Asian Journal of Information Technology 13 (6): 356-362, 2014
ISSN: 1682-3915
© Medwell Journals, 2014

Deadlock Maps: A Dynamic Deadlock Detection for Multithreaded Programs

[1]A. Mohan and [2]P. Senthil Kumar
[1]Saveetha Engineering College, Anna University, Chennai, Tamil Nadu, India
[2]SKR Engineering College, Chennai, Tamil Nadu, India

Abstract: Deadlock freedom is the major challenge in developing multithreading programs. To avoid the potential risk of blocking a program, prior monitoring of threads can be made during the execution process. The proper monitoring scheme can monitor the threads and can identify whether the threads might enter a deadlock stage. It maintains a back up to store the threads. So, after the execution of one thread the injection of the other thread can be made from backup into the processing stage. By using this process the deadlock can be avoided in the multithreading environment. In the proposed system, the thread monitoring and thread mapping techniques are implemented to identify the threads running in the program. A map is present which is used to store the thread objects, the locks acquired and requested by them. Whenever, a thread tries to acquire a lock and if the access is denied, then it waits for certain period of time. After the time period expires, the thread again tries to access the lock. If the access is still denied then the thread traverses the map to identify the threads that have requested or held the same locks requested by it. If it finds any such threads then it detect that deadlock has occurred. The deadlocked threads wait for each other for infinite time. Now, the thread releases all the locks acquired by it, thereby allowing the deadlocked threads to complete their operations. If more than one thread detects deadlock then priorities are assigned to them at random manner. According to the priorities of threads, they wait for a while (i.e., let other threads to complete their operation). According to the priority, thread execution states are changed. It helps the threads to recover from deadlock situation and allows the threads to complete their execution.

Key words: Multithreaded program, synchronization, deadlock monitor, thread map, priority

INTRODUCTION

Deadlock-freedom is a major challenge in developing multi-threaded programs as a deadlock cannot be resolved until one restarts the program (mostly by using manual intervention). To avoid the potential risk of blocking, a program may use try-lock operations rather than lock operations. In this case, if a thread fails to acquire a lock, it can take appropriate action such as releasing existing locks to avoid a deadlock. In the existing system, the usage of mapping is not implemented. In another approach circular mutex wait deadlocks and lock graphs are cleared but this model is not suited for all environments. The existing Dynamic methods have less efficiency when compared to the Static Deadlock Analysis Method. The proposed system provides an efficient mapping technique for avoiding deadlocks depending upon priority.

The main aim of the project is to avoid the deadlocks occurred in the threads during execution. It is done by providing a map that stores the thread objects, locks acquired and requested by the thread. In this case, if a thread fails to acquire a lock, it can take appropriate action such as releasing existing locks to avoid a deadlock.

In order to avoid deadlocks in threads during the execution process a monitor is introduced in the proposed research that identifies the threads running in the program, i.e., the thread objects are identified. After this process, a map is generated that store the thread objects and the locks acquired and requested by them. Whenever, a thread tries to acquire a lock and if the access is denied then it waits for certain period of time. After the time period expires, the thread again tries to access the lock. Due to some reasons, if accessing the locks is still denied then thread traverses the map to identify the threads that have requested or held the same locks requested by it. If it finds any such threads then it recognizes that deadlock has occurred after which the deadlocked thread will wait for each other for infinite time. When, it finds that deadlock condition prevails the thread releases all the locks acquired by it, so that it might allow the deadlocked threads to complete their required operation. In another

Corresponding Author: A. Mohan, Saveetha Engineering College, Anna University, Chennai, Tamil Nadu, India

scenario, if more deadlocks are detected then according to the priority the execution of the priority based threads are executed in random manner. During this process the threads involved in execution are been backed up. According to the priority, the threads execution states are changed. This helps the threads to recover from deadlock situation and let the other threads to complete their execution.

LITERATURE REVIEW

When developing a dynamic deadlock detection technique for multithreaded programs, a multi disciplinary approach is essential. The static and dynamic techniques are used for exposing deadlock potential (Agarwal *et al*., 2010). It has three extensions to the basic algorithm (logic graph) to eliminate and to label as low severity or false warning of possible deadlocks. The extension of Lock Graph algorithm is to detect the deadlock in static and dynamic techniques.

A new technique in practical static race detection is proposed for parallel loops in java (Radoi and Dig, 2013). The utilization of these constructs and libraries improves accuracy and scalability. The new tool called IteRace has been introduced which includes a set of methods that are specialized to employ the intrinsic thread, safety and dataflow structure of collections. The IteRace is fast and perfect enough to be realistic. It scales well for programs of lakhs of lines of code and it reports little race warnings, thus shuning a common consequence of static analyses. The tool implementing this method is fast does not delay the programmer with many warnings and it finds latest bugs that were conformed and fixed by the developers.

Detecting atomicity violations using dynamic analysis technique is presented (Flanagana and Freund, 2008). A more fundamental non-interference property is atomicity. When, a method execution is not affected by concurrently-executing threads, then that method is called as Atomic Method. It contains both formal and informal correctness arguments. Detecting atomicity violations combine ideas of both Lipton's theory of reduction and early dynamic race detectors. It is effective error detection for unintended interactions between threads. It will be more effective than standard race detectors.

Flanagana and Freund (2006) proposes the Type Inference algorithm for rccjava. The performance of the algorithm is applied on programs up to 30,000 lines of code. The resulting annotations and race-free guarantee our type inference system. Type Inference algorithm is applied to the concurrent program to manipulate the shared variable without synchronization. This algorithm has some lock variables. Extending this Inference algorithm to larger benchmark has some issue. It produces reliable error reporting.

Hasanzade and Babamir (2012) describe an approach for online deadlock detection for multithreaded programs using the prediction of future behavior of threads. The 74% of deadlocks were predicted using the proposed method. Some specific behaviors of threads are extracted at run time and converted into predictable format using Time Series Method. The proposed method has several advantages compared to the existing static methods. Powerful technique is used for predicting complex deadlocks.

Bodden and Havelund (2010) implemented an efficient algorithm to sense concurrent programming errors online. System programmers monitor the program events where locks are approved or handed back and in places where values are accessed that may be shared among multiple Java threads. The proposed RACER algorithm uses ERACER for Memory Model of java and AspectBench compiler for implementation. In this study, they projected a language extension towards the aspect-oriented programming language AspectJ. The proposed AspectJ have implemented the following three points. They are Lock(),Unlock() and Maybeshared().

Chen *et al.* (2011) examines the performance scaling of various processor cores and application threads. It analyzes the performance and scalability by correlating low-level hardware data to JVM threads and system components. It uses the JVM tuning techniques to solve the problems regarding lock conditions and memory access latencies. The study of performance and scalability of multi threaded java application on multi core systems is done. The proposed method reduces the bottlenecks using JVM tuning techniques. Inappropriate use of synchronization leads to large number of stall cycles.

Joshi *et al.* (2009) present a Novel Dynamic Analysis Method to find real dead-locks in multi-threaded programs. DEADLOCK-FUZZER is the new technique used to find the deadlocks in two phases. In the first phase, a potential deadlock in a multi-threaded program is found using dynamic analysis technique by execution of the program. In the second phase, deadlock creation is controlled using threads scheduler. DEADLOCK-FUZZER is implemented to find the all previously known deadlocks in large benchmarks but it does not discover previously unknown deadlocks in an efficient manner. This technique needs both static and dynamic techniques.

Wen *et al.* (2011) describe a new Java thread deadlock detection approach called as JdeadlockDetector. This system requires source code and built on non-official JVMs for Java thread deadlock detection solutions. Many Java programs cannot be evaluated with these solutions. JdeadlockDetector is fabricated on the official Java Virtual Machine, viz., OpenJDK's HotSpot. JdeadlockDetector have three unique advantages compared to the existing system. They are application transparency, detection accuracy and minimized performance overhead. JdeadlockDetector attains no false negative and diminished false positive. JdeadlockDetector detects Java thread deadlock based on the capability of monitoring the thread states and synchronization states on runtime. In this way, the technique achieves their advantages. To track the control flow and data flow of a Java program, Hotspot introspection architecture has to be extended. This will afford a capability to analyze the vulnerability of Java programs.

A new two phase deadlock detection scheme was introduced which provides efficient memory utilization and time constraints (Luo *et al.*, 2011). The performance of the proposed system is much higher than the traditional approach in finding the potential deadlock in application. First phase reduces lock by filtering out certain locks that cannot participate. Second phase creates smaller lock graph for potential deadlock detection. The proposed research can minimize the overall deadlock detection time and increases the performance.

Researchers focus on developing dynamic deadlock detection technique which reduces the deadlock occurrences.

SYSTEM ARCHITECTURE

The system architecture for the proposed system includes deadlock monitor, analyzing thread states, thread maps and locks. It explains the efficient way for detecting deadlock in multithread program using thread map and priority assignment. The java thread is created based on the set of condition that causes deadlock situation. Each thread is built based on certain lock to access the resources. The deadlock monitor identifies the threads running in the program, i.e., the thread objects is identified. After this process a map is generated that store thread objects and the locks acquired and requested by them (Fig. 1).

When it finds that deadlock condition prevails, the thread releases all the locks acquired by it, so that it might allow the deadlocked threads to complete their required operation. In another scenario, if more deadlocks are detected then according to the priority, the execution of the priority based threads are executed in random manner.

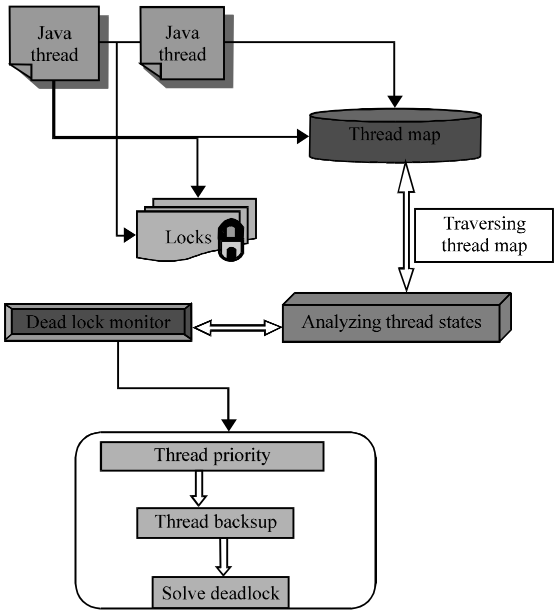


Fig. 1: System architecture

MODULE DESCRIPTION

Module 1 (Thread creation): A multithreaded program is designed such that it contains one or more Synchronized Methods. All the Synchronized Methods of an object are locked whenever a thread acquires any of its Synchronized Methods. Thus, the program is designed to have deadlock problems. The thread creation of multithreaded program includes 1-10 threads (user1 to user10). Each threads acquires different lock in nested way. The program has potential deadlock when each thread send a particular text file to other threads.

The program is constructed based on following set of conditions. User1 thread consists of two operation. First, it sends a file file6.txt to user user2 and receives a file file2.txt from user user7. After receiving file2.txt, it will display the file information. User2 thread does not send any file to other users. It only receives a file file2.txt from user user6 and reads file file2.txt. User3 thread follows the same process like user1. First, it sends a file file4.txt to user user5 and receives a file file7.txt from user user4. After receiving the file it reads file file7.txt. User4, User5 threads are sending some file to other user but it doesn 't receive any files.

User7 send a file file2.txt to user user1 and receives a file file3.txt from user user8. After receiving, it will display the file information. User8 has more than two operations. First, it sends a file file7.txt to user user9 then send a file file6.txt to user user7. Second, it receives a file file3.txt from user user5 and display that file. User9 threads sends a file file4.txt to user user10 and receives file file9.txt and reads the file informatoin. User10 sends a file file3.txt to user user9 and user10 receives a file file9.txt from user user9 and reads file file9.txt information. The sample coding for the thread creation and operation is given as:

```
Thread user1_TH = new Thread()
{
      public void run()
      {
                try
                {
                          Thread.sleep (500);
                           File[] files = new
                           File ("Files"). listFiles();
                           user 1.sendFileTo (user 2,files[new
                           Random ().nextInt(files.length)]);
                }
             catch (Exception ex)
                {
                    ex.printStackTrace();
                }
                } };
```

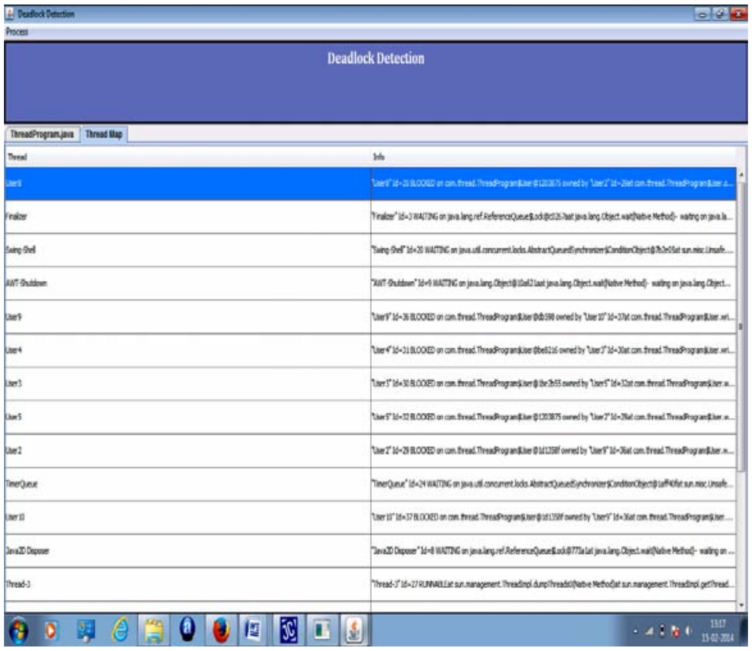
This code is used to send the file from one user to another user. The above code explains the file tranformation between the user1 and user2.

```
          public synchronized void writeFile(User userY, File file)
          {
                try {
                          File InputStream fis = new
                          FileInputStream (file);
                          byte[] data = new
                byte [fis.available()];
                          fis.read(data);
                          fis.close();
                          FileOutputStream fos = new
                          FileOutputStream (this.getName()+
                          File. separator+file.getName());
                          fos.write(data);
                          fos.close();
                          jta.append("\n "+name+" received a
                        file ["+file.getName()+"] from
                        User
                        "+userY.get Name()+"");
                         sendAckTo(this,file);
                }
             catch (Exception ex)
                {
                    ex.printStack Trace();
                 }
              }
```

Module 2 (Mapping the thread): A deadlock monitor is designed to observe the details of threads running in the program. The details of locks acquired and requested by the threads are entered into thread map. Thread map contains list of threads and its corresponding resources. The states of threads are continuously updated. During the program execution the threads states (user1 to user10) will be entered into the thread map. The thread map contains the information about deadlocked threads. Sample example:

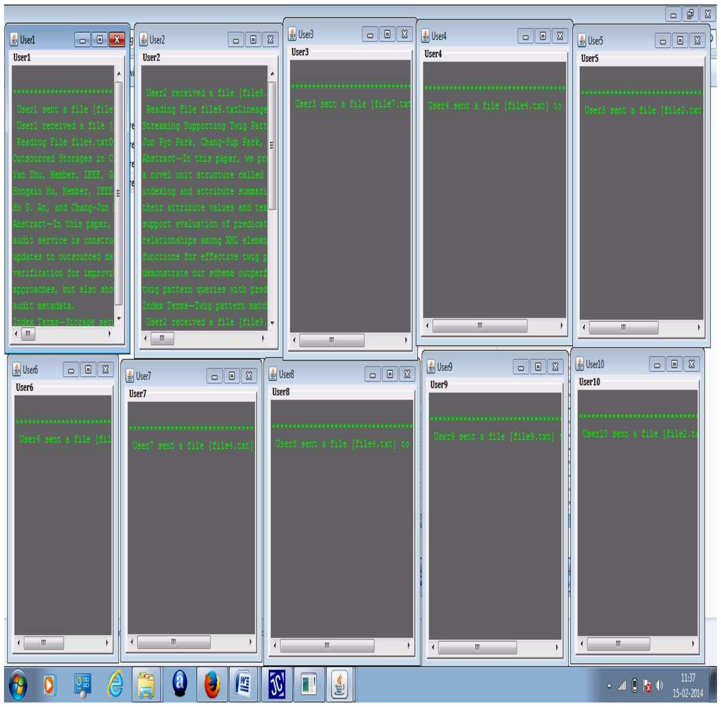
```
"User8" Id = 35 BLOCKED on com.thread. ThreadProgram
$User@1203875 owned by "User2" Id=29 at com. thread. Thread
Program$User.sendFileTo(ThreadProgram.java:364)-blocked on
com.thread.ThreadProgram$User@1203875 at com.thread.Thread
Program$8.run (ThreadProgram.java:200)
```

When the program executes thread 8 enters the deadlock state. The information related to the particular deadlock will be entered into the ThreadMap. It contains the information that explains which threads block the current running thread and object id of the particular threads (Fig. 2).



Thread	Info
User8	"User8" Id=35 BLOCKED on com.thread.ThreadProgram$User@1203875 owned by "User2" Id=29at com.thread.ThreadProgram$User.s...
Finalizer	"Finalizer" Id=3 WAITING on java.lang.ref.ReferenceQueue$Lock@c0267aat java.lang.Object.wait(Native Method)- waiting on java.la...
Swing-Shell	"Swing-Shell" Id=20 WAITING on java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@7b2e05at sun.misc.Unsafe...
AWT-Shutdown	"AWT-Shutdown" Id=9 WAITING on java.lang.Object@10a621at java.lang.Object.wait(Native Method)- waiting on java.lang.Object...
User9	"User9" Id=36 BLOCKED on com.thread.ThreadProgram$User@db598 owned by "User10" Id=37at com.thread.ThreadProgram$User.wri...
User4	"User4" Id=31 BLOCKED on com.thread.ThreadProgram$User@be8216 owned by "User3" Id=30at com.thread.ThreadProgram$User.wri...
User3	"User3" Id=30 BLOCKED on com.thread.ThreadProgram$User@1be2b55 owned by "User5" Id=32at com.thread.ThreadProgram$User.w...
User5	"User5" Id=32 BLOCKED on com.thread.ThreadProgram$User@1203875 owned by "User2" Id=29at com.thread.ThreadProgram$User.w...
User2	"User2" Id=29 BLOCKED on com.thread.ThreadProgram$User@1d1358f owned by "User9" Id=36at com.thread.ThreadProgram$User.w...
TimerQueue	"TimerQueue" Id=24 WAITING on java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@1aff40fat sun.misc.Unsafe...
User10	"User10" Id=37 BLOCKED on com.thread.ThreadProgram$User@1d1358f owned by "User9" Id=36at com.thread.ThreadProgram$User....
Java2D Disposer	"Java2D Disposer" Id=8 WAITING on java.lang.ref.ReferenceQueue$Lock@77a1at java.lang.Object.wait(Native Method)- waiting on ...
Thread-3	"Thread-3" Id=27 RUNNABLEat sun.management.ThreadImpl.dumpThreads0(Native Method)at sun.management.ThreadImpl.getThread...

Fig. 2: Deadlock detection window with threadmap



User1

```
*************************
 User1 sent a file [file
 User1 received a file [
 Reading File file4.txtD
Outsourced Storages in C
Yan Zhu, Member, IEEE, G
Hongxin Hu, Member, IEEE
Ho G. An, and Chang-Jun
Abstract—In this paper,
audit service is constru
updates to outsourced da
verification for improvi
approaches, but also sho
audit metadata.
Index Terms—Storage secu
```

User2

```
 User2 received a file [file9.
 Reading File file9.txtLineage
Streaming Supporting Twig Patt
Jun Pyo Park, Chang-Sup Park,
Abstract—In this paper, we pr
a novel unit structure called
indexing and attribute summari
their attribute values and tex
support evaluation of predicat
relationships among XML elemen
functions for effective twig p
demonstrate our scheme outperf
twig pattern queries with pred
Index Terms—Twig pattern match
 User2 received a file [file9.
```

User3

```
******************************
 User3 sent a file [file7.txt
```

User4

```
**********************************
 User4 sent a file [file4.txt] to
```

User5

```
******************************
 User5 sent a file [file2.txt
```

User6

```
**********************
 User6 sent a file [fil
```

User7

```
*******************************
 User7 sent a file [file4.txt]
```

User8

```
**********************************
 User8 sent a file [file4.txt] to
```

User9

```
********************************
 User9 sent a file [file9.txt]
```

User10

```
*****************************
 User10 sent a file [file2.tx
```

Fig. 3: Threads program execution with deadlock

Module 3 (Program execution with deadlock): The program will be executed after constructing more number of threads with the above set of conditions. The deadlock monitor window helps to run the program. The snapshot of program execution in Fig. 3.

Figure 3 represents the output of the multithreaded program with deadlocks. In this case, user1, user2, user4 and user6 have executed successfully without deadlock. User3, user5, user7, user8, user9 and user10 enter into the deadlock situation. Deadlock has occurred when threads access same resource (e.g., public synchronized void writeFile (User userY, File file)) at same time. Thus, all processes will be waiting to complete its operation.

Mdoule 4 (Random pririoty assignment): The threads running in the program are periodically examined by the monitor to detect deadlock situation. The deadlock situation is identified if one or more threads wait for each other for infinite time. The monitor identifies this situation by analyzing the states of threads. Deadlock monitor periodically executes traverse map operation. During traversal, monitor examines the states of threads running in the program. If the state of thread is identified as deadlock then it has to wait for resourcesfor infinite time. The thread cannot continue its operation any more. After deadlock is detected, threads are randomly assigned to some priority levels at run time. The priority will be assigned based on the map traversing with the help of deadlock monitor. The monitor should observe the details when each thread is running. The random priority will be assigned based on the following process. The process is:

```
private void
checkForDeadLockedThreads (ThreadInfo[]
threadInfo)
{
              try
              {
                 for(int i=0; i<threadInfo.length; i++)
                 {
                if (thread Info[i].getThreadState().equals
                          (Thread.State.BLOCKED))
                          {
                          Thread DeadLockedThread =
                            (Thread)Thread IDMap.get(threadIn
                          fo[i]. get ThreadId());
                          Thread LockOwnerThread =
                           (Thread)ThreadIDMap.get (threadIn
                          fo[i].getLockOwnerId());
                if (!is Dead Lock Detected)
                {
                          is Dead Lock Detected = true;
                          Dead Count++;
                          System.out.println ("Deadlocked
                          Threads found...
                           ["+Dead LockedThread.get Name()+
                           ","+LockOwnerThread. getName()+
                           "] ");
                 }
                           LockOwnerThread.yield();
               Dead Locked Thread. set Priority (secure_rand
               om.nextInt(Thread.MAX_PRIORITY));
               nullifyAll(thread Info[i],DeadLockedThread,
               LockOwnerThread);
LockList.add(DeadLockedThread.getName());
LockList.add(LockOwnerThread.getName());
                                              } }
```

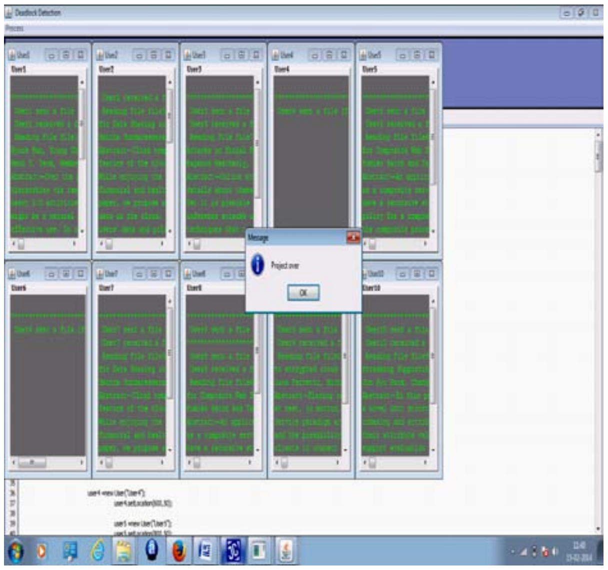


Fig. 4: Threads program execution without deadlock

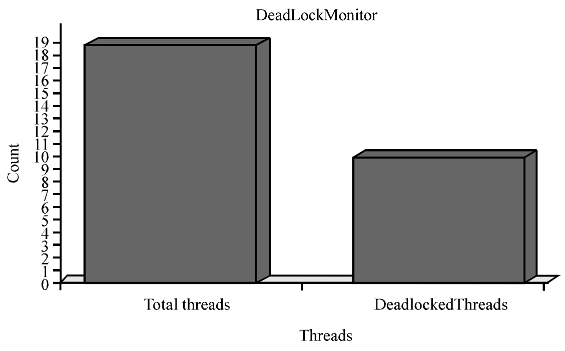


Fig. 5: Perfomance measurement of deadlock detection technique

checkForDeadLockedThreads() Method checks the current states of each thread in thread map. If the map contains any information related to block then the deadlock monitor will assign the random priority number to deadlocked threads using the above process.

Module 5 (Solve the deadlock): Deadlock monitor assigns the priority based on the thread map information. The program will be executed after assigning the random priority number. The program execution after priority assignment is shown in Fig. 4.

Mdouel 6 (Perfomance mesurement): The perfomance measurement of the dynamic deadlock detection technique is given. Figure 5 shows how many threads are running in the thread map and how many deadlocks are solved based on priority level of deadlocked threads. Based on the program, nearly 10 deadlocked threads are solved (Fig. 5).

RESULTS

The proposed system maintains a deadlock map that holds the information about all the threads running in a program. This will reduces the deadlock occurrences based on the random priority assignment. The deadlock monitor is control the overall process of program execution.

CONCLUSION

This study provides an efficient way for accessing shared memory by the multiple threads using deadlock monitor. Deadlock monitor can be used to monitor the thread process regularly. Monitor will reduce the deadlock occurrences. Thread map will be used for Dynamic Deadlock Detection Method. Also, it reduces the cost of accessing memory and improves the efficiency of multithreaded applications.

REFERENCES

Agarwal, R., S. Bensalem, E. Farchi, K. Havelund and Y. Nir-Buchbinder *et al*., 2010. Detection of deadlock potentials in multithreaded programs. J. Res. Dev., Vol. 54. 10.1147/JRD.2010.2060276.

Bodden, E. and K. Havelund, 2010. Aspect-oriented race detection in Java. IEEE Trans. Software Eng., 36: 509-527.

Chen, K.Y., J.M. Chang and T.W. Hou, 2011. Multithreading in Java: Performance and scalability on multicore systems. IEEE Trans. Comput., 60: 1521-1534.

Flanagana, C. and S.N. Freund, 2006. Type inference against races. Sci. Comput. Program., 64: 140-165.

Flanagana, C. and S.N. Freund, 2008. Atomizer: A dynamic atomicity checker for multithreaded programs. Sci. Comput. Program., 71: 89-109.

Hasanzade, E. and S.M. Babamir, 2012. An artificial neural network based model for online prediction of potential deadlock in multithread programs. Proceedings of the 16th CSI International Symposium on Artificial Intelligence and Signal Processing, May 2-3, 2012, Shiraz, Iran, pp: 417-422.

Joshi, P., C.S. Park, K. Sen and M. Naik, 2009. A randomized dynamic program analysis technique for detecting real deadlocks. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 15-21, 2009, Dublin, Ireland.

Luo, Z.D., R. Das and Y. Qi, 2011. Multicore SDK: A practical and efficient deadlock detector for real-world applications. Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation, March 21-25, 2011, Berlin, Germany, pp: 309-318.

Radoi, C. and D. Dig, 2013. Practical static race detection for java parallel loops. Proceedings of the International Symposium on Software Testing and Analysis, July 15-20, 2013, Lugano, Switzerland.

Wen, Y., J. Zhao, M.H. Huang and H. Chen, 2011. Towards detecting thread deadlock in Java programs with JVM introspection. Proceedings of the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, November 16-18, 2011, Changsha, China, pp: 1600-1607.