To appear in The Symposium on Engineering Test Harness 2013 co-located with The 13th International Conference on Quality Software (QSIC 2013)

Taming Deadlocks in Multithreaded Programs

Yan Cai
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
ycai.mail@gmail.com

W.K. Chan
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

Y.T. Yu
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
csytyu@cityu.edu.hk

***Abstract*—Many real-world multithreaded programs contain deadlock bugs. These bugs should be detected and corrected. Many existing detection strategies are not consistently scalable to handle large-scale applications. Many existing dynamic confirmation strategies may not reveal detectable deadlocks with high probability. And many existing runtime deadlock-tolerant strategies may incur high runtime overhead and may not prevent the same deadlock from re-occurring. This paper presents the current progress of our project on dynamic deadlock detection, confirmation, and resolution. It also describes a test harness framework developed to support our proposed approach.**

Keywords—deadlock; healing; detection; multithreaded programs

I. Introduction

Multithreaded programs may contain deadlock bugs. Detecting these bugs is critical in improving the dependability of these programs. Some deadlock bugs however may have not been fixed before the programs are used. Hence, it is necessary to develop a runtime resolution scheme to handle programs that may trigger deadlocks. This paper reports our progress in formulating and engineering a framework for dynamic detection, confirmation, and resolution of deadlock bugs.

Our framework includes components for lock trace reduction, object abstraction, deadlock detection, dynamic deadlock confirmation, and deadlock resolution. These components are briefly presented from Section II to Section VI, respectively. In Section VII, we present the current progress in the test harness implementation of our framework. Section VIII discusses the related work and our work. Section IX concludes the paper.

II. Lock Trace Reduction

A dynamic deadlock detection technique (Section IV) analyzes an execution trace to find potential deadlock scenarios. We have developed *LOFT* [4] to clean up certain locking operation records in such a trace (e.g., to alleviate data race detection [3]). A slight adaption of *LOFT* (by associating the execution contexts of the removed locking operations to the remaining lock operations) is applicable to the trace reduction for dynamic deadlock detection.

TABLE I. Memory and Time Comparisons among *iGoodlock*, *MulticoreSDK* (*MSDK*), and *Magiclock* (taken from [5])

Benchmark	Memory (MB)			Time in second (s)		
	iGoodlock	*MSDK*	*Magiclock*	*iGoodlock*	*MSDK*	*Magiclock*
SQLite	1.05	1.05	1.05	0.002	0.003	0.002
MySQL	>2800	1.15	1.05	>125	398	1.73
Chromium	>2800	>48.2	8.01	>6420	>3600	102
Firefox	>2800	122.41	4.14	>640	7.43	3.06
OpenOffice	245.20	>48.4	8.01	6360	>3600	0.67
Thunderbird	298.83	40.09	4.15	973	4.75	1.18

III. Object Abstraction

Effective deadlock confirmation relies on an effective modeling of the same object across multiple execution traces. We observe that in general, in an execution trace, the same program statement associating with the same call stack may be exercised multiple times (e.g., in a loop), and yet only some of them may be involved in a deadlock. On top of an existing stack-based object abstraction strategy [10][11], we have formulated the *Object Frequency Abstraction* (*OFA*) [6], which considers different occurrence counts of the same object type. This enhancement is to reduce the potential mismatches among different objects in the same trace or across multiple traces.

IV. Dynamic Detection

In an execution trace of a real-world program, only a small fraction of all lock dependencies [5] may be involved in cyclic chains; otherwise, the program may contain many deadlock bugs. We have proposed *Magiclock* [5] to firstly reduce the set of lock dependencies by removing irrelevant locks and then searching over the reduced set of lock dependencies to detect cyclic chains using a partition-based strategy. TABLE I shows that *Magiclock* can be more promising than *MulticoreSDK* [13] and *iGoodLock* [10] in analyzing large-scale benchmarks. Fig. 1 also depicts the relationship between *LOFT* and *Magiclock*.

V. Dynamic Confirmation

The component for deadlock confirmation aims at triggering deadlocks based on the detected cyclic chains. A preliminary version has been formulated as the *MagicScheduler* tool, which is a part of *MagicFuzzer* [5]. *MagicScheduler* is a randomized rescheduling approach. It uses the cyclic chains annotated with object abstractions as inputs, and monitors the trace until an event matching with some object abstraction of the given cyclic chains is detected. It then discards other cyclic chains and focuses on triggering deadlocks for the matched or partially matched cyclic chain(s).

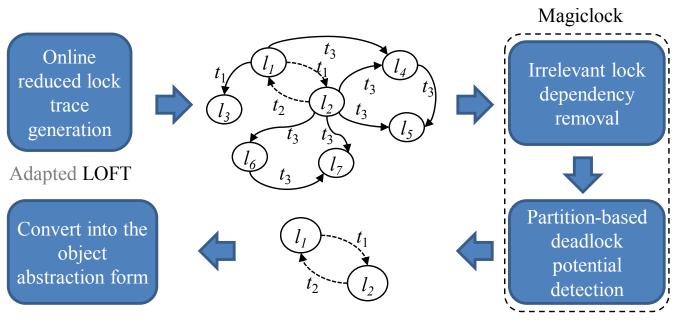

Magiclock
Online reduced lock trace generation
Adapted LOFT
Irrelevant lock dependency removal
Partition-based deadlock potential detection
Convert into the object abstraction form
l_1 l_2 l_3 l_4 l_5 l_6 l_7
t_1 t_2 t_3

Fig. 1. *LOFT* and *Magiclock*

VI. Dynamic Resolution

Deadlocks may also be resolved at runtime. In this category, existing approaches [11][14][15] often aim to serialize the execution of the program portion involved in deadlocks. To ease our presentation, we refer to a lock involved in a deadlock as a *wait-lock*, and a thread involved in the same deadlock as a *wait-thread*. We have not constructed this component.

We plan to actively assign the corresponding wait-lock of a wait-thread to the wait-thread when the wait-thread acquires any wait-lock of a deadlock. We conjecture that this active lock assignment strategy resolves the circular waiting condition for deadlock, which is also illustrated as a Java program shown in Fig. 2(a) because many programming languages (e.g., Java and C#) themselves support reentrant locks. This feature allows the same thread to successfully acquire the same lock that the thread is holding. Hence, a pre-acquisition of a wait-lock by a wait-thread is to avoid blocking the thread's acquisition of the wait-lock at the deadlocking site.

Our strategy suggests two possible solutions for the scenario in Fig. 2(a). They are the pre-acquisitions of the locks `B` and `A` by the threads t_1 and t_2, respectively, as shown in Fig. 2(b) and Fig. 2(c). For example, for the solution in Fig. 2(b), during execution, two threads will compete for the lock acquisition on the lock `B`. After one of the two threads firstly acquires the lock `B`, another thread has to wait until one thread exits from the execution protected by the lock `B`. For the thread t_1, though it needs to acquire the lock `B` twice, the deadlock is prevented. We note that even though the thread t_1 has a lock order from `A` to (the second acquisition of) `B` which still violates the lock order in the thread t_2, the order is protected by its first one from (the first acquisition of) `B` to `A` which is the same as that of the thread t_1. (The gate-lock resolution strategy proposed in [14] is inapplicable if there is a pair of `wait()-notify()` statements between t_1 and t_2 within the inserted gate-lock block.)

Several technical challenges still exist: A pre-acquisition of a lock by a thread may alter the original lock acquisition order of the program, introducing new deadlocks. We plan to analyze the lock dependency set to determine whether some potential solutions are undesirable and avoid generating them. The involved static or dynamic analysis on such a set could be imprecise. We also plan to devise a novel dynamic lock retreat strategy to release the actively pre-acquired wait-lock from a wait-thread to resolve "thrashing" [10]. However, it may not be generally feasible to pre-acquire a wait-lock at the boxed positions as illustrated by Fig. 2. Multiple deadlocks may interfere with one another. A further investigation is necessary.

A deadlock resolution strategy could be applied to either binary or source code of the program. It depends on the testing harness support which needs to be further studied.

VII. Engineering the Test Harness

We are building the test harness implementation of our framework. To handle C/C++ programs on the Linux platform, we build the implementation on top of the `Pin` tool [12] using its Probe mode. The implementation also maintains the relationship between parent thread and child thread (when a new thread is created) to support dynamic analysis. This information is useful for techniques that need to track the happens-before relationship between threads. For instance, in data race detection, the vector clock of a parent thread should be known

Thread t_1: `sync(A){ sync(B){…}}`
Thread t_2: `sync(B){ sync(A){…}}`
(a) A simple deadlock code illustration

Thread t_1: $\boxed{\texttt{sync(B)\{}}$ `sync(A){ sync (B){…}}`$\boxed{\texttt{\}}}$
Thread t_2: `sync (B){ sync(A){}}`
(b) Pre-acquisition of the lock `B` by thread t_1

Thread t_1: `sync(A){ sync(B){…}}`
Thread t_2: $\boxed{\texttt{sync(A)\{}}$ `sync(B){ sync(A){…}}`$\boxed{\texttt{\}}}$
(c) Pre-acquisition of the lock `A` by the thread t_2

Fig. 2 Two potential solutions in (b) and (c) may handle the deadlocking scenario illustrated in (a). The dynamically inserted codes are $\boxed{\textbf{boxed}}$.

Algorithm 1: Call Stack Computation

```
 1 void getCallStack
   (uint sp, uint ebp, int depth, vector<uint>& st){
 2    uint eip, insPtr; RTN rtn; int cnt = 0;
 3    insPtr = (uint) *(uint*) sp;
 4    while (cnt++ < depth && sp <= ebp) {
 5      rtn = RTN_FindByAddress(insPtr);
 6      if (RTN_Valid(rtn)) {
 7       uint addrImgLow =
 8         IMG_LowAddress(SEC_Img(RTN_Sec(rtn)));
 9        uint rtnAddr = RTN_Address(rtn);
10        st.push_back(rtnAddr - addrImgLow);
11      } else break; //end of if-else
12      eip    = ebp + sizeof(uint*);
13      insPtr = (uint) *(uint*) eip;
14      ebp    = (uint) *(uint*) ebp;
15    }//end of while
16 }
```

to a child thread right after the parent thread creates the child thread before the execution of the latter thread [4]. Similarly, for deadlock detection, one can use such happens-before relationship information to eliminate false warnings [1].

The implementation of a Pthread in a Linux system is to use the system call `clone` to create a new thread and then invoke the system call `start_thread` to start the newly created thread, where `clone` is called in the parent thread and `start_thread` is called in the child thread. An argument of either function is the *memory address* of the newly created thread. Our testing harness implementation thus produces two customized events (`preCloneThread` and `preStartThread`), respectively whenever it monitors the occurrences of these two system calls. It then uses the memory address of the child thread (`child_addr`) to relate these two events to model the parent-child relationships by checking the values in the *memory address* before and after a child is created.

Our test harness implementation also computes the object abstraction because `Pin` offers no API to index the same thread/lock across multiple execution traces. Moreover, it computes an object abstraction for each thread and for each lock whenever the thread or the lock is created. It furthermore computes a call stack and a counter (hash value) mapped from each creation event.

The algorithm to compute the required call stack for an object abstraction is shown in Algorithm 1. Given a program *`sp`* (stack top pointer) and *`ebp`* (stack base pointer), the algorithm iteratively searches for the valid call functions (lines 3 and 12–14), computes their relative addresses (lines 7–9), and saves them into the given vector instance (line 10). The `depth` argument is used to limit the length of the required stack (line 4).

Fig. 3 shows an example program containing one resource deadlock (at lines 5, 6 and 11, 12). When executing this program, our test harness implementation generates two log files

```
1    pthread_mutex_t lock1, lock2;
2    pthread_t child1, child2;
3    void* child( void* arg ){
4       if( 1 == (int)arg ){
5           pthread_mutex_lock(&lock1);
6           pthread_mutex_lock(&lock2);
7           pthread_mutex_unlock(&lock2);
8           pthread_mutex_unlock(&lock1);
9       } else {
10          sleep(1000);
11          pthread_mutex_lock(&lock2);
12          pthread_mutex_lock(&lock1);
13          pthread_mutex_unlock(&lock1);
14          pthread_mutex_unlock(&lock2);
15      }
16   }
17   int main ( void ) {
18      pthread_mutex_init(&lock1, NULL);
19      pthread_mutex_init(&lock2, NULL);
20      pthread_create(&child1,NULL,child,(void*)1);
21      pthread_create(&child2,NULL,child,(void*)2);
22      pthread_join(child2, NULL);
23      pthread_join(child1, NULL);
24      return 0;
25  }
```

Fig. 3. An example program with a real deadlock

TABLE II. A fragment of the execution trace for the code shown in Fig. 3. We assign an identical integer to each thread and each lock and there are two map files from an integer mapped to an abstraction of a thread or a lock. Here we have omitted these two map files due to page limitation.

thread 1.log //child1	**thread 2.log** //child2
1, <stack1, 1>, 2, <stack1, 2>	1, <stack2, 1>, 2, <stack2, 2>

(one for each thread) as shown in TABLE II, in which each line in a log file has the following format: {*lockID*, ⟨*call_stack*, *counter*⟩}$^+$, which can be used to derive a particular lock dependency [5]. For simplicity, we do not show those lock dependencies containing an empty lockset.

The test harness (which implements *Magiclock*) reads these files and detects one cyclic lock dependency chain from the trace, and TABLE II shows the result ⟨⟨1, lock1, {lock2}⟩, ⟨2, lock2, {lock1}⟩⟩, whose abstraction is shown in Fig. 2 (a).

Consider the example illustrated in Fig. 2 (a). The test harness implementation (which implements *MagicScheduler*) firstly reads this cyclic dependency chain, and then schedules the thread execution of the example program over the same input. In such a confirmation execution, when the thread `child1` is about to execute the statement at line 6 or the thread `child2` is about to execute the statement at line 12, the test harness suspends the execution of every thread, and check whether a real deadlock has been triggered. As such, the cyclic dependency chain is confirmed as a real deadlock and is reported to `standard output` by the test harness implementation.

Our current framework implementation instruments the program under test using `Pin` [12] to generate lock traces. Then it analyzes each lock trace to find deadlock potentials. For each reported potential deadlock, our implementation executes the given program with the same test case and confirms the deadlock potential if it is a real deadlock. Our current implementation has not been able to resolve the detected deadlocks yet.

In the rest of this section, we demonstrate how to use the current test harness to detect and confirm real deadlocks. They have been packaged as the *MagicFuzzer* tool. The tool can be invoked in three phases, which we refer to as Phase I, Phase II, and Phase III, respectively, in this section.

In the sequel, we suppose that a tester has installed the `Pin` framework and placed it at the directory denoted by PIN_HOME. (We have tested this tool on Pin 2.10 (45467) on a 32-bit Linux system.) The tool can be downloaded from the URL: *http://www.cs.cityu.edu.hk/~51948163/magicfuzzer/*

To invoke Phase I of *MagicFuzzer*, the tester types the following command, where `traceDIR` is the name of the folder which keeps the execution trace(s) generated by *MagicFuzzer*.

```
PIN_HOME/pin -t ./MagicFuzzer.so traceDIR --
Your_Program Program_Arguments
```

To invoke Phase II, the tester should execute *Magiclock*:

```
./Magiclock traceDIR
```

If the execution generated by Phase I contains any cyclic dependency chain, the tester will find these chains in the file: `traceDIR/Deadlock.dlk` (which is a text file).

The tester can further invoke Phase III (i.e., *MagicScheduler*) of *MagicFuzzer* to confirm whether any cyclic chain may be manifested into a real deadlock by the following command:

```
PIN_HOME/pin -t ./MagicScheduler.so traceDIR --
Your_Program Program_Arguments
```

If a cyclic dependency chain is confirmed as a real deadlock, *MagicFuzzer* reports the deadlock to the standard output.

Fig. 4 shows a sequence of images and related operations of the test harness implementation that illustrate the actual operations of the implementation. The benchmark that we used in the demonstration is *hawknl*, which was taken from the following site: *http://code.google.com/p/dimmunix/*.

The first image illustrates Phase I and the command is highlighted in bold. Similarly, the second image illustrates Phase II, and the last image illustrates Phase III. The third image shows the result of identifying the parent-child relationships among threads, as discussed in Section II of this paper. The fourth image shows a potential deadlock.

VIII. Discussion and Related Work

Several kinds of object abstractions have been proposed in the literature, including the *k*-object-sensitivity abstract [10], and call stack based abstraction [11], for example.

MulticoreSDK [13] and *iGoodlock* [10] have been compared to *Magiclock* [5] as shown in TABLE I of this paper for dynamic deadlock detection.

In [2], an experiment was conducted using *PCT*, a purely randomized scheduler tool which is unaware of bug types. It shows that the probability of detecting deadlocks by *PCT* can be an order of magnitude lower than 1%, even though *PCT* suffers from no thrashing. *ConTest* [7] simply injects arbitrary timeouts to alter the thread schedule with the intent to trigger deadlocks only with empirical data to back up the claim on effectiveness. *DeadlockFuzzer* [10] was shown to be effective in triggering deadlock on small to medium-scale benchmarks.

We have reviewed how we use `Pin` to perform dynamic deadlock analysis (including trace generation, detection, confirmation, and resolution) for C/C++ multithreaded programs. For Java programs, there are other tools, such as *Calfuzzer* [9] and *RoadRunner* [8]. Unlike `Pin` [12], these tools instrument the whole Java program, producing many irrelevant events.

Nir-Buchbinder et al. [14] proposed inserting a gate lock right before each thread involved in a deadlock acquires its problematic lock. Nonetheless, their method cannot handle some non-trivial resource deadlocks, and may introduce new deadlocks due to gate lock insertions. *Deadlock Immunity* [11] prevented the second occurrence of a deadlock by recording the pattern of its first occurrence and matching the pattern in later executions. Such a pattern matching strategy is imprecise, failing to avoid deadlocks from re-occurrence even though it suffers from a light slowdown factor (e.g., 15% [11]). *Gadara* [15] detects potential deadlocks offline. At runtime, any matched cycle along the run will trigger a serialization of the corresponding deadlock potential code; however, many such occurrences in the same run may prolong the run significantly.

IX. Conclusion

In this paper, we have reviewed our current progress in building a test harness for dynamic deadlock detection, confirmation, and resolution. We have described the techniques for individual components of the test harness, and discussed some standard elements across similar test harness implementations. These standard elements include maintaining the relationship between parent thread and child thread and obtaining the call stack and using it for further computation. We have also described how to use our current harness tool.

Acknowledgment

This work is supported in part by Strategic Research Grant (project no. 7002873) from City University of Hong Kong and General Research Fund (project no. 111410) and Early Career Scheme (project no. 123512) from Hong Kong Research Grants Council.

References

[1] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proc. HVC'05*, 208–223, 2005.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. ASPLOS'10*, 167–178, 2010.

[3] Y. Cai and W.K. Chan. LOFT: Redundant synchronization event removal for data race detection. In *Proc. ISSRE'11*, 160–169, 2011.

[4] Y. Cai and W.K. Chan. Lock trace reduction for multithreaded programs. In *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), 2013, to appear.

[5] Y. Cai and W.K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In *Proc. ICSE'12*, 606–616, 2012.

[6] Y. Cai, K. Zhai, S.R. Wu, and W.K. Chan. TeamWork: synchronizing threads globally to detect real deadlocks for multithreaded programs. In *Proc. PPoPP'13*, poster article, 311–312, 2013.

[7] E. Farchi, Y. Nir-Buchbinder, and S. Ur. A cross-run lock discipline checker for Java. In *PADTAD'05*, 2005.

[8] C. Flanagan and S.N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proc. PASTE'10*, 1–8, 2010.

[9] P. Joshi, M. Naik, C.S. Park, and K.Sen. CalFuzzer: an extensible active testing framework for concurrent programs. In *Proc. CAV'09*, 675–681, 2009.

[10] P. Joshi, C.S. Park, K. Sen, amd M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. PLDI'09*, 110–120, 2009.

[11] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proc. OSDI'08*, 295–308, 2008.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI'05*, 191–200, 2005.

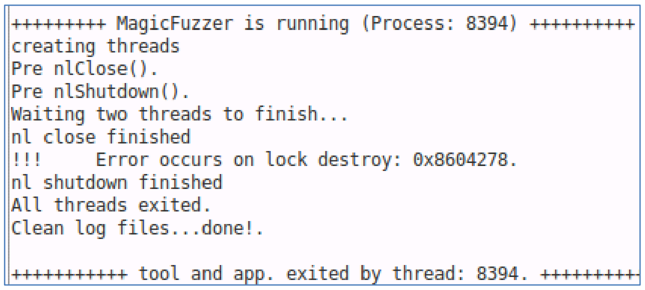
[13] Z.D. Luo, R. Das, and Y. Qi. MulticoreSDK: A practical and efficient deadlock detector for real-world applications. In *Proc. ICST'11*, 309–318, 2011.

[14] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In *Proc. RV'08*, 104–118, 2008.

[15] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In *Proc. OSDI'08*, 281–294, 2008.

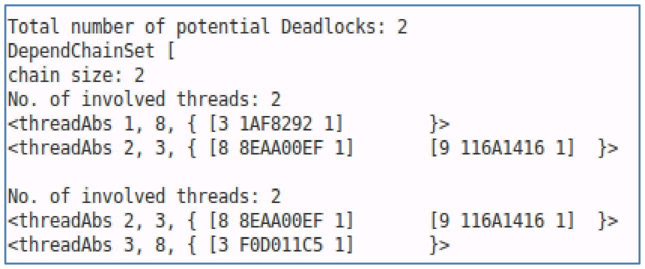
(1) *Command*

$pin -t ./MagicFuzzer.so ./demoTrace --./test/ hawknl/deadlock3



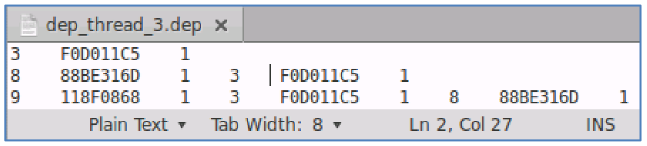
```
+++++++++ MagicFuzzer is running (Process: 8394) ++++++++++
creating threads
Pre nlClose().
Pre nlShutdown().
Waiting two threads to finish...
nl close finished
!!!     Error occurs on lock destroy: 0x8604278.
nl shutdown finished
All threads exited.
Clean log files...done!.

+++++++++++ tool and app. exited by thread: 8394. +++++++++
```

(2) *Command* **$./Magiclock ./demoTrace**



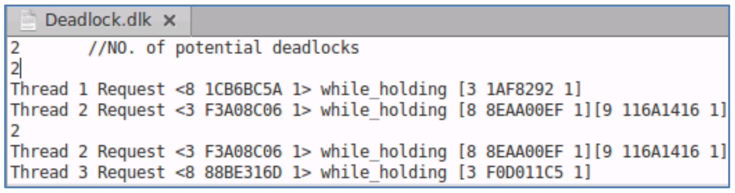
```
Total number of potential Deadlocks: 2
DependChainSet [
chain size: 2
No. of involved threads: 2
<threadAbs 1, 8, { [3 1AF8292 1]        }>
<threadAbs 2, 3, { [8 8EAA00EF 1]       [9 116A1416 1]  }>

No. of involved threads: 2
<threadAbs 2, 3, { [8 8EAA00EF 1]       [9 116A1416 1]  }>
<threadAbs 3, 8, { [3 F0D011C5 1]       }>
```

(3) Example dependencies of a thread obtained



```
3    F0D011C5    1
8    88BE316D    1    3    F0D011C5    1
9    118F0868    1    3    F0D011C5    1    8    88BE316D    1
```

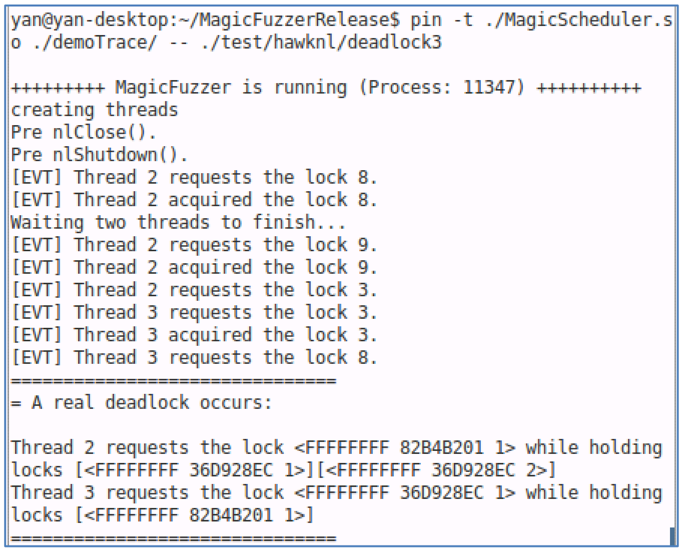
(4) Detected potential deadlock.



```
2       //NO. of potential deadlocks
2
Thread 1 Request <8 1CB6BC5A 1> while_holding [3 1AF8292 1]
Thread 2 Request <3 F3A08C06 1> while_holding [8 8EAA00EF 1][9 116A1416 1]
2
Thread 2 Request <3 F3A08C06 1> while_holding [8 8EAA00EF 1][9 116A1416 1]
Thread 3 Request <8 88BE316D 1> while_holding [3 F0D011C5 1]
```

(5) *Command*

$pin -t ./MagicScheduler.so ./demoTrace -- ./test/ hawknl/deadlock3



```
yan@yan-desktop:~/MagicFuzzerRelease$ pin -t ./MagicScheduler.s
o ./demoTrace/ -- ./test/hawknl/deadlock3

+++++++++ MagicFuzzer is running (Process: 11347) ++++++++++
creating threads
Pre nlClose().
Pre nlShutdown().
[EVT] Thread 2 requests the lock 8.
[EVT] Thread 2 acquired the lock 8.
Waiting two threads to finish...
[EVT] Thread 2 requests the lock 9.
[EVT] Thread 2 acquired the lock 9.
[EVT] Thread 2 requests the lock 3.
[EVT] Thread 3 requests the lock 3.
[EVT] Thread 3 acquired the lock 3.
[EVT] Thread 3 requests the lock 8.
===============================
= A real deadlock occurs:

Thread 2 requests the lock <FFFFFFFF 82B4B201 1> while holding
locks [<FFFFFFFF 36D928EC 1>][<FFFFFFFF 36D928EC 2>]
Thread 3 requests the lock <FFFFFFFF 36D928EC 1> while holding
locks [<FFFFFFFF 82B4B201 1>]
===============================
```

Fig. 4. A sequence of images for illustration of the three phases of the operations of *MagicFuzzer*