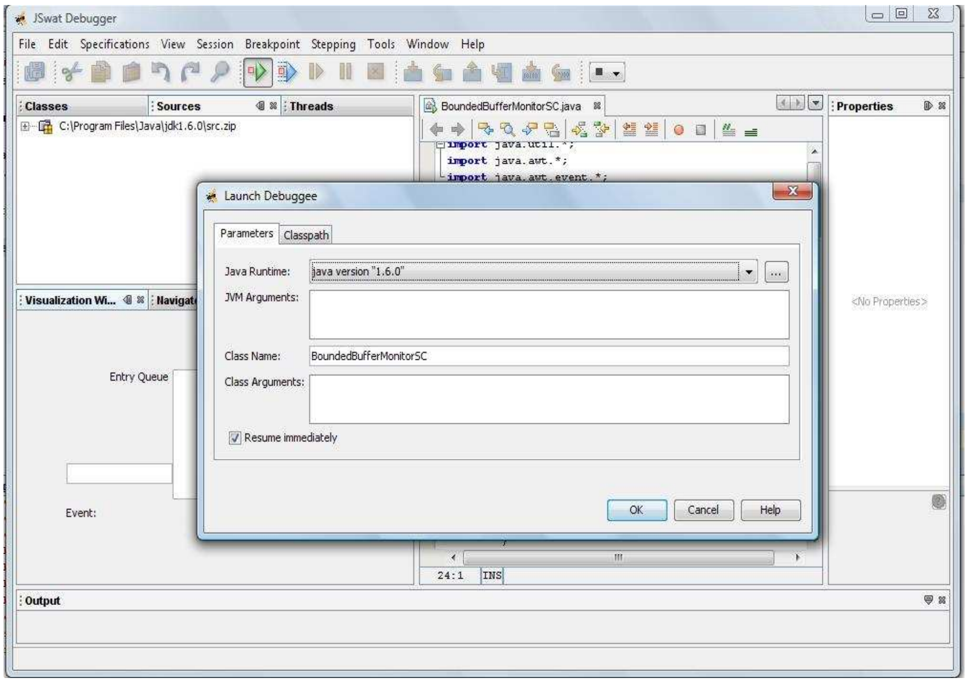
In the next step, the visualization window is opened so as to visualize the threads. It is to be noted that at this stage, the visualization window does not have the condition queues. The condition queues are displayed dynamically based on the application debugged. Hence the visualization window at this point will look as shown in Figure. 7.4. The next step is to click the *start* icon to start the debugging session. Once, the session is started, the condition queues are dynamically.

The next step is to specify the name of the class to be debugged. This is done through the *Launch Degbuggee* panel as shown in Figure. 7.5. The Launch Debuggee panel is also used to load any run time dependency like a class or a jar file as shown in Figure. 7.6.



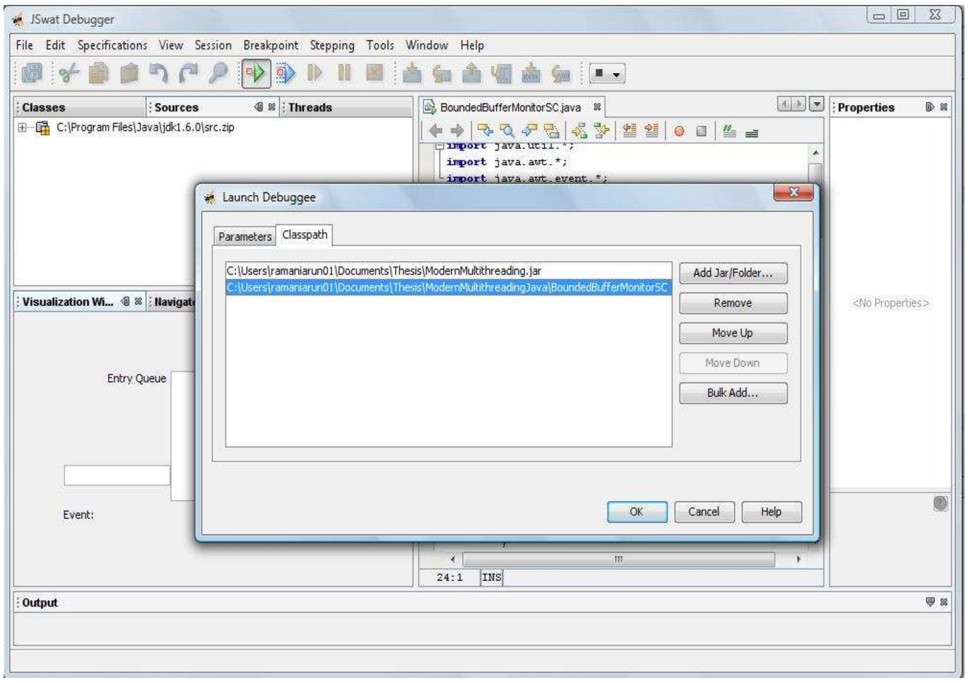
```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
```

Figure 7.4 Visualization Window



Java Runtime: java version "1.6.0"
JVM Arguments:
Class Name: BoundedBufferMonitorSC
Class Arguments:
[✓] Resume immediately

Figure 7.5 Specifying the main class name



C:\Program Files\Java\jdk1.6.0\src.zip

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
```

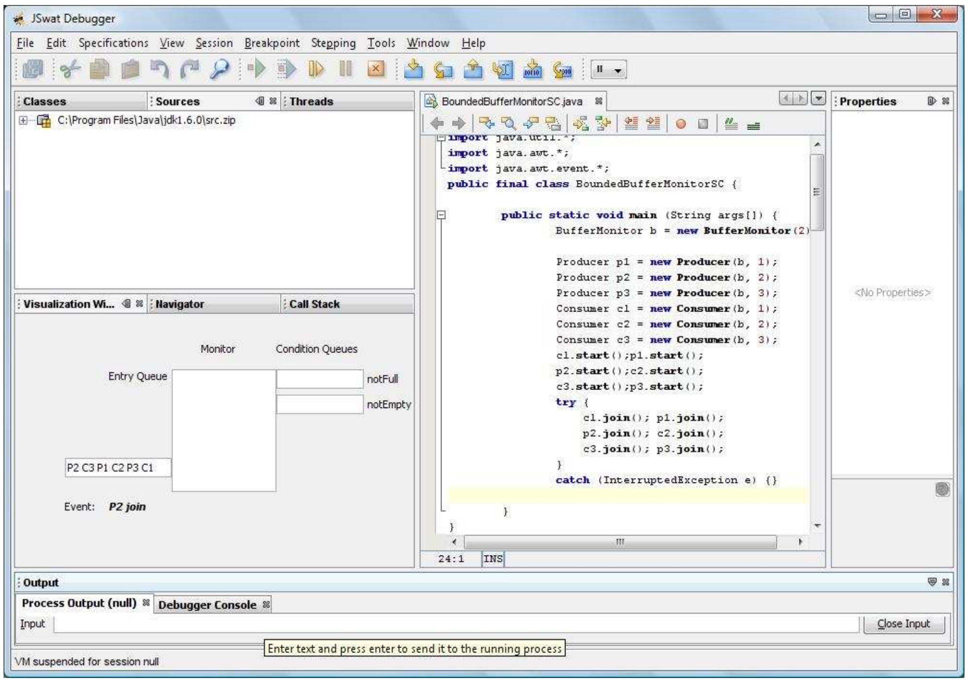
C:\Users\ramaniarun01\Documents\Thesis\ModernMultithreading.jar
C:\Users\ramaniarun01\Documents\Thesis\ModernMultithreadingJava\BoundedBufferMonitorSC

Figure 7.6 Specifying the classpath of the application

When the *OK* button is clicked, the visualization window is populated with the condition queues. All the threads executed a *join* method to get queued up in the entry queue as shown in the Figure. 7.7

Since, the buffer is empty, Consumer C1 cannot withdraw. Hence, it executes the *waitC()* method to to the *notEmpty* condition queue as shown in the Figure. 7.8.

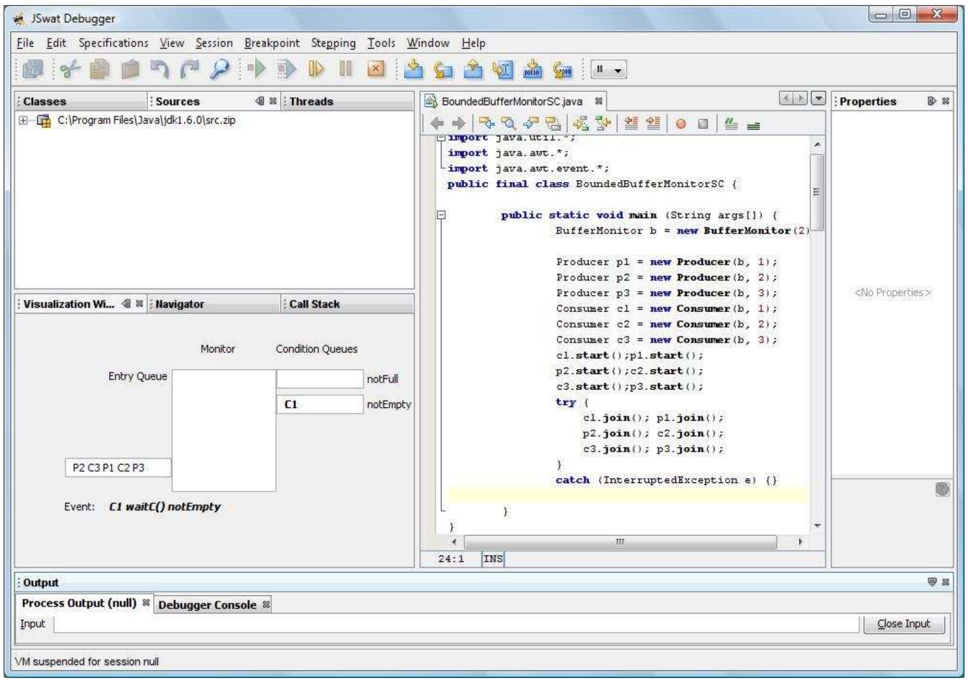
Since the producer thread *P3* is the next thread and since the buffer is empty, it gets the chance to enter the monitor to execute a *deposit()* method. Hence thread P3 enters the monitor as shown in Figure. 7.9. Upon executing a deposit() method, the thread P3 executes a *signalCall()* to signal the consumer thread C1 from the notEmpty condition queue. The thread C1 then joins the entry queue to compete with other threads to enter the monitor. The thread P3 then executes an *exitMonitor()* method to exit the monitor as shown in the Figure. 7.10.



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

VM suspended for session null

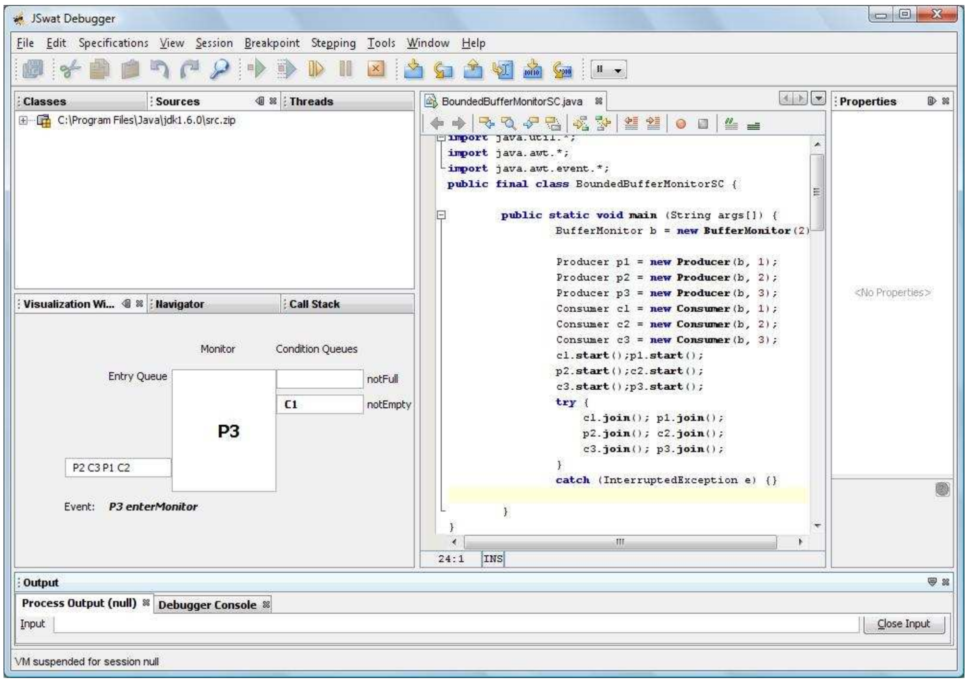
Figure 7.7 Execution of join() by all threads



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

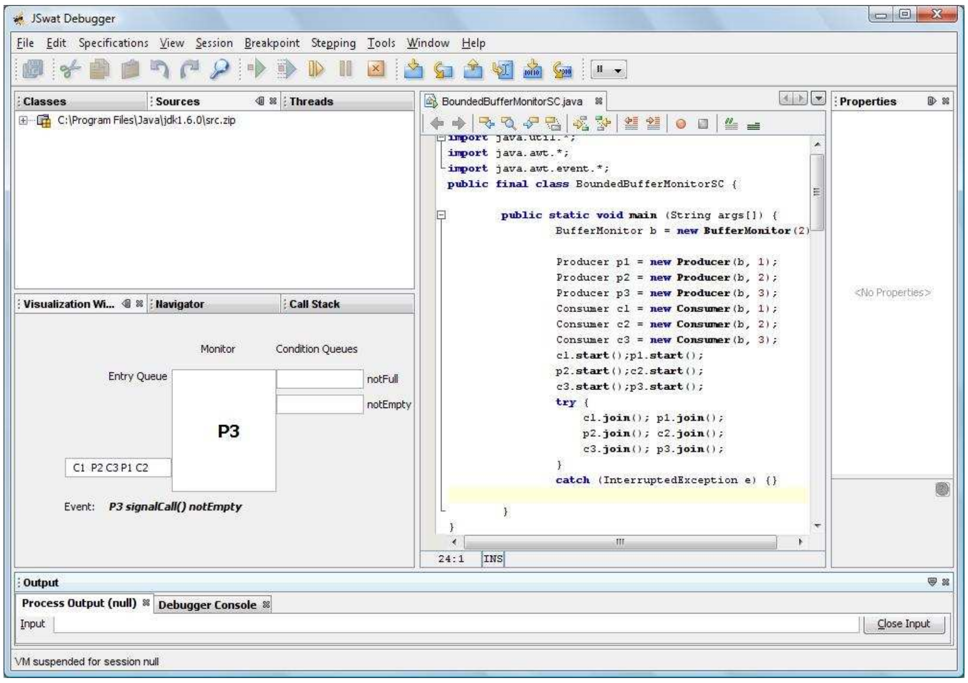
VM suspended for session null

Figure 7.8 Execution of wait() by C1



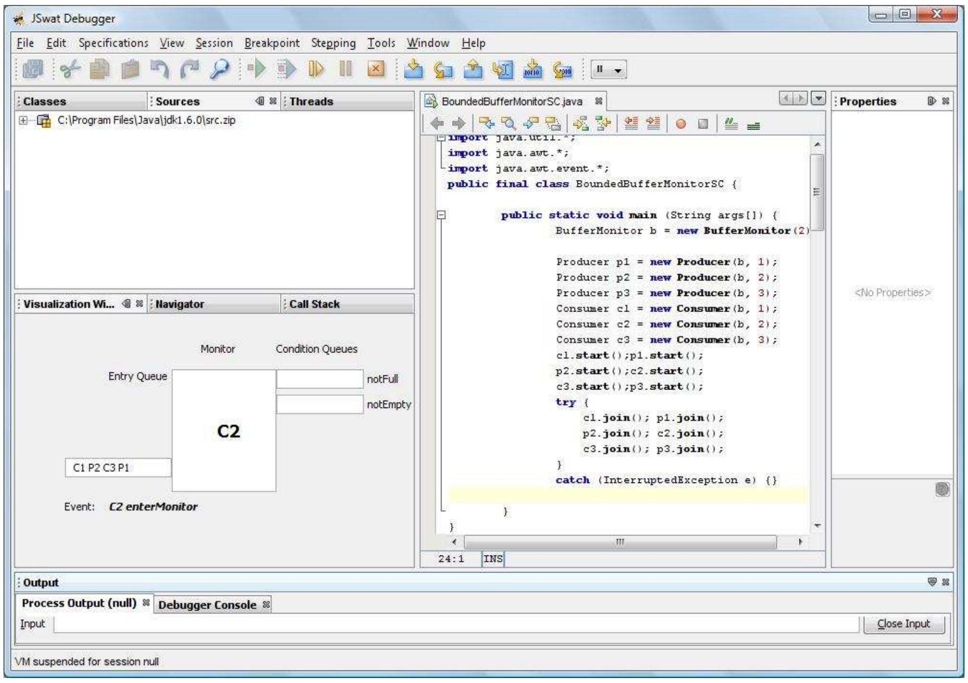
```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.9 Thread P3 enters the monitor



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.10 Execution of signalCall() and exit() by P3



```java
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

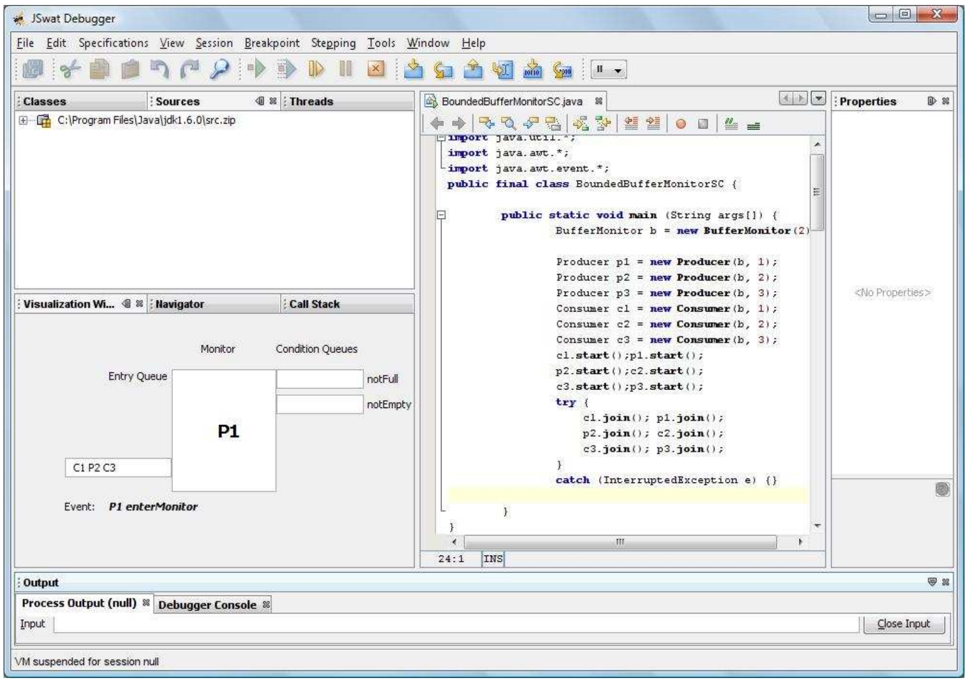
Figure 7.11 C2 enters the monitor, consumes, signals and exits the monitor

Now, the buffer is not empty. Hence the next thread C2 enters the monitor, consumes, signals and exits the monitor as shown in Figure. 7.11. Since there is no threads in the not Full condition queue, the signalCall() executed by the thread C2 does not have any effect.

The buffer is now empty. Hence the next thread P1 get the chance to enter the monitor as shown in Figure .7.12. The thread P1 then executes a signalCall() and exits the monitor.

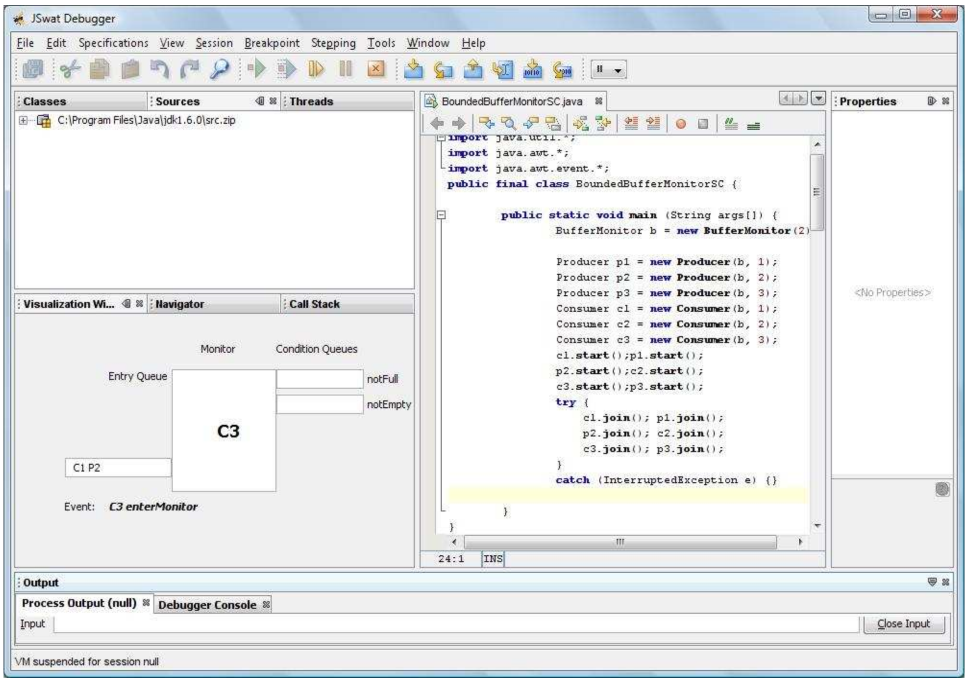
Now, the buffer is not empty. Hence the next thread C3 enters the monitor as shown in Figure. 7.13. The thread C3 then executes a signalCall() and exits the monitor.

The buffer is now empty. Hence the next thread P2 get the chance to enter the monitor as shown in Figure .7.14. The thread P1 then executes a signalCall() and exits the monitor.



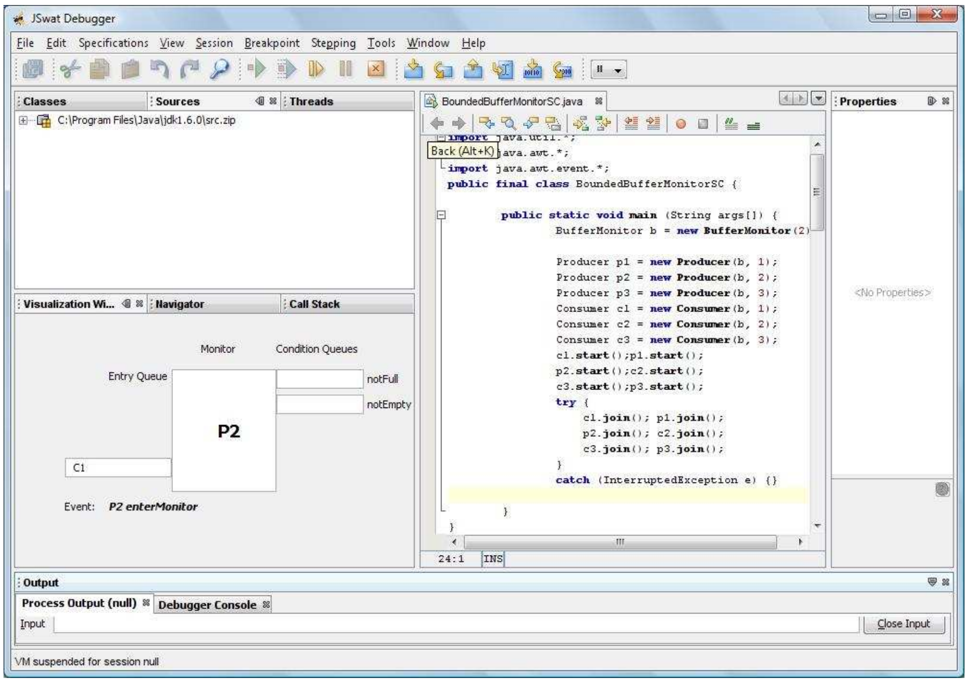
```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.12 P1 enters the monitor, deposits, signals and exits the monitor



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.13 C3 enters the monitor, consumes, signals and exits the monitor



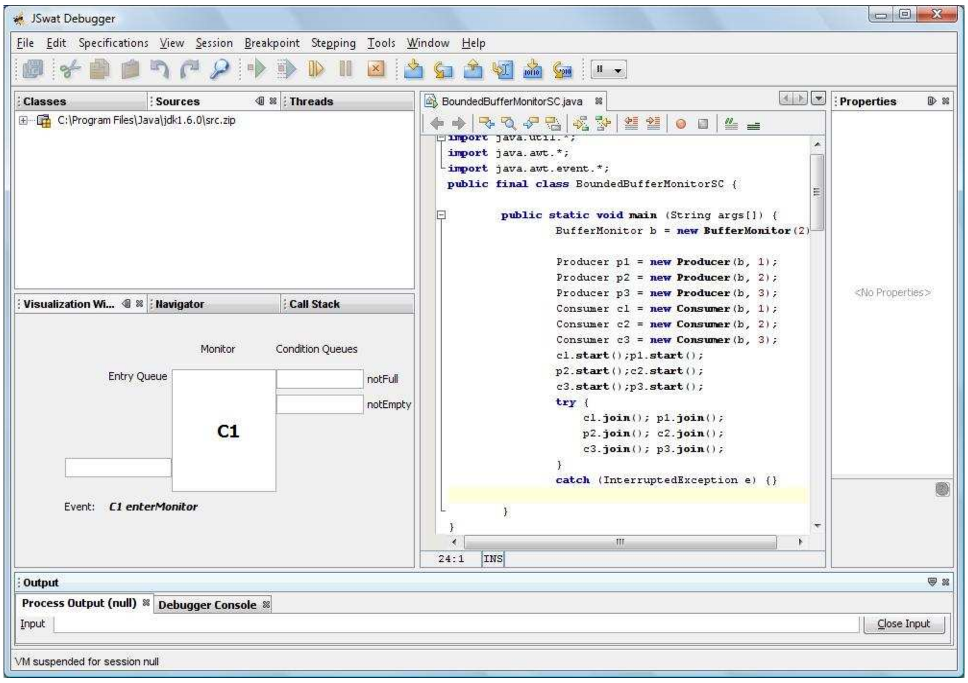
```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.14 P2 enters the monitor, deposits, signals and exits the monitor

Now the buffer is not empty. Hence the last consumer thread C1 gets a chance to enter the monitor as shown in Figure. 7.15. The thread C1 then executes a signalCall() and exits the monitor.

7.3.2 Synchronous Queue

A Synchronous Queue is a blocking queue in which each put must wait for a take, and vice versa. It does not have any internal capacity [8]. 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()*. The condition queues are inCapacity and outCapacity. In the first step, all the threads are queued in the entry queue as in Figure. 7.16.



```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public final class BoundedBufferMonitorSC {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2)

                Producer p1 = new Producer(b, 1);
                Producer p2 = new Producer(b, 2);
                Producer p3 = new Producer(b, 3);
                Consumer c1 = new Consumer(b, 1);
                Consumer c2 = new Consumer(b, 2);
                Consumer c3 = new Consumer(b, 3);
                c1.start();p1.start();
                p2.start();c2.start();
                c3.start();p3.start();
                try {
                    c1.join(); p1.join();
                    p2.join(); c2.join();
                    c3.join(); p3.join();
                }
                catch (InterruptedException e) {}

        }
}
```

Figure 7.15 C1 enters the monitor, consumes, signals and exits the monitor

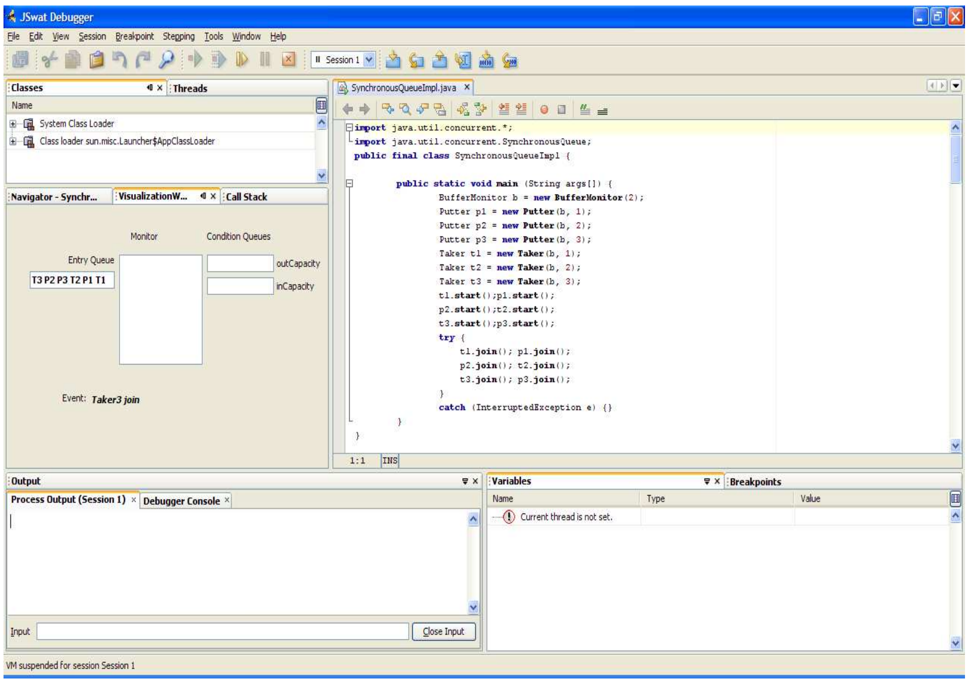
In Figure. 7.17, T3 attempts to enter the monitor but the it is not allowed to consume since there are no full slots. Therefore, it waits in incapacity condition queue.

In the next step, T2 tries to enter the monitor from the entry queue and is made to wait in the incapacity condition queue as shown in Figure. 7.18.

In the next step, T1 tries to enter the monitor from the entry queue and is made to wait in the incapacity condition queue as shown in Figure. 7.19.

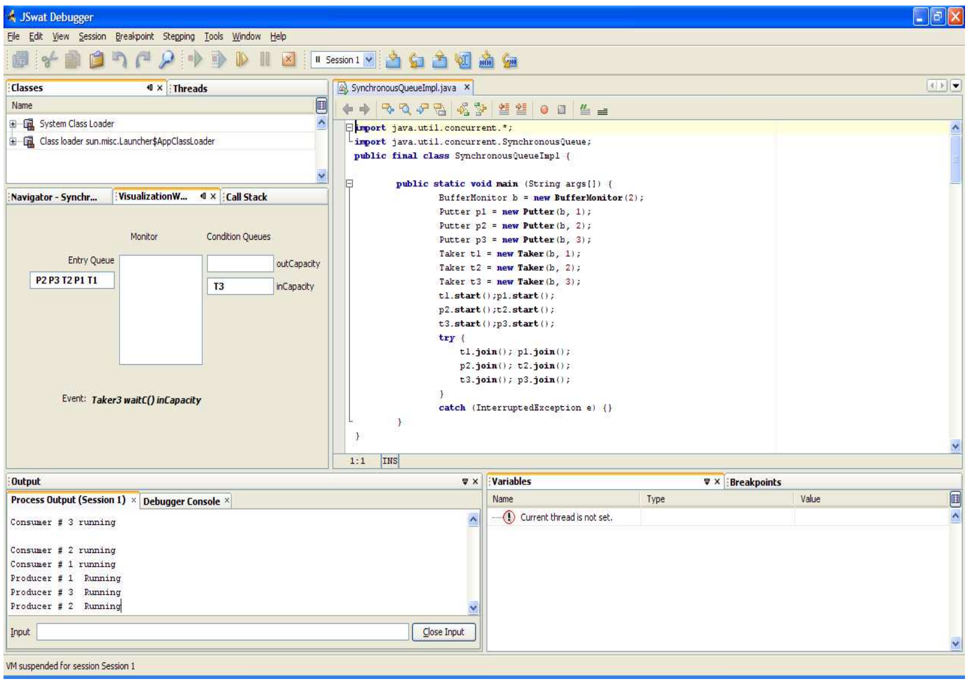
In Figure. 7.20, P3 is allowed to enter the monitor and deposit since there are empty slots in the buffer.

After depositing, P3 checks the incapacity condition queue and finds that is not empty. Hence, it signals all the threads in the condition queue. Therefore, threads T1, T2, T3 are moved to the entry queue as in Figure. 7.21. P3 then exits the monitor.



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

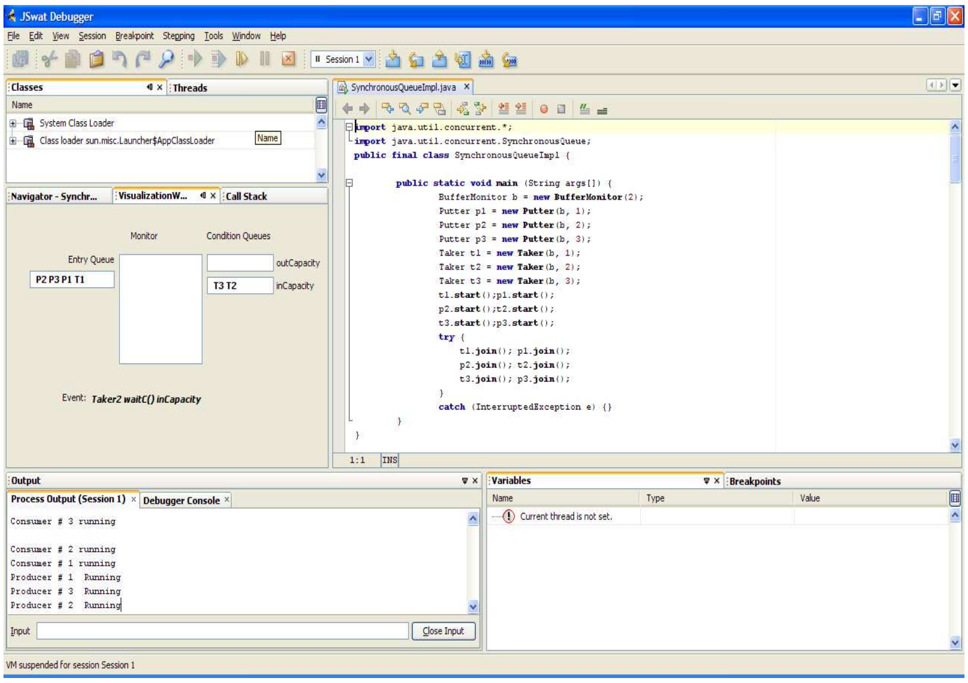
Figure 7.16 Threads waiting in entry queue



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 3 running

Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
```

Figure 7.17 T3 waits in condition queue.



Monitor

Condition Queues

Entry Queue: P2 P3 P1 T1

outCapacity:

inCapacity: T3 T2

Event: *Taker2 waitC() inCapacity*

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

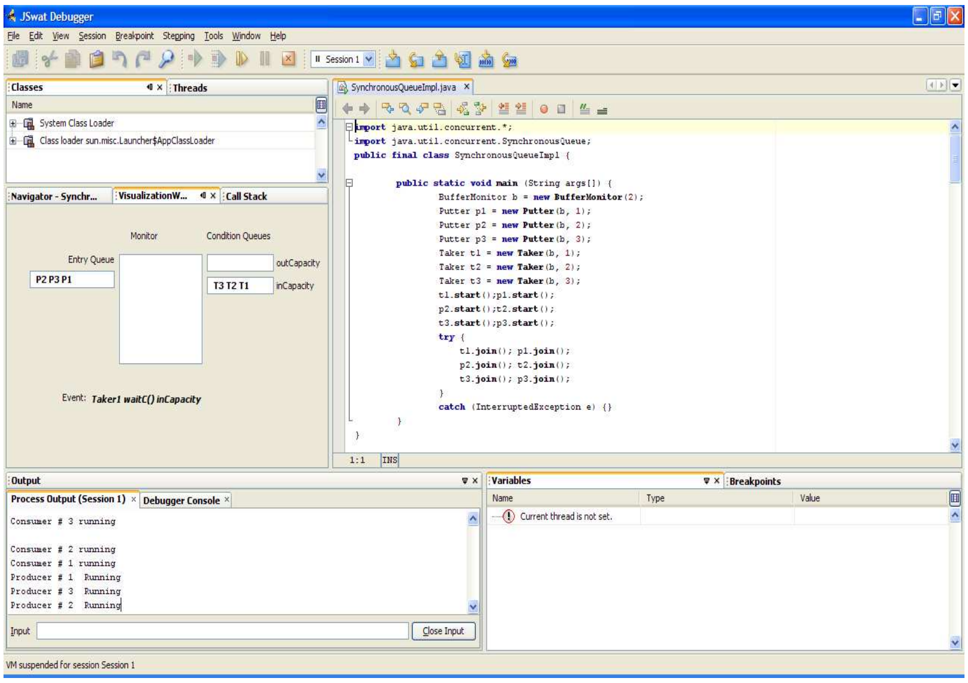
1:1 INS

```
Consumer # 3 running

Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
```

Current thread is not set.

VM suspended for session Session 1

Figure 7.18 T2 joins T3 in the condition queue.



Monitor

Condition Queues

Entry Queue: P2 P3 P1

outCapacity:

inCapacity: T3 T2 T1

Event: *Taker1 waitC() inCapacity*

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

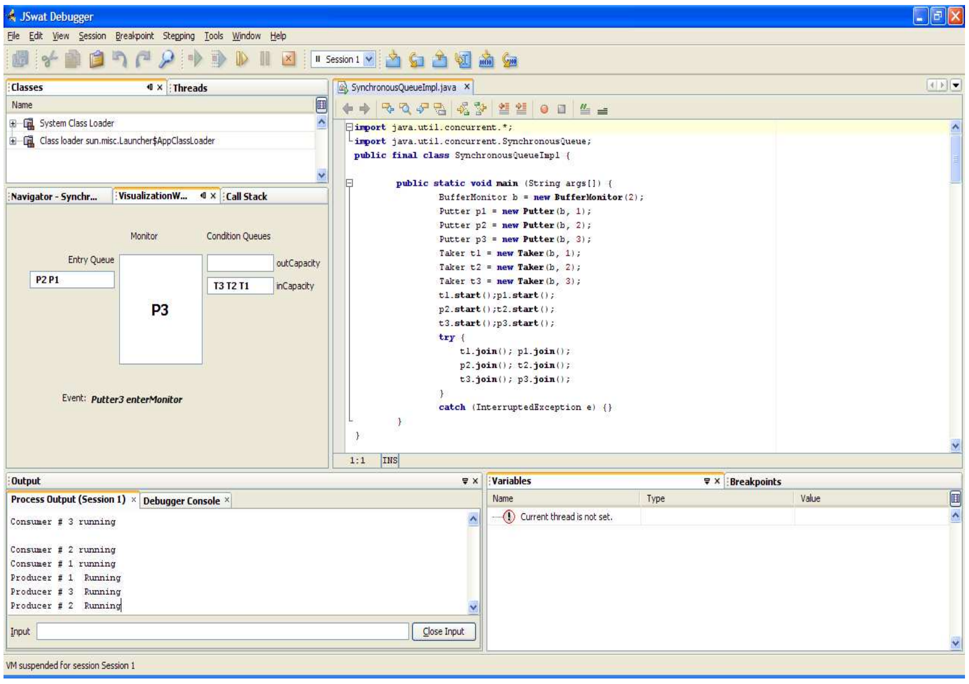
1:1 INS

```
Consumer # 3 running

Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
```

Current thread is not set.

VM suspended for session Session 1

Figure 7.19 T1 waits in the condition queue with T2 and T3.



Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader

Monitor | Condition Queues
Entry Queue: P2 P1
Monitor: P3
outCapacity:
inCapacity: T3 T2 T1
Event: *Putter3 enterMonitor*

```java
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

1:1 INS

Process Output (Session 1) | Debugger Console

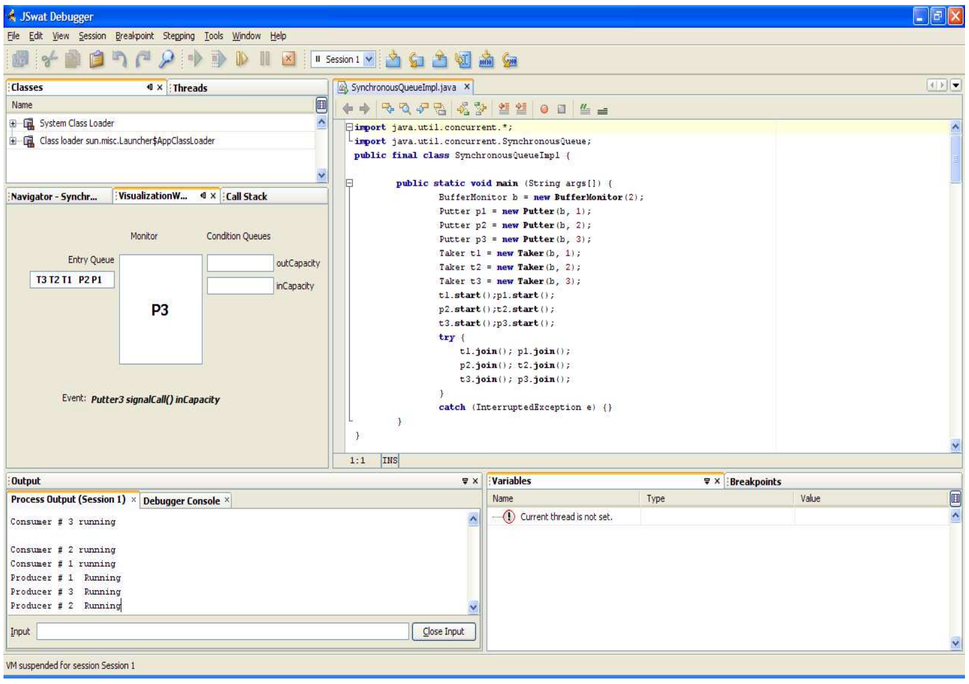
```
Consumer # 3 running

Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
```

Input

Name | Type | Value
Current thread is not set.

VM suspended for session Session 1

Figure 7.20 P3 enters the monitor.



Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader

Monitor | Condition Queues
Entry Queue: T3 T2 T1 P2 P1
Monitor: P3
outCapacity:
inCapacity:
Event: *Putter3 signalCall() inCapacity*

```java
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

1:1 INS

Process Output (Session 1) | Debugger Console

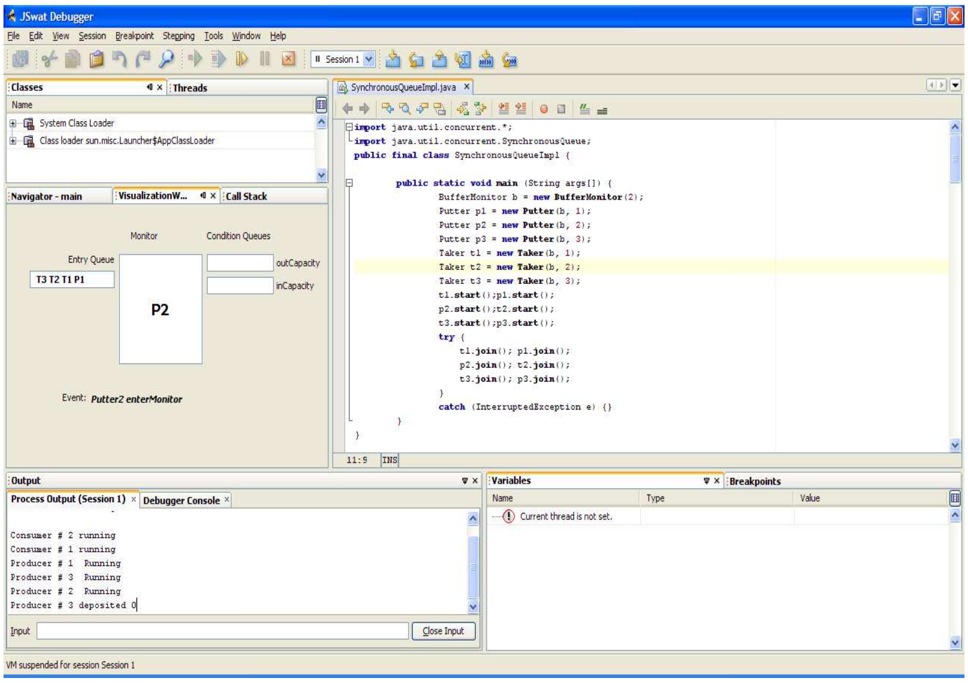
```
Consumer # 3 running

Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
```

Input

Name | Type | Value
Current thread is not set.

VM suspended for session Session 1

Figure 7.21 P3 awakens the threads from the condition queue.



Event: *Putter2 enterMonitor*

```java
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
```

Current thread is not set.

Figure 7.22 P2 enters the monitor.

In Figure. 7.22, it is shown that P2 is the next thread to enter the monitor.

After depositing, P2 checks the condition queue and since it is empty, it exits the monitor, as shown in Figure. 7.23.

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 condition queue, as in Figure. 7.24.

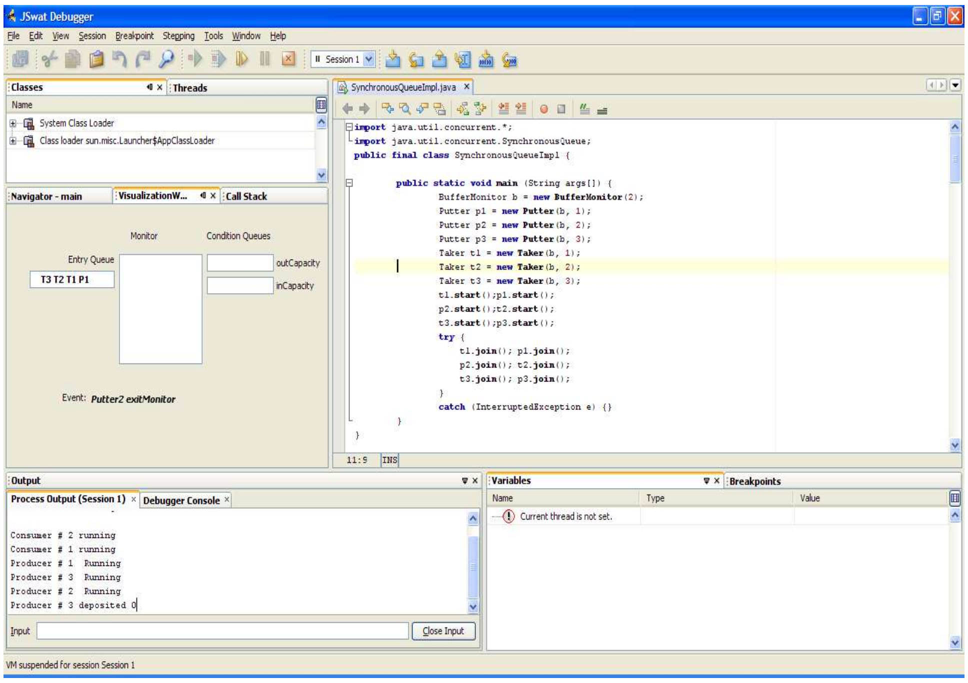
The next thread T3 enters the monitor in Figure. 7.25.

After consuming, T3 signals the condition queue in Figure. 7.26.

T3 then proceeds to exit the monitor in Figure. 7.27.

In Figure. 7.28, T1 enters the monitor from the entry queue.

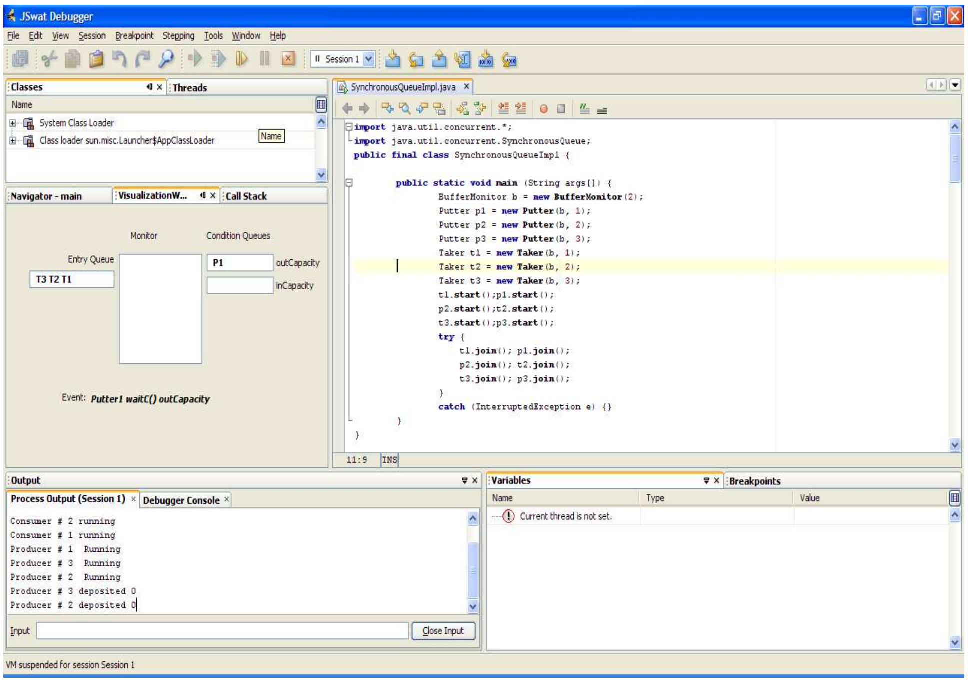
After consuming, T1 signals a thread in the condition queue and exits the monitor, refer Fig. 7.29.



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
```

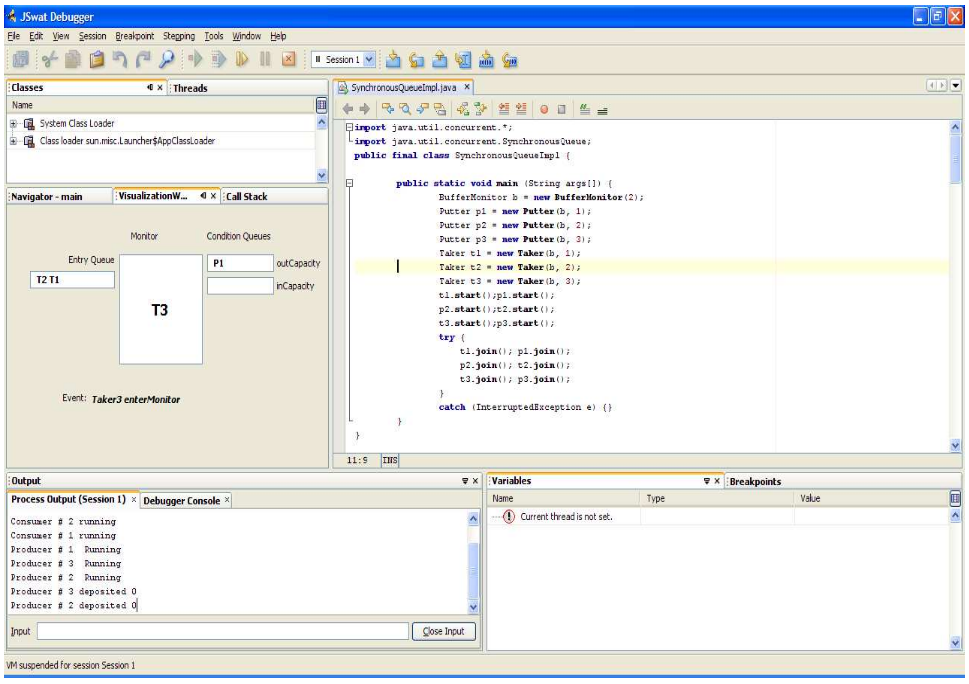
Figure 7.23 P2 exits the monitor.


JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader
Name
Navigator - main
VisualizationW...
Call Stack
Monitor
Condition Queues
Entry Queue
T3 T2 T1
P1
outCapacity
inCapacity
Event: Putter1 waitC() outCapacity
SynchronousQueueImpl.java
11:9
INS
Output
Process Output (Session 1)
Debugger Console
Input
Close Input
Variables
Breakpoints
Name
Type
Value
Current thread is not set.
VM suspended for session Session 1

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
```

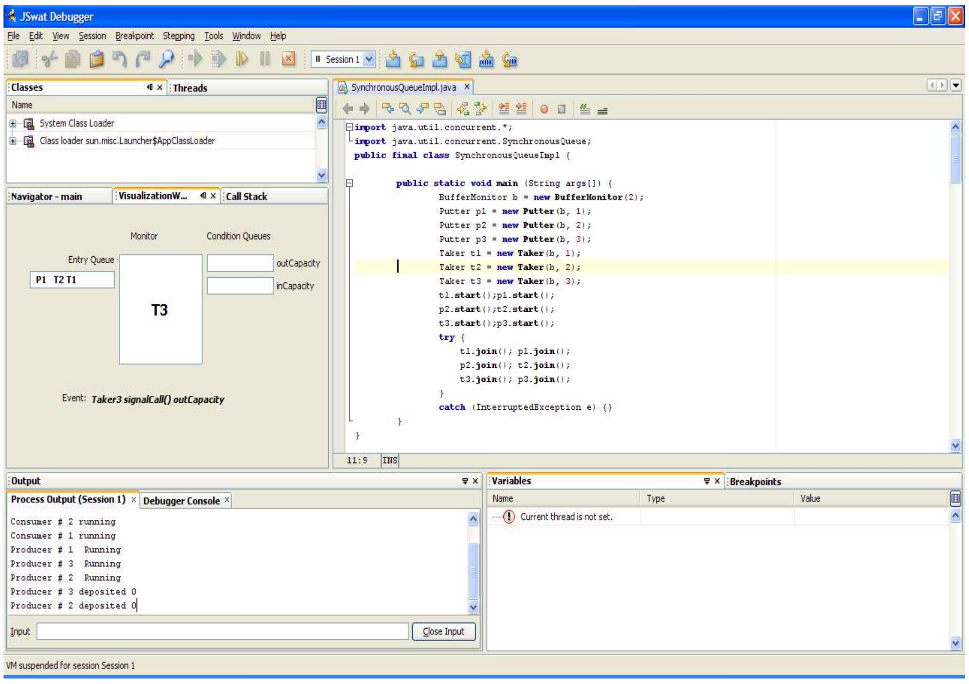
Figure 7.24 P1 waiting in condition queue.



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
```

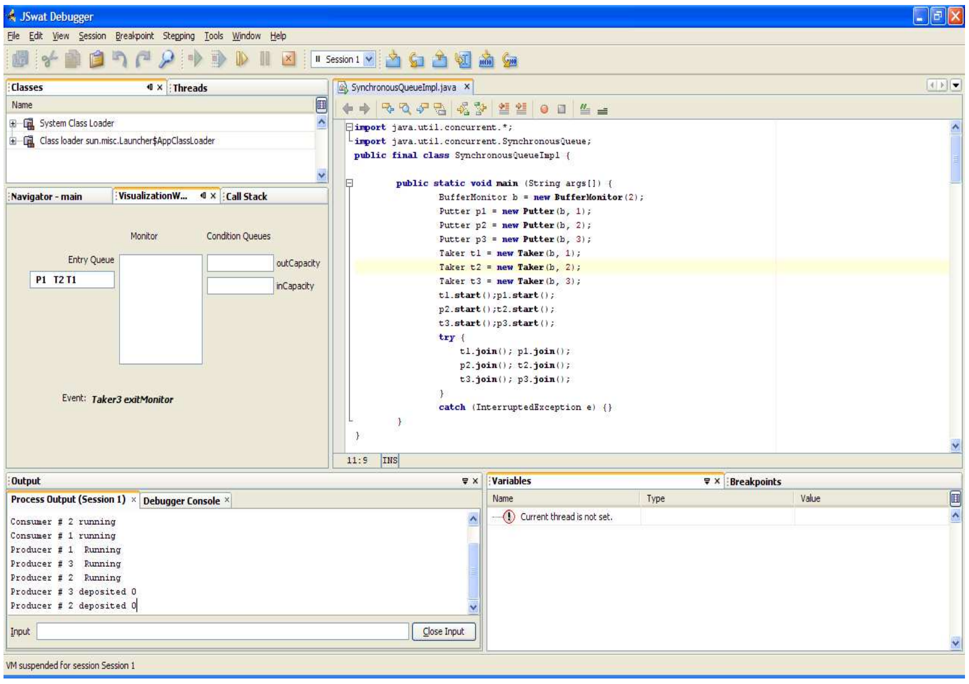
Figure 7.25 T3 enters the monitor.


JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader
Navigator - main
VisualizationW...
Call Stack
Monitor
Condition Queues
Entry Queue
P1 T2 T1
T3
outCapacity
inCapacity
Event: Taker3 signalCall() outCapacity
SynchronousQueueImpl.java
11:9
INS
Output
Process Output (Session 1)
Debugger Console
Input
Close Input
Variables
Breakpoints
Name
Type
Value
Current thread is not set.
VM suspended for session Session 1

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
```

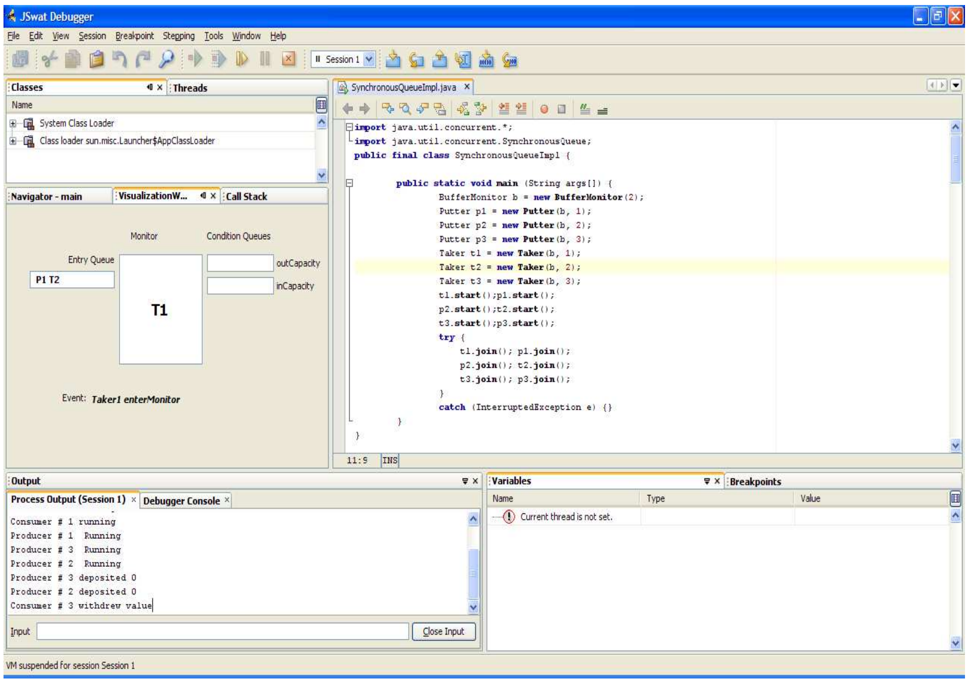
Figure 7.26 P1 awakened from the condition queue.



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

    public static void main (String args[]) {
        BufferMonitor b = new BufferMonitor(2);
        Putter p1 = new Putter(b, 1);
        Putter p2 = new Putter(b, 2);
        Putter p3 = new Putter(b, 3);
        Taker t1 = new Taker(b, 1);
        Taker t2 = new Taker(b, 2);
        Taker t3 = new Taker(b, 3);
        t1.start();p1.start();
        p2.start();t2.start();
        t3.start();p3.start();
        try {
            t1.join(); p1.join();
            p2.join(); t2.join();
            t3.join(); p3.join();
        }
        catch (InterruptedException e) {}
    }
}
```

```
Consumer # 2 running
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
```

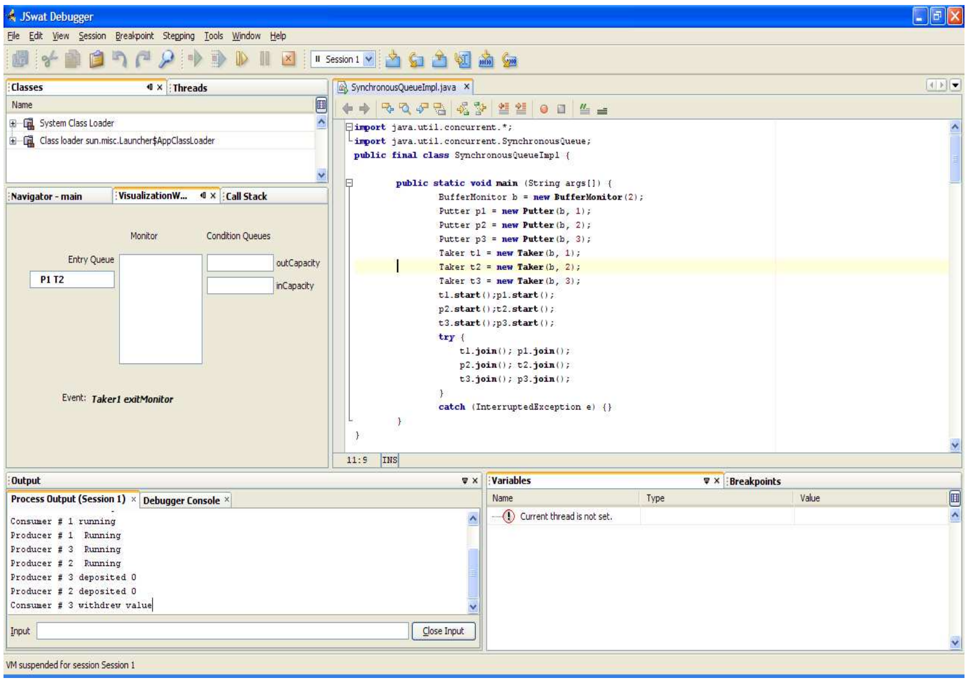
Figure 7.27 T3 exits the monitor.



```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

    public static void main (String args[]) {
        BufferMonitor b = new BufferMonitor(2);
        Putter p1 = new Putter(b, 1);
        Putter p2 = new Putter(b, 2);
        Putter p3 = new Putter(b, 3);
        Taker t1 = new Taker(b, 1);
        Taker t2 = new Taker(b, 2);
        Taker t3 = new Taker(b, 3);
        t1.start();p1.start();
        p2.start();t2.start();
        t3.start();p3.start();
        try {
            t1.join(); p1.join();
            p2.join(); t2.join();
            t3.join(); p3.join();
        }
        catch (InterruptedException e) {}
    }
}
```

```
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
```

Figure 7.28 T1 enters the monitor.


JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader
Navigator - main
VisualizationW...
Call Stack
Monitor
Condition Queues
Entry Queue
P1 T2
outCapacity
inCapacity
Event: Taker1 exitMonitor
SynchronousQueueImpl.java
11:9
INS
Output
Process Output (Session 1)
Debugger Console
Input
Close Input
Variables
Breakpoints
Name
Type
Value
Current thread is not set.
VM suspended for session Session 1

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Consumer # 1 running
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
```

Figure 7.29 T1 exits the monitor.

In Figure. 7.30, T2 tries to enter the monitor but since all the slots are empty, it has to wait in the outCapacity condition queue.

P1 then enter the monitor and deposits in Figure. 7.31.

After depositing P1 signals the condition queue and exits in Figure. 7.32.

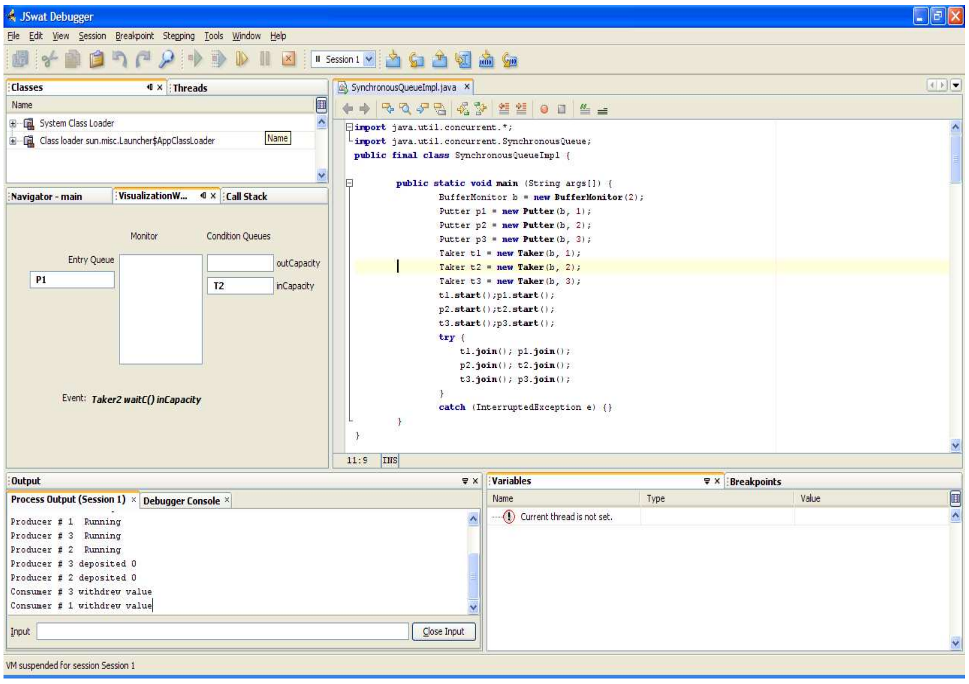
In Figure. 7.33, P1 exits the monitor.

T2 then enter the monitor and consumes in Figure. 7.34.

T2 signals the condition queue and exits in Figure. 7.35. This example showcases the visualization process for a bounded buffer problem implemented using a synchronous queue.

7.3.3 Readers - Writers problem

Let us consider a Readers - Writers problem. Readers - Writers problem is a classic problem involving a list of Reader threads trying to read from a buffer and a list of Writer threads trying to write to the buffer. In this experiment, let us visualize these reader and writer threads. There are three reader threads and

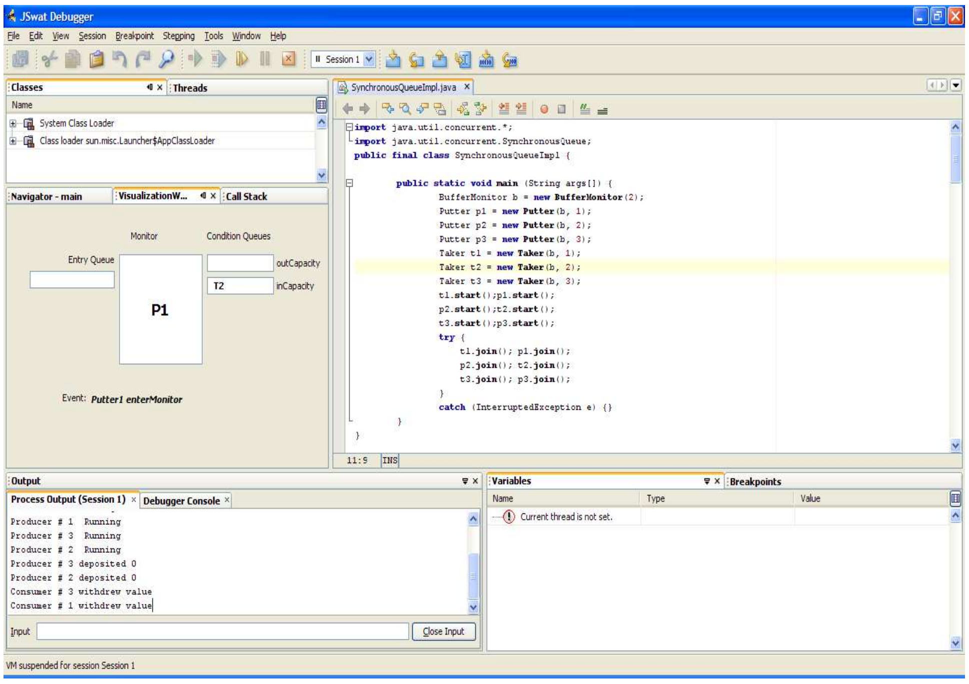

JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader
Navigator - main
VisualizationW...
Call Stack
Monitor
Condition Queues
Entry Queue
P1
outCapacity
T2
inCapacity
Event: Taker2 waitC() inCapacity
SynchronousQueueImpl.java
Output
Process Output (Session 1)
Debugger Console
Input
Close Input
Variables
Breakpoints
Type
Value
Current thread is not set.
VM suspended for session Session 1

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
11:9  INS
```

```
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
```

Figure 7.30 T2 waits in the condition queue.

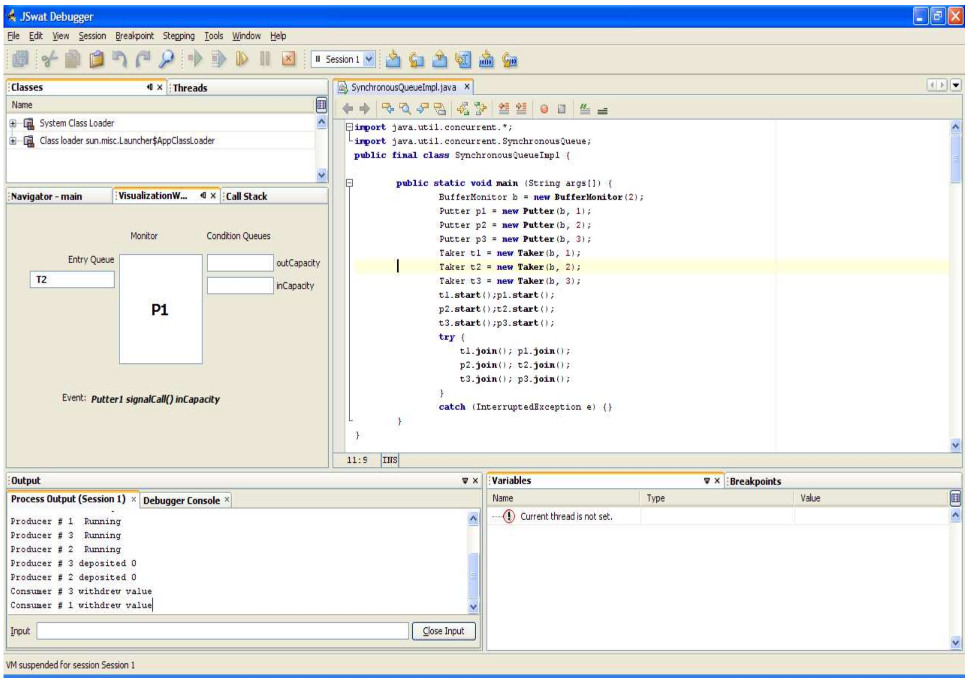

JSwat Debugger
File Edit View Session Breakpoint Stepping Tools Window Help
Session 1
Classes
Threads
Name
System Class Loader
Class loader sun.misc.Launcher$AppClassLoader
Navigator - main
VisualizationW...
Call Stack
Monitor
Condition Queues
Entry Queue
P1
outCapacity
T2
inCapacity
Event: Putter1 enterMonitor
SynchronousQueueImpl.java
Output
Process Output (Session 1)
Debugger Console
Input
Close Input
Variables
Breakpoints
Type
Value
Current thread is not set.
VM suspended for session Session 1

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
11:9  INS
```

```
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
```

Figure 7.31 P1 enters the monitor.

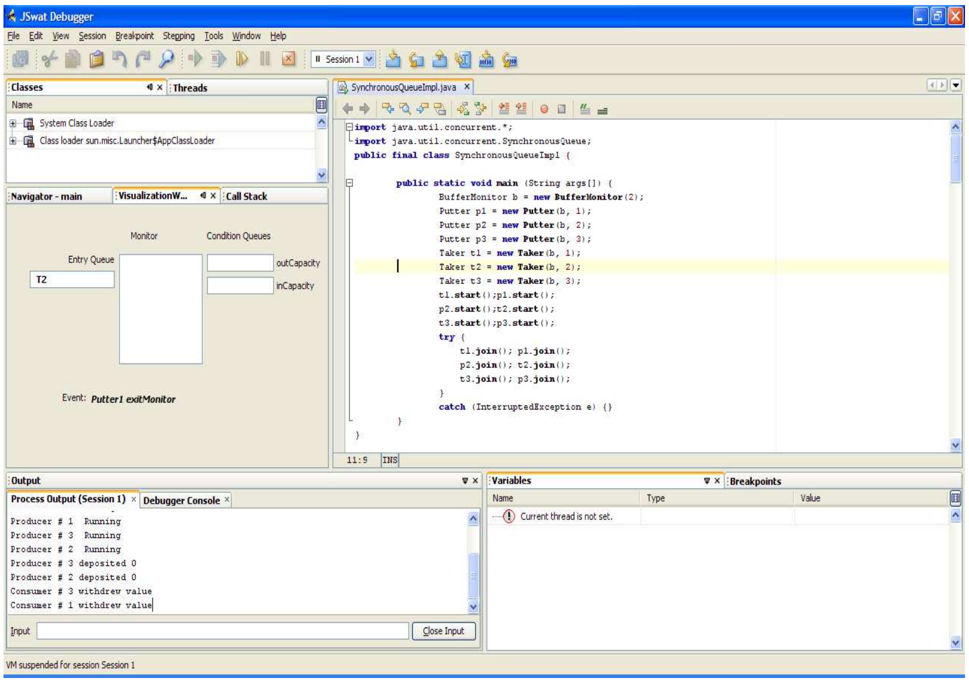


Event: *Putter1 signalCall() inCapacity*

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
```

Figure 7.32 T2 awakened by P1 and enters the entry queue again.

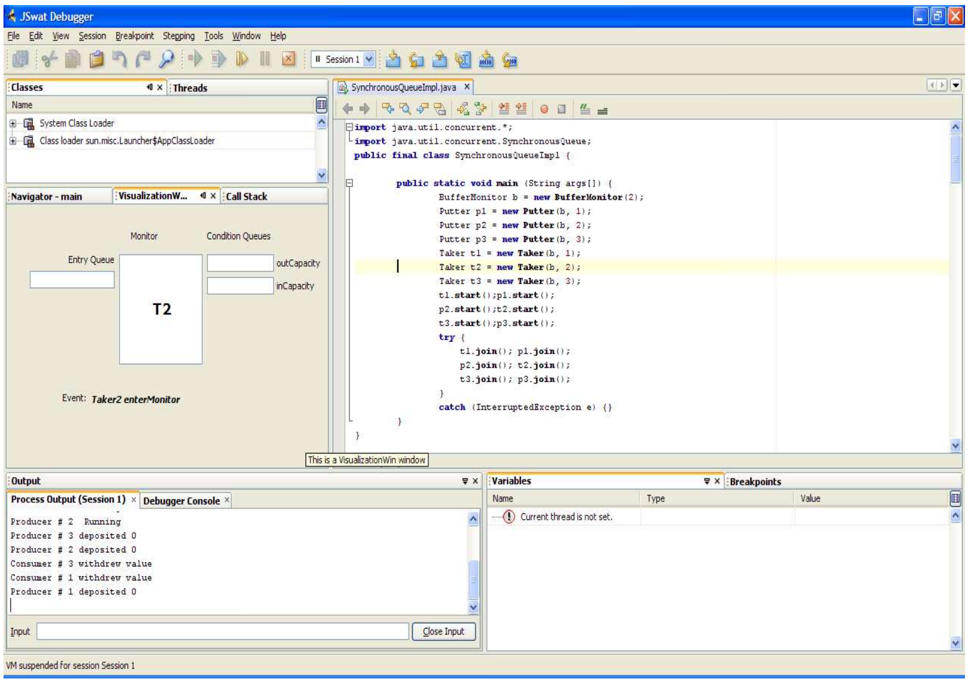


Event: *Putter1 exitMonitor*

```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Producer # 1  Running
Producer # 3  Running
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
```

Figure 7.33 P1 exits the monitor.



Event: *Taker2 enterMonitor*

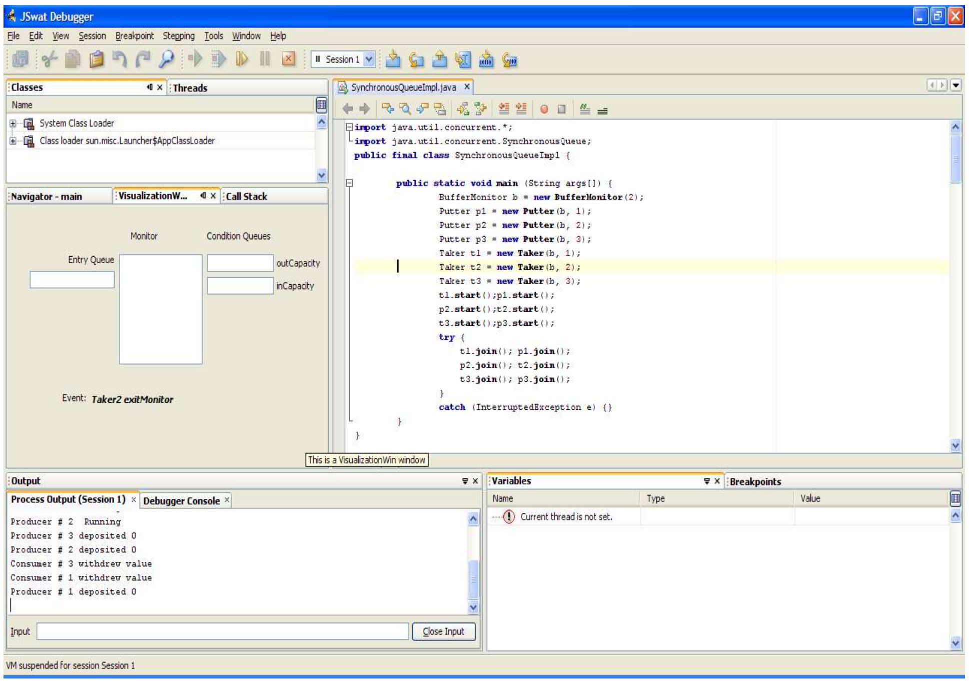
```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
Producer # 1 deposited 0
```

Current thread is not set.

VM suspended for session Session 1

Figure 7.34 T2 enters the monitor.



Event: *Taker2 exitMonitor*

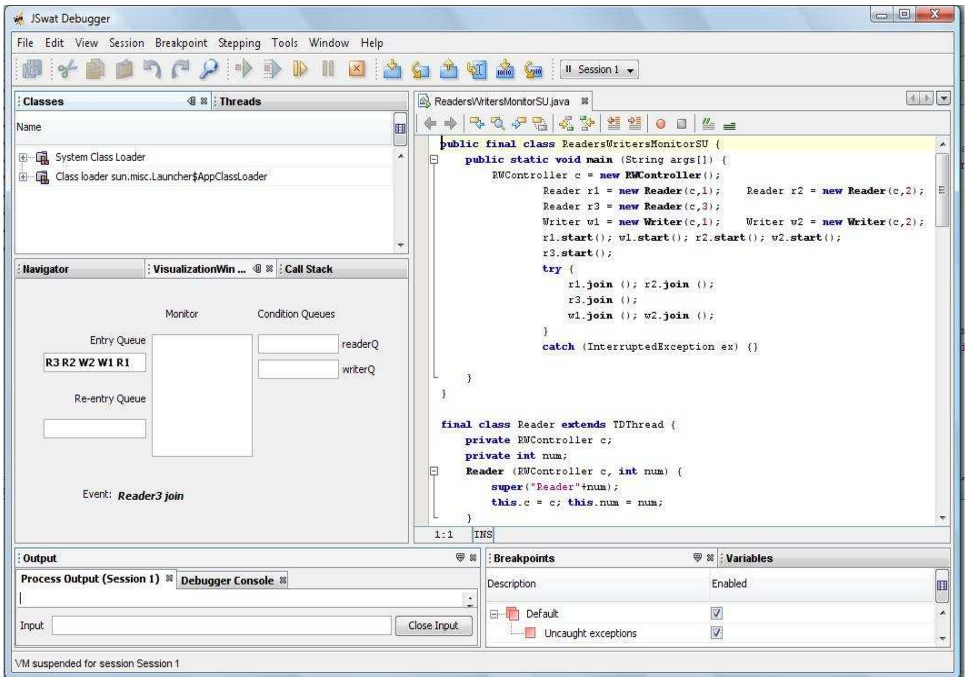
```
import java.util.concurrent.*;
import java.util.concurrent.SynchronousQueue;
public final class SynchronousQueueImpl {

        public static void main (String args[]) {
                BufferMonitor b = new BufferMonitor(2);
                Putter p1 = new Putter(b, 1);
                Putter p2 = new Putter(b, 2);
                Putter p3 = new Putter(b, 3);
                Taker t1 = new Taker(b, 1);
                Taker t2 = new Taker(b, 2);
                Taker t3 = new Taker(b, 3);
                t1.start();p1.start();
                p2.start();t2.start();
                t3.start();p3.start();
                try {
                    t1.join(); p1.join();
                    p2.join(); t2.join();
                    t3.join(); p3.join();
                }
                catch (InterruptedException e) {}
        }
}
```

```
Producer # 2  Running
Producer # 3 deposited 0
Producer # 2 deposited 0
Consumer # 3 withdrew value
Consumer # 1 withdrew value
Producer # 1 deposited 0
```

Current thread is not set.

VM suspended for session Session 1

Figure 7.35 T2 exits the monitor.



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

```
1:1   INS
```

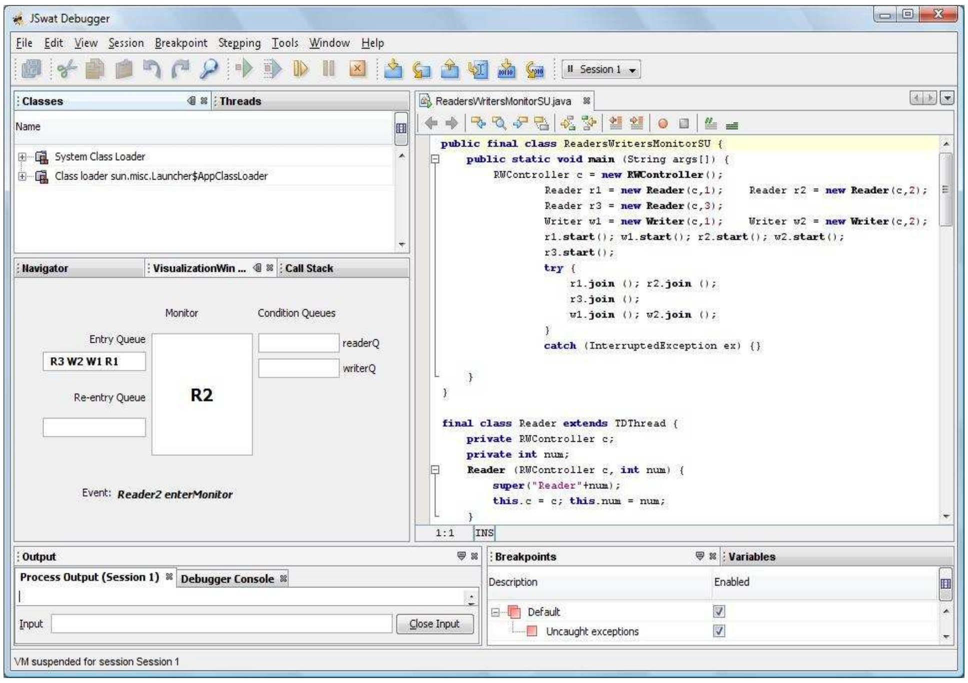
Figure 7.36 Example Readers - Writers problem with SU Signaling discipline

two writer threads. These threads execute *join* method to get queued up in the entry queue as shown in the Fig. 7.36. The presence of the *re-entry* queue in Fig. 7.36 is to be noted. This is because, we adopt Signal and Urgent wait signalling discipline for this experiment to solve the Readers - Writers problem.

Readers - Writers problem is different from Producer - Consumer problem because, in the Readers - Writers problem, the actual reading and writing happens outside the monitor as oppossed to Producer - Consumer problem where the actual deposit and withdraw happens inside the monitor. Hence in this experiment, the reader and the writer threads, enter monitor to get to read or write respectively while the actual reading and writing happens outside the monitor.

Now,since there is no writer in the monitor, the reader thread *R2* gets the chance to enter the monito as shown in the Fig. 7.37.

As mentioned above, the reader thread *R2* executes *Signal and exit*to exit the monitor after entering the monitor as shown in the Fig. 7.38. The Signal and



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

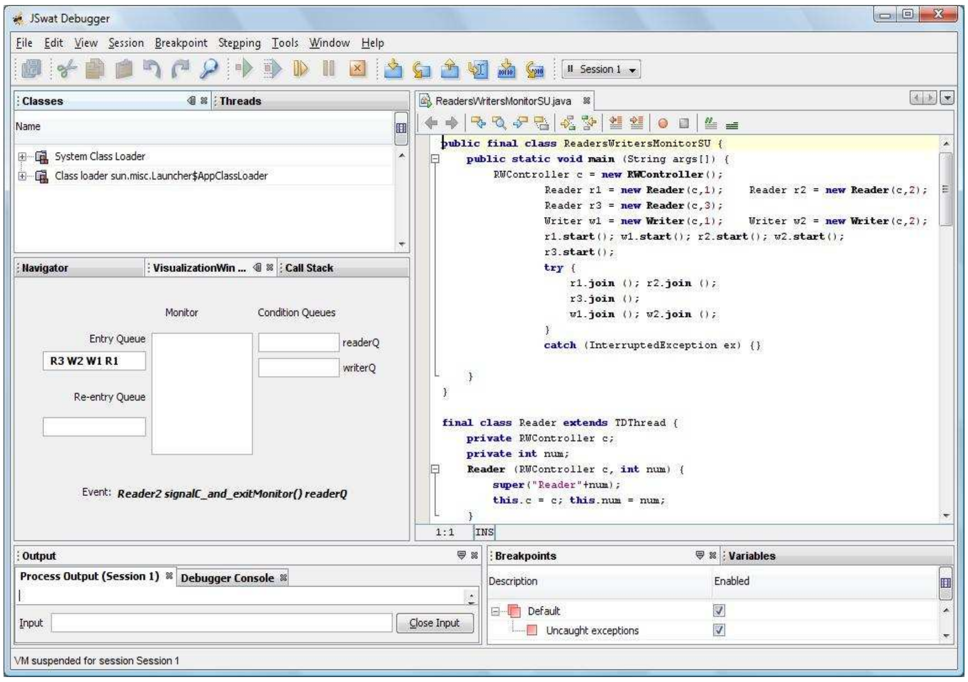
Figure 7.37 R2 enters the monitor

exit monitor signal call does not signal any writer thread as there is no writer threads waiting in the writer queue at this point.

After the Signal and exit monitor signal, the reader thread *R2* exits the monitor to start reading. Now, since multiple readers are permitted to read at the same time, the next thread *R3* gets the chance to enter monitor as shown in Fig. 7.39. While the thread R3 enter the monitor, thread R2 would have been reading. Then, The thread executes *Signal and exit monitor* to exit the monitor as shown in the Fig. 7.39.and Fig. 7.40.

Similarly, the reader thread R1 joins the monitor by execute the enterMonitor() as shown in the Fig. 7.41. The thread R1 then executes *Signal and exit monitor* call and exits the monitor

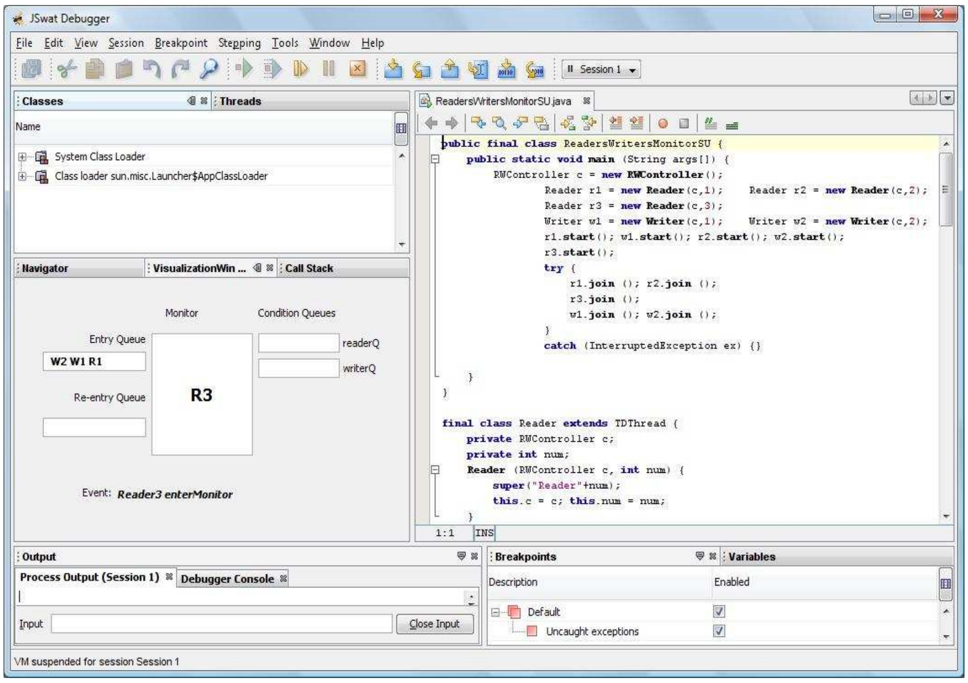
Now, at this point, all the three readers are busy reading. Then, the writer thread *W2* waits in the condition till it is signalled off. This is shown in the Fig. 7.42



Event: *Reader2 signalC_and_exitMonitor() readerQ*

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

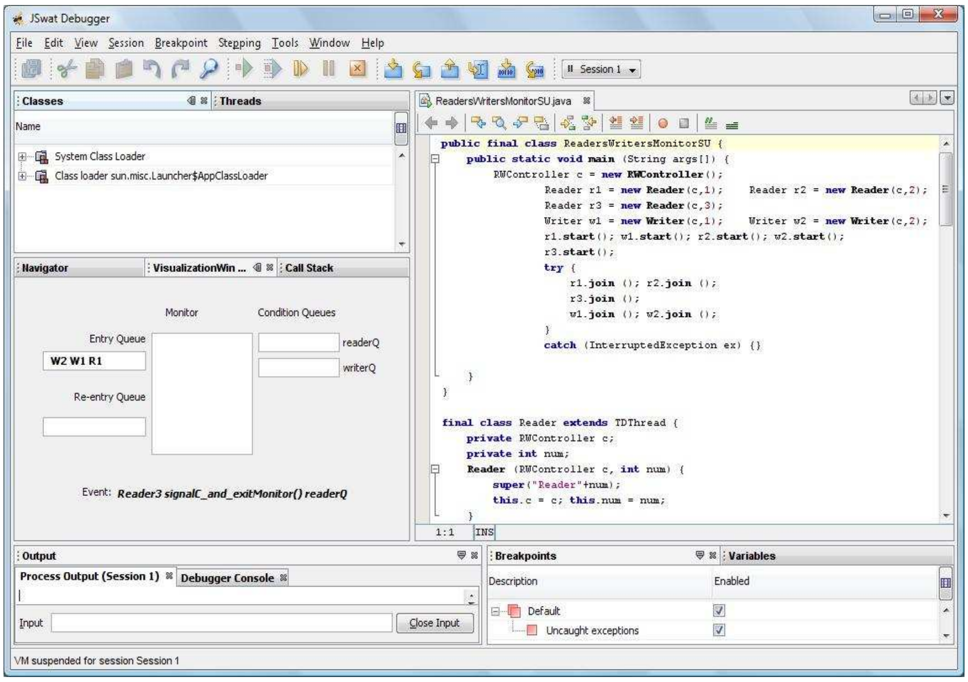
Figure 7.38 R2 exits the monitor



Event: *Reader3 enterMonitor*

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.39 R3 enters the monitor

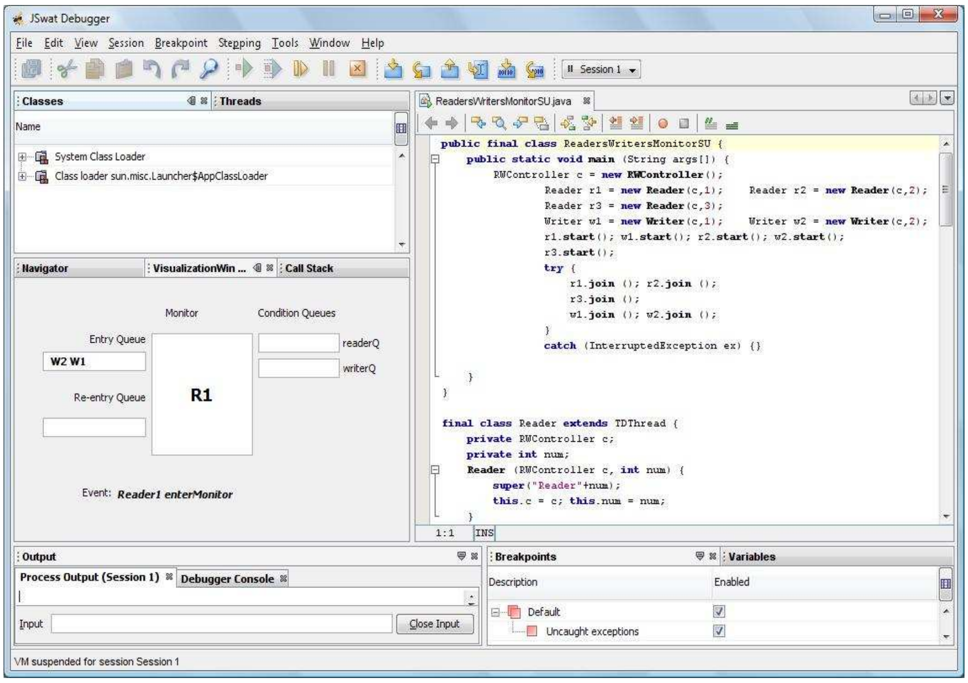


Event: *Reader3 signalC_and_exitMonitor() readerQ*

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

VM suspended for session Session 1

Figure 7.40 R3 enters the monitor

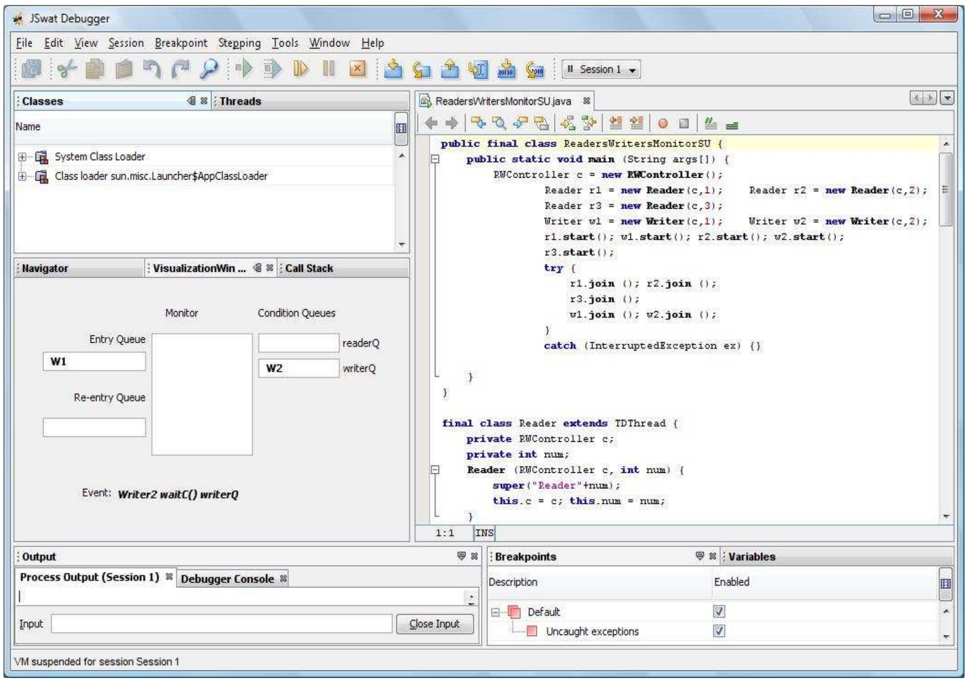


Event: *Reader1 enterMonitor*

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

VM suspended for session Session 1

Figure 7.41 R1 enters the monitor



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

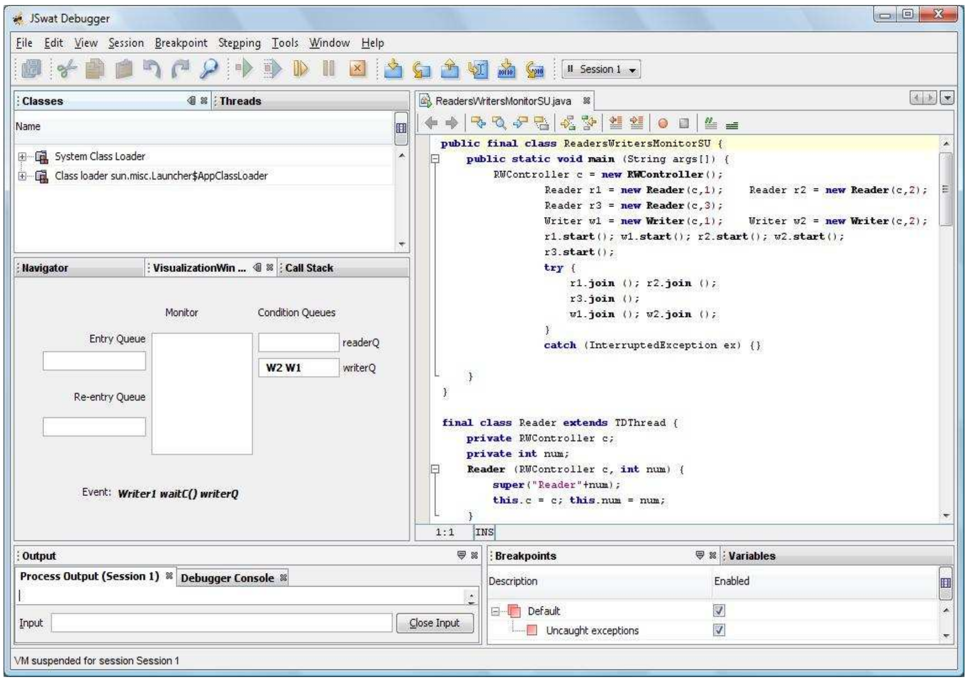
Figure 7.42 W2 waits in the condition queue Writer Q

Now the thread *W1* tries to enter the monitor but since the readers are still reading, *W1* goes to the writersQ. This is shown in the Fig. 7.43.

Now, these threads in the writer queue would be signalled by the readers while exiting the monitor. Now, the thread *R2* enters the monitor to exit as shown in the Fig. 7.44.

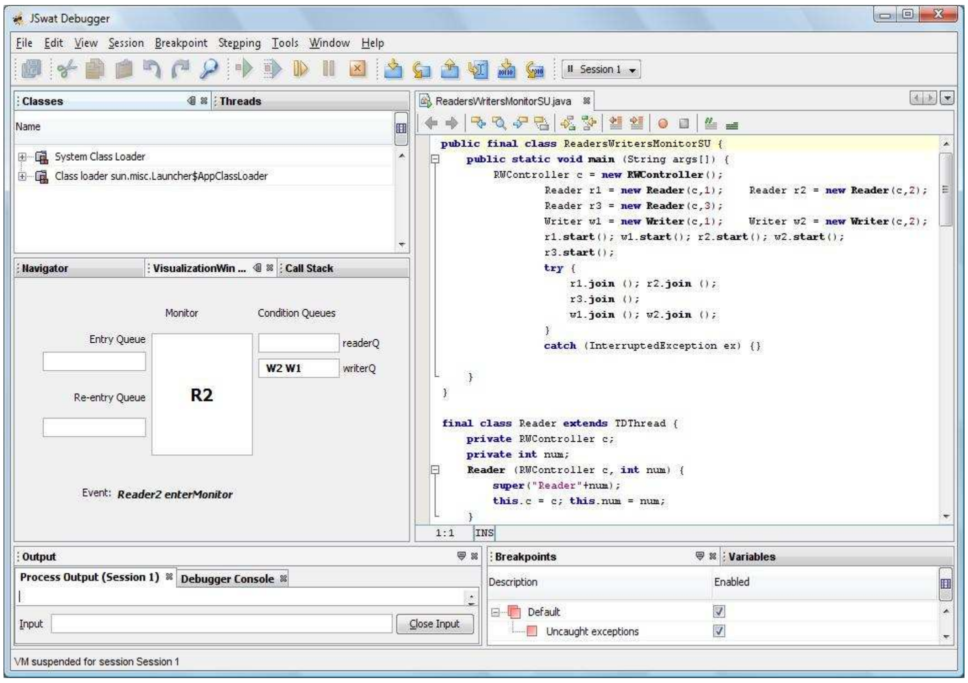
Since, there are two more readers waiting to exit the monitor after reading and since readers have more priority than writers, the thread *R2* does not execute a SignalC and Urgent wait signal. Instead, it executes an exit monitor call to exit and let the other reader threads to enter the monitor. This is shown in the Fig. 7.45.

Similarly, reader 2 executes and exit Monitor method as there is one more reader availble. Hence the thread *R1* then enters the monitor as shown in Fig. 7.46.



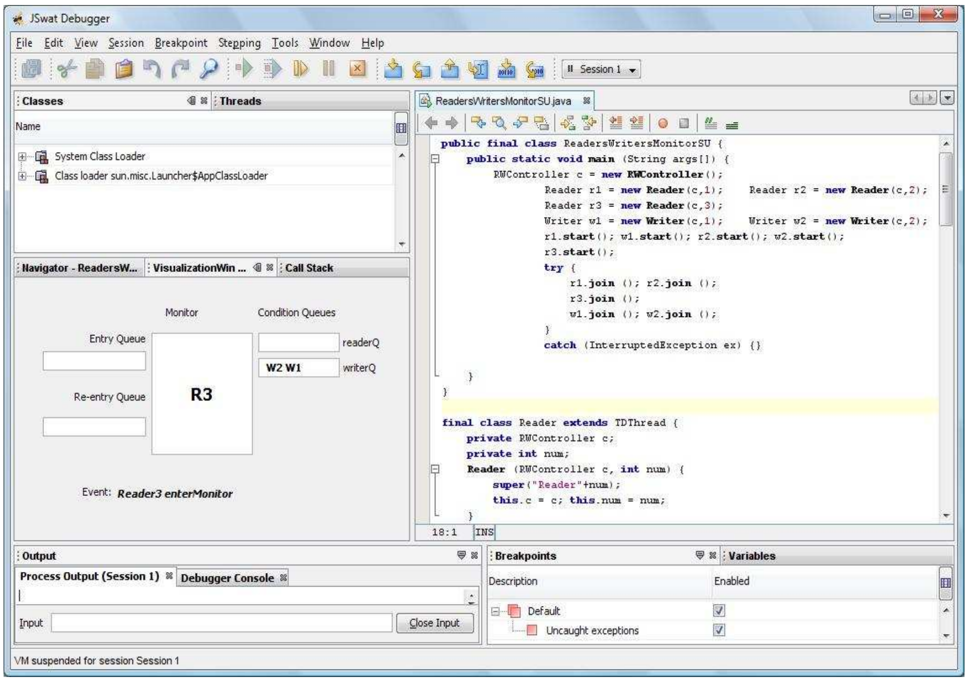
```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.43 W1 waits in the condition queue Writer Q



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.44 R2 enters the monitor to indicate its completion



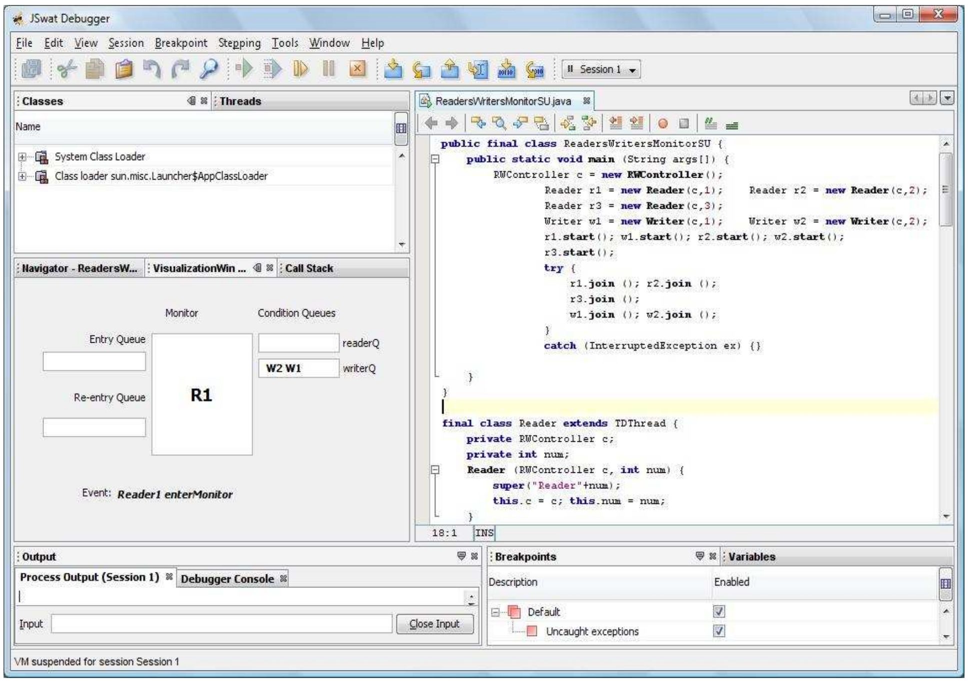
ReadersWritersMonitorSU.java

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

18:1 INS

Event: *Reader3 enterMonitor*

Figure 7.45 R3 enters the monitor to indicate its completion



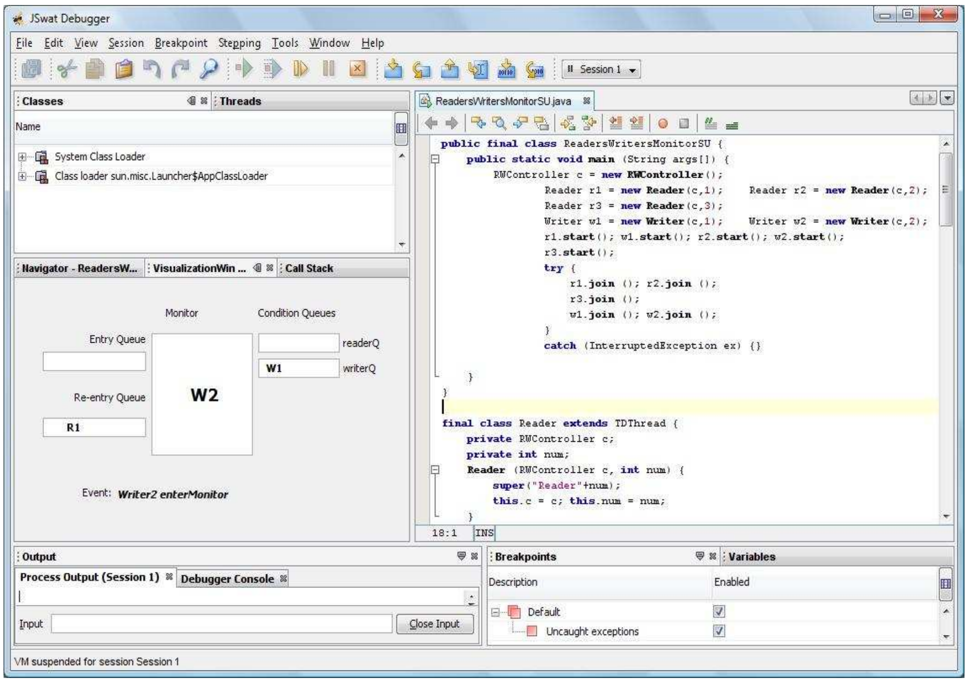
ReadersWritersMonitorSU.java

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

18:1 INS

Event: *Reader1 enterMonitor*

Figure 7.46 R1 enters the monitor to indicate its completion



Event: *Writer2 enterMonitor*

ReadersWritersMonitorSU.java

```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

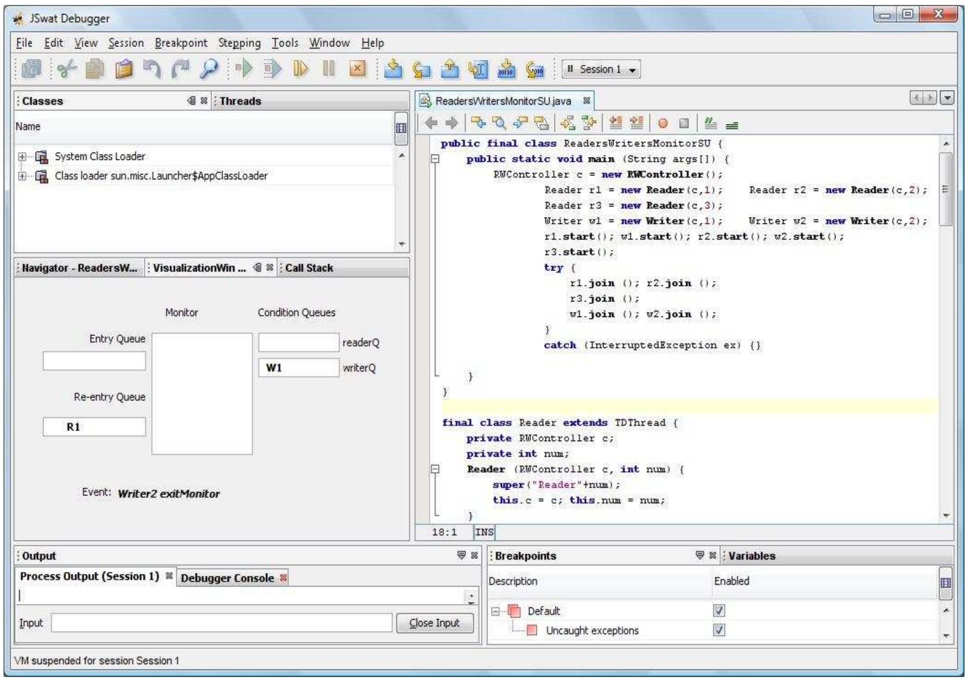
18:1 INS

Figure 7.47 R1 enters the re-entry queue

Now, since there were readers reading all these times, the writer threads in the writer queue is not signalled. Now, since there are no more reader threads, the thread *R1* executes a Signale and Urgent wait call. Upon executing this command, the current thread would go to the re-entry queue and one of the threads in the writer queue would be signalled to enter the monitor. In this case, the writer thread *W2* gets a chance to write and exit. These are shown in the Fig. 7.47 and Fig. 7.48

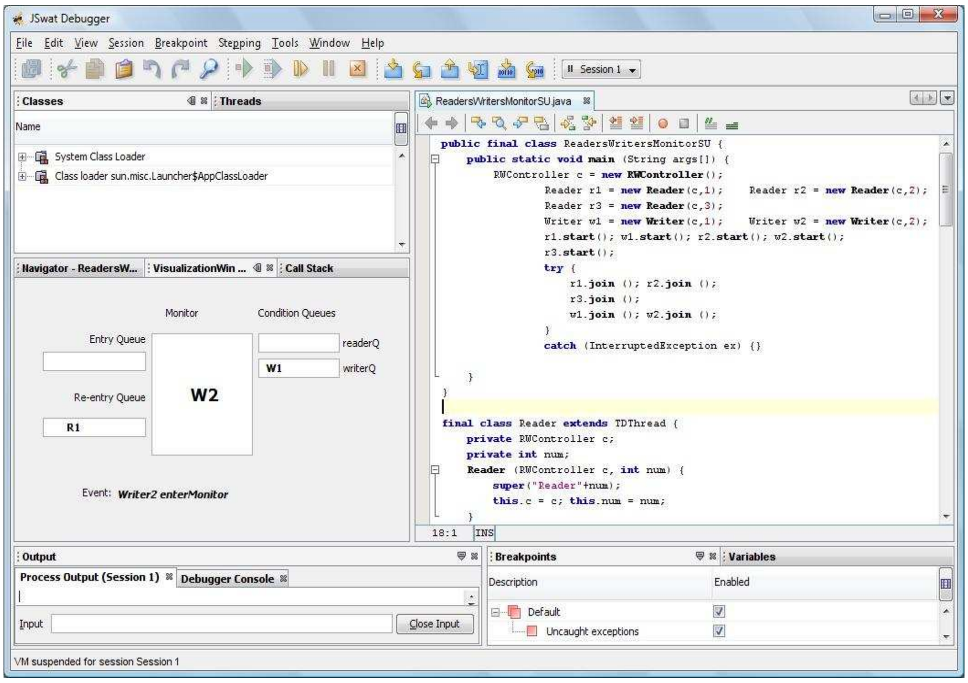
Now, the writer thread W2 enters again to exit the monitor after writing. This is shown in Fig. 7.49 and Fig. 7.50

Now the final thread enters the monitor and executes a signal and exit monitor call. After finishing writing, it once again enters the monitor and exits the monitor. These are shown in the Fig. 7.51 and Fig. 7.52



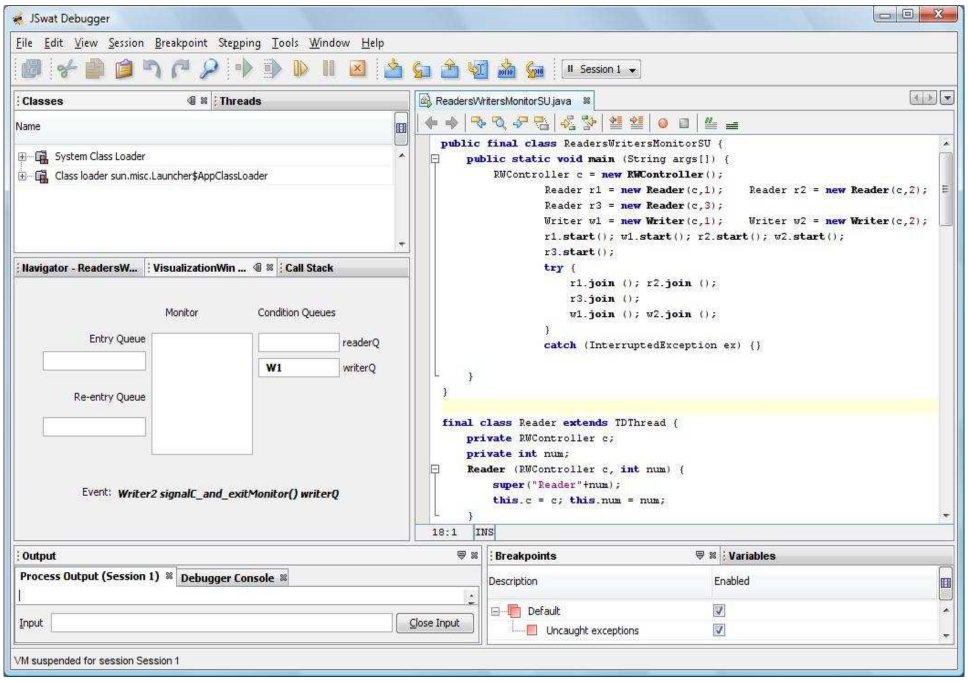
```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.48 W2 exits monitor



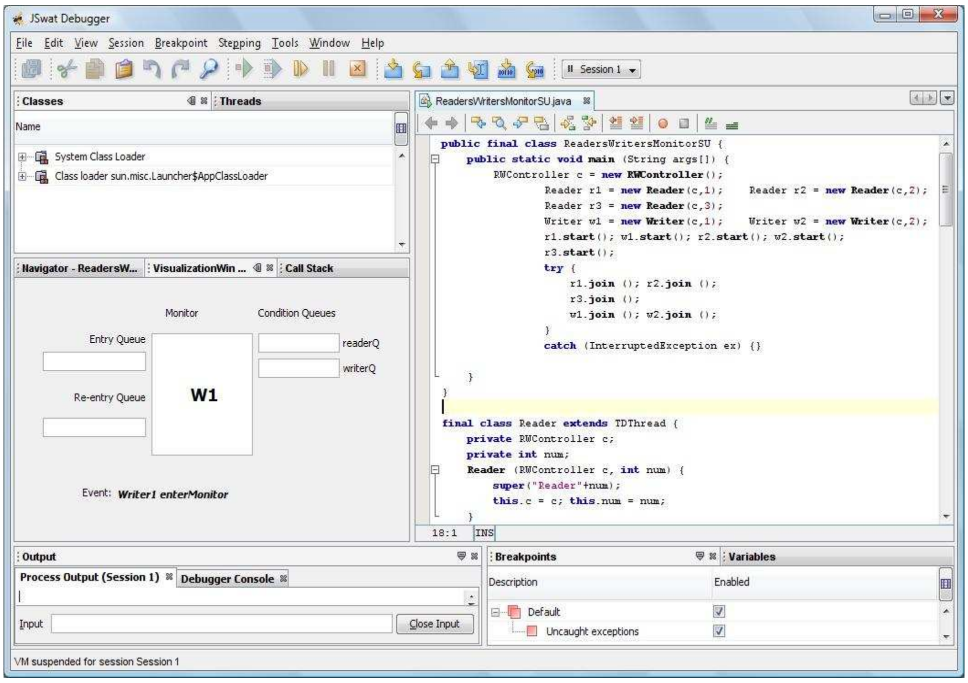
```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.49 W2 enters monitor again to exit



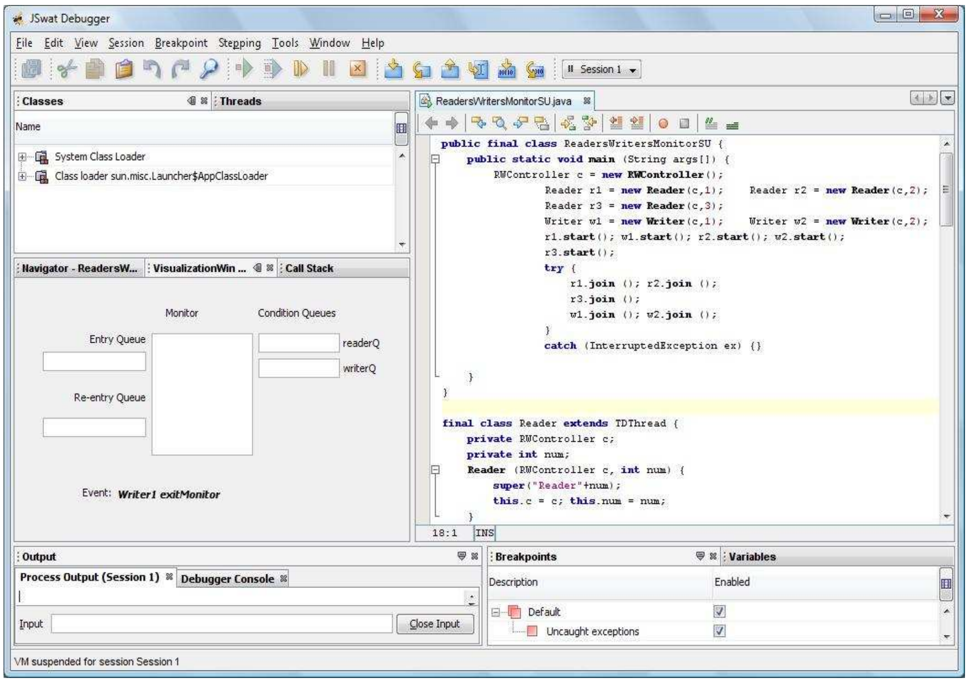
```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.50 W2 executes Signal and exit monitor to exit the monitor



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

Figure 7.51 W1 enters monitor again to exit



```
public final class ReadersWritersMonitorSU {
    public static void main (String args[]) {
        RWController c = new RWController();
                Reader r1 = new Reader(c,1);    Reader r2 = new Reader(c,2);
                Reader r3 = new Reader(c,3);
                Writer w1 = new Writer(c,1);    Writer w2 = new Writer(c,2);
                r1.start(); w1.start(); r2.start(); w2.start();
                r3.start();
                try {
                    r1.join (); r2.join ();
                    r3.join ();
                    w1.join (); w2.join ();
                }
                catch (InterruptedException ex) {}

    }
}

final class Reader extends TDThread {
    private RWController c;
    private int num;
    Reader (RWController c, int num) {
        super("Reader"+num);
        this.c = c; this.num = num;
    }
```

```
18:1  INS
```

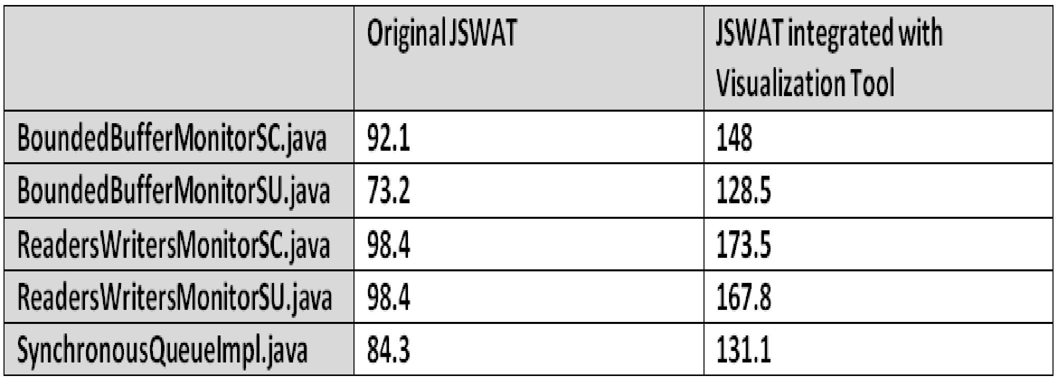
Figure 7.52 W2 executes Signal and exit monitor to exit the monitor

7.4 Execution Time Comparison

To determine the overhead of the visualization tool, we compared the execution time taken by the visualization tool with that of the original JSWAT. The concurrent programs taken into consideration for this measurement are BoundedBufferMonitorSC.java, BoundedBufferMonitorSU.java, ReadersWritersMonitorSC.java, ReadersWritersMonitorSU.java and SynchronousQueueImpl.java. The following are the list of facts about the test programs considered.

- Each of the concurrent programs had about 150 lines of code with six threads
- Each of the concurrent programs had two condition variables
- Monitor events considered to visualize are *enterMonitor()*, *exitMonitor()*, *waitC()*, *signalC()*, *signalCall()*, *signalC_and_exitMonitor()*

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 imple-



	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 7.53 Average execution time

mented for both original JSWAT and the visualization tool. The readings are listed in Figure. 7.53. As we can see, the visualization tool, on average, increases the execution time by order of tens of milliseconds.

CHAPTER 8

RELATED WORK

8.1 Jitan

Jitan [4] is a visualization environment for concurrent, object-oriented programming, developed in the INRIA(France's national computer science research institute) Oasis project. Jitan provides both textual and graphical visualization of objects and thread activities at execution. It displays information such as object graph's topology, thread activities and status, locks and synchronizations. Jitan is derived from Java's syntactic specifications and an operational semantics of the language based on Natural Semantics and Structural Operational Semantics. The specifications were written within Centaur, a generic programming environment.

In Jitan, Java's syntactical specification is used to derive a parser that transforms a program's textual form into a structural representation. Jitan represents every structures object as an abstract syntax tree. Natural Semantics style is used to describe object-oriented features and Structural Operational Semantics to specify the mulththreading semantics. The operational semantics simulates concurrency with a deterministic thread interleaving. Currently, programmers cannot choose a particular interleaving or act on the scheduler. Here, Java operational semantics in terms of a transition system, modeling possible transitions from one configuration to another. Objects, threads and configurations are modeled as semantic structures. Any activity consists of a status and a continuation. A continuation consists of a thread identifier, the name of the current method, and execution environment composed of parameters(name-value pairs) and local variables (name-value pairs), and an instruction list. Centuar automatically generates a simulator that takes as input a syntactically correct Java program and outputs a list of objects and threads denoting the program's behavior. Jitan shows the object list using two visualization engines, both based on the semantic structure modeling the object list. The textual view is a direct printing of the list of objects and threads. To generate the graphical view, which shows the object graph's complete topology, Centaur's graph server is used. It builds the graph's nodes and edges by traversing the abstract syntax tree representing the object list. For each object, it creates a node representing the object, and for each attribute or local variable value that is a reference, it creates an edge between the two involved objects.

The semantic interpreter's duty is to notify the visualization engines of important events. Notification is done by calling highlighting primitives with the entity which has changes passed as a parameter. The semantics is equipped with notifications. Notifications tell the engines to change a field's value or a thread's status or to add or remove a lock on a given object or an arc between two objects.

In the graphical visualization, elliptical nodes stand for objects or classes, and black arrows symbolize references between objects. Threads are distinguished by a rectangle around thread names. The visualization highlights references (arrows) between objects and object types (labels). Different colors make thread status visible. By clicking on an object, the programmer can use a zooming process to examine the object's references-that is, fields and local variable values. In this mode, variable names (field or local variable) appear in a rectangular node attached to the corresponding arrows. A smaller font and a dark blue frame distinguish local variables.

Jitan helps in identifying data access problems, that is one thread might call a method that modifies data that is being read by another thread. Using the zooming feature of the graphical visualization, the programmer will be able to see that several threads can access an object. This indicates the need for synchronization to protect the object's fields. Users can abstract an object graph that represents a Java program's execution. An abstraction lets programmers consider real Java programs with a large number of objects and threads. It also provides a clearer view of a selected part of the constructed graph. An abstraction can focus on an object subgraph-for example, by graying out the other parts of the global graph without changing shapes. An abstract view also can visualize only this subgraph and what happens to it during program execution.

The advantages of Jitan are:

- No instrumentation is required at the source level.
- Because interpretation takes place at the source level, developers have a better understanding of program behavior, thanks to the link between execution and source code.
- The effects of program execution are shown on the fly, via visualization and animation of the program interpretation.
- Jitan can complement traditional debugging tools.

Jitan displays the visualization in a graphical manner where each construct is displayed as a node of a graph. This necessitates the user to be well-versed with graphical notations.

8.2 Visualizing Java in Action

[5] This research work has been conducted in Brown University by Steven P. Reiss and his research team. The goal of the research is to build a visualization system with minimum overhead, maximum information and show what the program is doing in real-time. The system should also maximize the amount of information that was collected. The visualization system displays information about what classes were currently executing, what was happening to memory, and what the various threads were doing.

The entire program execution is split into time intervals and then the summary of what the program did in each interval is displayed. This is achieved by classifying classes into three categories. Detailed classes are those directly in the user's application. Library classes are grouped into packages and only the initial entry into the library is visualized in the system. Classes that are neither detailed classes nor library are treated at an intermediate level of granularity. To identify which threads are executing and the state of each thread, entry and exit events are used. This should be augmented with information about synchronization and synchronized methods and blocks. This information is collected by inserting calls immediately before and after each synchronized entry and a call immediately before a synchronized exit. This helps in identifying states where the thread is waiting on a monitor, running inside a monitored region, or releasing a monitor. Finally, event calls are inserted on each allocation, noting the type of object being allocated for each. Also, information about the total number of objects of each class allocated and the class or package that is the source of the allocation is gathered.

The visualization shows a large number of objects and several pieces of information about each object such as the number of entries, the number of synchronization calls, the number of allocations, and the number of allocations by methods for a class; the time spent in each of the possible states for a thread. For displaying this large amount of data box display visualization is used. Each class or thread is represented as a box on the display. The height of the rectangle is used to represent the number of calls, the width represents the number of allocations by methods of the class. The hue of the rectangle represents the number of allocations of objects of the given class. The saturation of the rectangle is used as a binary indicator if the class has been used in the interval. The brightness of the box is used to represent the number of synchronization events on objects of the present class.

This visualization system displays to the user the number of threads executing at any given time and their states, number of allocations, memory used etc. But the interaction among the different threads is not displayed and the support extended to multi-threaded applications is limited.

CHAPTER 9

CONCLUSION

We conclude that visualization of threads is an essential feature of debuggers for testing and debugging multithreaded applications. Though there are a lot of debuggers that shows the thread values at various instance of time during execution, it is often very difficult to test and debug multithreaded applications in those debuggers as they dont visualize the threads at run time.

JSWAT is an open source debugger that has most of the features that makes it easier for the programmers to debug software but also for visualization of threads. Hence we decided to add this enhancement to this debugger.

The GUI for the Visualization of threads, called Visualization Window is built in such a way that it accommodates the threads at run time. Hence the visualization happens on the fly, which makes it easier for the programmer to better understand the program to debug. Also the visualization window is integrated in a synchronous manner with the JSWAT UI so that if JSWAT encounters a breakpoint, then the visualization is also stopped at that point. Later, it is resumed from there on. This helps in understanding the program and tracing bugs in the code.

Since the GUI is capable of handling multiple threads at the same time, the visualization does not get blocked at any point in time. Overall, the design of the visualization window and the integration is such that it requires very few skills to operate it.

My contribution in this thesis work is to identify the conditional queues from the user application. I also implemented a module to identity the events in the user application that changes the state of the thread. These events are interpreted by the visualization rules to visualize the threads. I helped in the design and implementation of the visualization rules. Another important component that I implemented is the synchronous queue. This synchronous queue helps in temporary storage of the events identified from the user program as well as in the synchronization of the threads.

REFERENCES

[1] Modern Multithreading Java code package: http://cs.gmu.edu/ rcarver/ModernMultithreading/LangLibTools.htm

[2] JPDA Architecture: http://java.sun.com/javase/6/docs/technotes/guides /jpda/architecture.html

[3] Carver, Richard H., and K.C. Tai, *Modern Multithreading*,Wiley, 2006

[4] Attali, Isabelle, Caromel, Denis, and Russo, Majorie, *Graphical Visualization of Java Objects, Threads, and Locks*, IEEE Distributed Systems Online, January 2001

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

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

[7] Concurrency tutorial: http://java.sun.com/docs/books/tutorial/essential/ concurrency

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

BIOGRAPHICAL STATEMENT

Arun Ramani was born in India, in 1984. He obtained his B.Tech in Information Technology in 2005. Subsequently, he worked as a Graduate Engineer Trainee for Sify (Pvt) Ltd, in Chennai, India upto 2005. His interest in Software Design and Development brought him to the University of Texas at Arlington, where he obtained his M.S. degree in Computer Science and Engineering in 2008.