**Java HotSpot’s Deadlock Detector**

CS 356 - Winter 2014

Cal Poly Pomona

**Team Red:**

Nicolas Ohannesian

Alexa Dye

Thu Lam

Sapir Avrahami

Table of Contents

Introduction…………..….…….3

Development History…......4

Thu’s Contribution……...…..6

Sapir’s Contribution…...…..8

Alexa’s Contribution…..…..10

Nicolas’s Contribution..…..12

Conclusion…………..…………..14

**Introduction**

Java has many built-in functionalities. The Java HotSpot virtual machine is a specific implementation of the JVM that is used in the official Java releases by Oracle and the popular OpenJDK that is used primarily on Unix platforms. Within Java HotSpot lies a specific set of functions designed to detect deadlocks at runtime and help developers find and eliminate potential problems in their multi-threaded applications. While the virtual machine runs the program it monitors the threads for possible deadlocks. If a deadlocking situation is detected, the Ctrl-break handler will execute a function to print out, into the thread dump, information about the deadlock. It will display the threads that deadlocked, what element deadlocked, and the location in the code that the deadlock occurred. While this ability that Java HotSpot contains is very useful and can tremendously improve the development time by reducing the amount of time it takes to debug a multithreaded application, a better understanding of how to obtain the information it collects and the limitations of its programming, is essential.

**History**

The first week we learned about our tool, the java virtual machine. With some knowledge about how it is capable of detecting deadlocks in code, we researched about a basic shared resource lock. This research gave us a jump start to our project and also helped us understand a little bit more how deadlocks worked on a basic level. The second week was major challenge for our group; two of our members had dropped the course last minute, and the task for that week was to test Java Grande benchmarks with our deadlock detection tool. At the time, the group was still trying to get used to Java HotSpot, and deadlocks in general, and we had also picked up a new member to the team, so there was somewhat of a rush to teach the new member what the group has learned so far, as well as get Java Grande up and running. That was our only major bump in the road, so to speak.

The third week was much easier for the group, as we had each agreed to research about deadlocks Java HotSpot could detect and could not detect. The big challenge here was to find deadlocks HotSpot could not detect, since most of them were easily detected by HotSpot. Of all the deadlocks we researched, we found only one deadlock that could not be detected, the ReentrantReadWriteLock, due to the Ctrl-break handler not having a record of read locks, only write locks.

From the fourth week, up until the day we presented on Java HotSpot, our group was pretty relaxed about our final task, which was to create a Graphic User Interface for our tool. Our group had come up with a rough flow chart of how we were going to execute everything, which included the use of Java Swing and AWT, and a shell script. A couple of our members were comfortable with Java Swing, while the captain of the group knew how to work with shell scripts. Every single week, the day before class, we would all meet up and combine our work. The shell script essentially tied our GUI and Java HotSpot together.

When the GUI started, the user would be prompted to choose a java source code file. After the user chose to perform deadlock detection on that file, the shell script would grab that file path, compile it, run it, and process its thread dump. It would then trim that thread dump so that it only showed the most valuable information to the user, which was the number of deadlocks found, the number of threads, and where in the java source code the deadlock happened. The trimmed thread dump was then shown to the user in a pop up message, where the user was able to click on the hyperlinked texts that said where specifically the deadlock was found in the source code. The hyperlink would then take the user to another popup, where the source code was displayed, along with a highlighted line that indicated where the deadlock happened.

Hyperlinking was given to us as a task in the latter stages of the project. We faced multiple challenges when we had to deal with hyperlinking. Continuous research was involved on the subject over a span of two weeks, but soon, we found out that Java Swing had a Hyperlink Listener under its Event class. We had also found out that Java Swing’s JTextArea had its own Highlighter class, so highlighting specific lines of the source code was pretty straightforward from that point on.

Everyone in the group contributed towards our project, in one way or another, and although we had a few bumps in the road in the first few weeks, we were able to produce a smooth and sleek GUI for detecting deadlocks with Java HotSpot.

**Thu**

I faced many challenges during this project. The first three weeks were tough because I was the new member and did not know what the group was doing. Plus I did not know anything about deadlocks or Java HotSpot. These concepts were totally new to me especially when I’ve never used Linux before. Luckily, I had a very helpful team that was always there to assist and answer my questions. I was fortunate to have the team captain who was very knowledgeable about Linux and scripting and another team member who was knowledgeable about Java Swing. My task for the third week was to find any types of deadlocks that Hotspot could detect. I did some research and was able to find two of them.

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock. An example of deadlock taken from Oracle Documentation describes a situation where Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. Inside the class “Friend”, there are two synchronized method called “bow” and “bowBack.” In the main method of the application, two Friend objects and Runnable Threads are created to mirror this bowing action. When the application runs, both Threads will block when they attempt to invoke “bowBack.” Neither block will ever end, because each thread is waiting for the other to exit “bow”. There are many types of deadlock that Java Hotspot is capable of detecting. During the course, we specifically tested for two types of deadlocks that Hotspot was able to detect. The first type was basic shared resource deadlock which was tested on two and three resources. In the tested program using three resources, SyncThread implemented Runnable interface and it worked on two Objects by acquiring lock on each one of them one by one using synchronized block. In the main method, three threads would be running for SyncThread and there was a shared resource between each of the threads. The threads were run in such a way that it would be able to acquire lock on the first object but when it was trying to acquire lock on second object, it went on wait state because it was already locked by another thread. This formed a cyclic dependency for resource between Threads causing deadlock. When the program was executed, it never terminated because of deadlock. To analyze a deadlock, the java thread dump of the application was examined and it clearly stated the deadlock situation as well as threads and resources involved causing such situation. For analyzing deadlock, every resource had a unique ID that showed which thread was already holding the lock on the object. The dump file showed the threads with state as “blocked” and the resources it was “waiting to lock.” The second type of deadlock that we tested was ReentrantLock deadlock. Using the same code created for the first test, the ReentrantLock class was imported and three ReentrantLock objects were created, replacing the three “Object”s in previous test. The deadlock situation was determined to be the same based on the thread dump.

Starting from the fourth week, I was assigned to help my team members to get the GUI to function as it was supposed to. It happened that we faced another challenge which was hyperlinking. We were able to get it to link to the source code. However, it did not highlight the line number and there was no indentation when the source code was transferred to a JEditorPane. I was thinking about writing a method to add indentation to the JEditorPane whenever the source code had a “{.“ One of our members researched and found out that JTextArea had its own Highlighter class and we were able to solve the problem of connecting the Hyperlinking to JTextArea and its Highlighter.

**Sapir**

The start of the project was probably the most difficult part. In the first two weeks I was barely getting by. Understanding deadlocks and how Java HotSpot detected them was a great challenge. As far as other tools we used, I did not know how putty, FileZilla, and the Virtual Machine worked either, and Github created a big difficulty for us, since its intricacies were a mystery to most of the group. As someone who had never used Linux before, the terminal seemed alien. Now I am not saying I am a master programmer that can solve anything and everything in a few minutes, but in those first two weeks I felt like a complete novice programmer. If that was not enough, two of our original group members dropped out of the class at the very last second. It took me a little while to learn and completely understand how everything worked. The hardest part of this process was understanding the logic behind deadlocks. Logically it is simple: two threads lock onto each other’s object and wait for the other one to finish, since either of them can finish we get a deadlock. Yet, when I actually programmed and looked at some code online it was another story. The only redeeming thing in all of this is the fact that Java HotSpot had the deadlock detection capabilities within. So I did not had to work extra hard integrate an external program into our test code. Thankfully at the end, I was able to understand the grand scheme of things.

On the second week we had to run the Java Grande Benchmark and see if Java HotSpot can detect the files in its folders. Since I was still unsure about how HotSpot’s deadlock detection worked I had worried I would not be able to help. I would like to believe I was wrong about that since I was able to run the programs and get their thread dumps. The section of files in the Java Grande Benchmark file collection I was designated to was section three. Once I got all the necessary dump files I looked them over and made tables and charts detailing how many deadlock were found, the program’s overall runtime, and the number of solutions.

When week three came around I had a more focused task. I was asked to find and/or create deadlock programs that Java HotSpot will not be able to detect. Sort of like break the system. I came up with what I thought was a good example. It was a Stack class that used LinkedList and whenever the user pushed or popped an item the list was synchronized. So if the user popped an empty Stack the program will reach a deadlock, which HotSpot could not detect. That was supposed to be the example, but after the group meeting of the same week it was just an error on the Stack object part and not HotSpot actually detecting a deadlock. I guess I messed up in this regard, but I learned from my mistakes.

From week four and onward we were tasked with creating a GUI that reads a java file and see if HotSpot can detect any deadlocks in it. My teammate, Alexa, plunged head first into working on that, but when she had a problem with the hyperlinking I stepped up to help. The problem was not the hyperlinking per say, it was more of a problem to make the java file open in the correct line where the deadlock was detected. For that we needed the line number of where the deadlock happened, which was not simple. So what I did was, in the code the place where the hyperlink was highlighted I created a for loop to look through the hyperlinked string (using .charAt(i)) and get the location where the line number exist. Since the line number was between two brackets I just had to purse the string using .substing() and then convert the number into a useable integer using Integer.parseInt(). So now when we had the actual line where the deadlock occurred we were able to highlight the line. In the end, this part of my code had to be completely re-written to be able to connect to the JTextArea properly. My code did form a basis to the way the lines were highlighted and it was helpful in developing the last step of our interface the long run.

**Alexa**

The first three weeks of the project were the most challenging for me. I initially expected either a project that had to do with Object Oriented Programming or an Android Application Programming. I was not familiar with deadlocks, or deadlock tools, at all. The first week was mainly getting familiarized Java HotSpot’s deadlock detection. The second week was a struggle, since I hadn’t practiced actually detecting deadlocks with HotSpot, on top of the already assigned task of having to get Java Grande to work. After we learned that two of our members had dropped the course at the last minute, I had to push myself to learn Java HotSpot and deadlocks; this only made it more stressful. With the help of the rest of the team and especially the team captain, I was able to find some common ground with our tool. My task for the third week was to find deadlocks that Java HotSpot could not detect. The number of deadlocks in this category was almost scarce. Most of the ones that I, and my teammates, had found were detected. The only one that I managed to find that hadn’t been detected was the ReentrantReadWriteLock. The challenge here was familiarizing myself with why the deadlock had occurred. I knew what deadlocks were, but as to why Java HotSpot could not detect this type of lock needed a bit more research.

From the fourth week, up until the last weeks of our project, I was assigned the task to get the GUI of our tool to work. At first, we, as a team, decided to just create a rough layout of how our GUI would look. Being ambitious as I was, I wanted to put a little more effort in and get some parts of it working, such as the JFileChooser, that allowed the user to choose a file to upload, as well as position and align a few labels and buttons. This extra step helped us lift some weight off our shoulders, as only minor details were needed before our GUI was essentially complete.

The biggest challenge for me was implementing hyperlinking that linked the user to specific lines in the source code. I researched about a plethora of possibilities for my team since I knew hyperlinking wasn’t going to be an easy task. Over a span of a couple weeks, I coded multiple possibilities, but each one tested to be more tedious than the last. Research had suggested that we use a JEditorPane, or VIM. However, a couple days before we had to present our results, I found out that Java Swing’s Event class has a Hyperlink Listener, and JTextArea has its own Highlighter class. Using these two and a couple JOptionPanes, I was able to connect the hyperlinking to the JTextArea and its Highlighter.

**Nicolas**

Our team began with five members. The whole research and test approach to the coursework was a very different and interesting approach, so I thought it would be a good experience to lead my team, and proposed to be team captain. The first week of work, we had to show Java HotSpot detecting deadlocks with a simple test program we wrote. Only myself and one other team member were familiar with using Linux, the terminal, and GitHub, so while we taught the rest of the team about installing the necessary software on their systems, we also had to learn how Java HotSpot’s deadlock detection worked. After presenting our efforts the first week and our team shrank and we had two members replaced by one member. Our team was in a bit of a tailspin, having to teach a new member (who was not a computer science student) everything we learned about at the start of class and still learning new material required meticulous effort.

My first goal was always to make sure the team understood what they had to do and could find the resources they needed to complete their work. In the first couple of weeks I did tend to put more work in than I should have, simply because my team was learning. I decided the best way to approach our tool was to use a bash script in Linux to run any necessary files through the Java virtual machine and extract the necessary thread dumps to check if there had been a deadlock in the code. I was able to utilize this script in the first week, to run a simple test with two deadlocking threads, and modify it to work the second week for the Java Grande benchmark tests. I had to do some slight modifications to Java Grande itself due to it being severely aged and not compatible with the newer Java versions. Using some separate functions and case switches in the script I was able to allow the user to select which benchmark to run, or even run multiple benchmarks at the same time if they wanted to. When we later had to make our own deadlocking code, showing different types of deadlocks, I was able to modify the script to simply use our constructed tests instead of the Java Grande benchmarks. This hurdle was slight compared to modifications needed to make the script compatible with our GUI.

The next logical step in our research and development was to make Java Hotspot’s deadlock detection easier to work with. When the professor suggested a GUI, our team went straight to work on designing its appearance and functionality. While my team worked with Java Swing to implement the interface, I was adapting my script to work with a java program rather than a user. Once done, it was entirely re-written. My team and I worked on integrating the two parts together. The GUI would take the Java file from the user, then save the file’s name and location. This information is passed to the script as parameters. First the GUI orders the script or compile the program. Then the GUI makes the script run the compiled program. The script waits to see if the program deadlocks. If it does reach a deadlocked state, the particular process is ended and the thread dump is analyzed. The pertinent part of the thread dump (involving deadlocks) is saved and sent to a file for the GUI to display, or if no deadlocks were detected, writes a message saying so to the file. The script then deletes compiled java classes from the directory to keep the folders clean. Finally, the GUI displays the Java HotSpot’s deadlock findings to the user.

I was quite pleased with the final product. It adds to the functionality of the Java virtual machine by making one of its features specifically accessible to developers of multithreaded applications. Throughout the project there was a struggle to complete the objectives for every week, but with my direction, and the entire team’s enthusiasm to learn, we were able to come together and complete all our goals. I am happy to say this experience proved to be very educational and helped me understand team dynamics in software research. It also taught me how to help colleagues learn about software and complete any work they set their sights on.

**Conclusion**

Deadlocks have been a problem since multiple threads were conceived. Java HotSpot has proved to be quite capable of detecting deadlocked threads. It does have the drawbacks of not automatically notifying the user and requiring the user to take several extra steps to access information on encountered deadlocks. This research project helped discover that with a script, those particular functions can be easily utilized. A GUI developed on top of that script helps with the access of the output to the user. In today’s world, there are many places where deadlocks can be encountered. With the internet and databases being used for most things computer-related, the possibility of deadlocking threads is dreaded constantly by developers. If threads deadlock while software is under use, it can lead to quite an ordeal. Customers could end up with sold out items, trains could be stuck without being able to proceed, and even programmers could be waiting for code to download from their repositories with no end in sight. Code must be thoroughly debugged and tested to make sure threads do not cause one another’s downfall. There are many tools available to help developers find and eliminate possible deadlocks in their multithreaded applications. If one is developing in Java, regardless of also utilizing a third party tool to detect deadlocks, there is one always running in the background, that is quite powerful and convenient.