Deadlocks as Runtime Exceptions

Rafael Brandao Lobo and Fernando Castor

Center of Informatics, Federal
University of Pernambuco, Brazil
rbl,castor@cin.ufpe.br
http://www.springer.com/lncs

Abstract. ...

Keywords: ...

1 Introduction

...

2 Protocols

In this chapter, we present the deadlock detection algorithm divided in three parts. In the first part, an overview of the protocol is described and we also present proof that this protocol is sufficient to detect deadlocks between 2 threads and 2 locks (in short, we will call it *2-deadlock*). We further change the protocol to guarantee that exception is raised on both threads. Finally we show pseudocode of the actual implementation we developed on this research.

2.1 Protocol: Deadlock Detection

We have modified the default implementation of Java's *ReentrantLock* to allow efficient runtime 2-deadlock detection. We take advantage of the current algorithm and some of its guarantees to avoid the need to introduce extra synchronization mechanisms or costly atomic operations.

1. Each lock has a pointer for a thread which is the current owner or null when there's no thread owning that lock.
2. Each lock has an integer to represent its current state: 0 means the lock is free and no thread owning it (the *unlocked* state), 1 means there's a thread owning the lock (the *locked* state). For simplicity, we are only interested on these two states and its change holds the most complexity, but in the implementation of *ReentrantLock* each time the thread owner acquires the same lock, this state would be incremented, and decremented each time the thread releases it.

3. Each thread has a thread-local list of pointers of locks they are currently owning.
4. Each lock has a waiting queue of threads that are waiting to acquire it. Whenever a thread try to obtain a lock when it's already acquired, the thread will add itself on the waiting queue before parking. Upon the event of releasing the lock, the owner of that lock will look for the first thread in the waiting queue and unpark it.
5. When a thread wants to acquire a lock, it will swap the current state to *locked* if the current state is *unlocked* atomically.
   (a) If the thread fails, it must be because the lock is already owned by some other thread, then it will add itself on the waiting queue for that lock. Finally, the thread will park.
   (b) Otherwise, the thread will set itself as the current owner of that lock and also add this lock to its thread-local list of pointers of locks it owns.
6. When a thread is about to release a lock, the current owner pointer of that lock is set to null and that lock is also removed from the thread-local list of owned locks. Finally, the lock state is changed to *unlocked.*
7. Before parking, a thread will check whether there's deadlock. When the current thread is unable to acquire its desired lock, it must be because another thread is owning it already. It is possible to know who is the owner of any lock, so the current thread identifies the owner of its desired lock as the conflicting thread. Then the current thread will search on each lock of its thread-local list of owned locks if the conflicting thread is waiting on it.
   (a) If positive, then we have a circular dependency (current thread is stuck waiting its desired lock and the conflicting thread is stuck waiting for a lock the current thread owns) thus a deadlock exception will be raised.
   (b) Otherwise, the thread parks.

Assumptions This protocol's correctness relies on a few guarantees provided by Java's *ReentrantLock* class on its default implementation.

1. The operation of swapping the state of a lock from *unlocked* to *locked* must be done atomically by the thread, so only one thread can be successful at a time.
2. A thread will only park when it's guaranteed some other thread can unpark it. Missing notifications will never happen and concurrent uses of park and unpark on the same thread will be resolved gracefully.
3. Inserts on each lock's waiting queue must be done atomically. If multiple threads concurrently attempt to insert themselves in the waiting queue on the same lock, they will both succeed eventually but the exact order of insertions is not important.
4. Once the last element in the waiting queue of a lock is read, it should be safe to read all threads in the waiting queue that arrived before the last element. Since the thread who reads the waiting queues is also the one who blocks every thread waiting on the queues, we can guarantee the only updates that could happen concurrently is new insertions at the end of each queue.

However insertions in the end of the queue are not important once a last element pointer is obtained.

2.2 Formal Proof

On this subsection, we proof this protocol is sufficient to detect 2-deadlocks. First, we show a proof for the *liveness* property which states we can always detect 2-deadlocks when they happen. Lastly, we show a proof for the *safety* property which states we never throw exceptions when 2-deadlocks doesn't really happen.

Lemma 1. *Protocol can always detect deadlock when a 2-deadlock happens.*

Proof. Suppose not and a 2-deadlock occured without deadlock exception being raised. Let's assume that threads A and B have both acquired locks a and b respectively, as follows:

$$write_A(state_a = locked) \rightarrow write_A(owner_a = A) \tag{1}$$

$$write_B(state_b = locked) \rightarrow write_B(owner_b = B) \tag{2}$$

And now each thread will attempt to acquire the oppositing lock: thread A is trying to acquire lock b and thread B is trying to acquire lock a, as follows:

$$read_A(state_b == locked) \rightarrow write_A(waiting\_queue_b.insert(A)) \tag{3}$$

$$read_B(state_a == locked) \rightarrow write_B(waiting\_queue_a.insert(B)) \tag{4}$$

If a 2-deadlock happened, then both threads are now parked and all previous equations should be correct. But before parking, each thread must check for deadlock by inspecting each lock it owns if the oppositing thread is on its waiting queue. As we initially assumed no deadlock exception has been raised, then both threads are parked and also the following equations must be correct:

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == false) \tag{5}$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == false) \tag{6}$$

The problem with the previous equations is that they both cannot be true simultaneously. Before checking for deadlock, each thread must add itself on the waiting queue of its desired lock. If it holds that the oppositing thread is not in the waiting queue yet, then it must be because it did not start to check for deadlock yet, thus a contradiction.

Lemma 2. *Protocol never throw a deadlock exception for a non-existent 2-deadlock.*

Proof. Suppose the opposite: a deadlock exception was raised and there's no real 2-deadlock. At least one of the following equations must be true in order to raise a deadlock exception:

$$read_A(owner_b == B) \rightarrow read_A(waiting\_queue_a.contains(B) == true) \tag{7}$$

$$read_B(owner_a == A) \rightarrow read_B(waiting\_queue_b.contains(A) == true) \quad (8)$$

Suppose without loss of generality the first equation is correct. It means thread B is waiting for lock a and it is also the owner of lock b. If it is on the waiting queue, that thread is either parked already or about to park and in both cases it means thread B is going to depend on the release of lock a to proceed. However, as we have seem previously, thread A at this point is also about to park and is checking for a deadlock. If this condition holds, we have a circular dependency between threads A and B, a real 2-deadlock, thus we have a contradiction.

The only problem with this protocol is the lack of guarantee that both threads involved in a 2-deadlock will throw deadlock exception. If both threads are about to park and are both running the deadlock detection procedure, then the equations 7 and 8 will both be true and deadlock exception will be raised by both threads. However, it is possible that one of the threads did not finish inserting itself on the waiting queue for the lock it desires, then the conflicting thread will hit the case when one of the equations 7 or 8 will be false, thus not throwing a deadlock exception.

2.3 Protocol: Exception Raised On Both Threads

We have further extended the previous protocol to allow both threads involved in the deadlock to throw deadlock exceptions. This does not affect how deadlock is detected but what should be done after a deadlock is detected.

1. Each lock has a list of tainted threads. This list should only be read or updated by the owner of that lock, allowing immunity from interference without any extra synchronization cost.
2. Once a deadlock is detected and the current thread is about to raise a deadlock exception, it already knows: which thread is conflicting with itself; and which lock that thread is desiring. Then the current thread (the owner of the desired lock) will add this conflicting thread in tainted threads list for that lock. After that, deadlock exception is raised.
3. When the conflicting thread is unparked and finally acquires its desired lock (it becomes the owner of that lock), then it is allowed to read the list of tainted threads. If this thread identifies itself into this list, then it must be because it was part of a deadlock before, so it removes its reference from the list and also raise a deadlock exception.
4. Every operation on the list of tainted threads of any locks (either reading or inserting values) should be followed up by some cleanup on all references of threads that are no longer running.

This is sufficient to force both threads to throw exceptions when only one of them would raise an exception in the initial protocol. However when they both would raise an exception anyway, then this change introduces a different problem: dangling references.
Each thread would have added their conflicting thread on its owned locks's

tainted threads list, but none of them would be able to acquire their respectives desired locks (as in *item 3*), thus leaving their references behind for others to cleanup (as in *item 4*).

2.4 Implementation

In this subsection we present pseudocode for the proposed protocols. Initially we present the pseudocode for the current implementation of *ReentrantLock*, then we present which changes were done on top of that implementation to follow the protocols covered previously. The actual code can be found on our repository (TODO: put code github and reference).

Pseudcode of Java's ReentrantLock

```
int state;
Thread owner;
Node head;
Node tail;

void lock() {
  if (!tryFastAcquire()) {
    slowAcquire();
  }
}

boolean tryFastAcquire() {
  if (!hasQueuedPredecessors() && COMPARE_AND_SET(state, 0, 1)) {
    setExclusiveOwner(currentThread());
    return true;
  }
  return false;
}

// Returns true if current thread
// is the first in the queue or it's empty
boolean hasQueuedPredecessors();

void setExclusiveOwner(Thread thread) {
  owner = thread;
}

void slowAcquire() {
  // Creates and atomically enqueue node with current thread
  Node waiterNode = new Node();
  enq(waiterNode);

  // Try a few times to acquire the waiterNode and then park
  // until its predecessor wakes up this thread
  boolean failed = true;
  try {
    while (true) {
      if (waiterNode.pred == head && tryFastAcquire())) {
        setHead(waiterNode);
        failed = false;
        return;
      }
      if (shouldParkAfterFailedAcquire(waiterNode.pred, waiterNode))
        park();
    }
  } finally {
    if (failed)
      cancelAcquire(waiterNode);
  }
}

void release() {
  if (tryRelease()) {
    unparkSuccessor(head);
  }
}

boolean tryRelease(int releases) {
  if (currentThread() != owner)
    return false;
  setExclusiveOwner(null);
  setState(0);
  return true;
}

void park() {
  LockSupport.park(this);
}

// Wakes up successor of a given node in the waiting queue
// if necessary by using LockSupport.unpark on its successor.
void unparkSuccessor(Node);

// Cancel the waiting node and remove from waiting queue.
// If there's a successor parked, unpark it.
void cancelAcquire(Node);

// Atomically checks if the node is really the head
// of the queue and try fastPath codepath.
// On success, the node is dequeued from queue
bool tryFastAcquireIfHead(Node);

// Atomically enqueues node in the waiting queue. It repeately
// tries to COMPARE_AND_SET to update tail until succeeds.
// If head and tail are not initialized yet, there will be
// an extra COMPARE_AND_SET on head to a new Node and tail
// will be set as head.
void enq(Node);

// Make sure to park only when is guaranteed an unpark signal
// can be received. It decides based on specific protocol
// between predecessor of a given node and that node.
shouldParkAfterFailedAcquire(Node, Node);

// Returns Thread corresponding to the current thread
Thread currentThread();

// Disables the current thread for thread scheduling
// purposes unless the permit is available.
LockSupport.park();

// Makes available the permit for the given thread,
// if it was not already available.  If the thread
// was blocked on park then it will unblock.
LockSupport.unpark(Thread);
```

Changes on ReentrantLock

```
// This is a thread-local inside a lock.
// It keeps the list of locks the current thread owns.
DEFINE_PER_THREAD(vector<int>, ownedLocks);

// As soon as a lock is acquired, the thread sets itself
// as the owner. When it's released, the owner is set to null.
void setExclusiveOwner(Thread thread) {
  owner = thread;
  if (owner == null) {
    unregisterOwnedLock();
  } else {
    registerOwnedLock();
  }
}

// These functions register or unregister the currnet lock
// in the thread-local list ownedLocks.
registerOwnedLock();
unregisterOwnedLock();

void park() {
  Thread conflictingThread = owner;
  if (isAnyOwnedLockDesiredBy(conflictingThread)) {
    clearOwnedLocksByCurrentThread();
    throw new DeadlockException();
  }
  LockSupport.park(this);
}

// Returns true if any of the locks owned by the current thread
// contain a given thread in the waiting queue.
isAnyOwnedLockDesiredBy(Thread);

// Clear all locks in the list of owned locks by the current thread.
clearOwnedLocksByCurrentThread();
```

3 Evaluation

In this paper we have also empirically evaluated how effective deadlock exception can be.

We had two implementations of reentrant locks where one of them was the one provided by Java's *ReentrantLock* and the other was that lock modified to throw exception when a deadlock happened. Later we may refer to our implementation as *LockA* and the default one as *LockB*.

In order to compare each implementation, we conducted a controlled experiment where students had to run two specific programs with deadlocks easy to reproduce while collecting the time taken to identify the problem. They also had to provide a clear explanation of the problem, describing what the problem is, which method calls were involved on it and a description of how it happens, so we could measure answer precision.

3.1 Experiment Definition

The goal of our experiment was to analyze the process of bug identification with the purpose of evaluating efficiency of deadlock exceptions, in respect to the time spent in order to identify the problem and the accuracy of the descriptions provided by the students. We can define two research questions we want to answer in this experiment:

RQ1. Is the time spent to identify the bug reduced for implementation with deadlock exception when compared to the default implementation?

The metric we watched to answer this question was the time, in seconds, to finish each question in the test.

RQ2. Is the accuracy of bug description improved for implementation with deadlock exception when compared to the default one?

Each question's answer was splitted in a few criterias and each criteria was rated between 0 and 1, where 0 means not present, 0.5 means partially present and 1 for fully present:

A. Correctly classified problem as deadlock.
B. Classified problem as different from deadlock.
C. Correctly identified method calls involved in the deadlock.
D. Correctly identified locks involved in the deadlock.
E. Pointed unrelated methods as part of the deadlock.

To answer this research question, we have classified students answers as either correct or incorrect. Correct answers should respect the following equation:

$$(A - B) + C \geq 1.5 \tag{9}$$

We decided to rule out criterias D and E because the problem statement was not clear they should describe which locks were involved in the deadlock; also, our deadlock implementation at that time could only guarantee at least one deadlock exception to be thrown thus affecting at least one method. In other words, this equation means that a correct answer is whenever the bug was described as deadlock and at least one of the methods involved were identified.

3.2 Experiment Planning

In order to evaluate each element described on the previous section, we describe the following statistical hypotheses.

1) Hypothesis: To answer *RQ1* regarding the time spent to identify a bug in the code:

$$H_0 : \mu_{TimeLockA} \geq \mu_{TimeLockB} \tag{10}$$

$$H_1 : \mu_{TimeLockA} < \mu_{TimeLockB} \tag{11}$$

And to answer *RQ2* regarding accuracy of answers:

$$H_0 : \mu_{CorrectAnswersLockA} \leq \mu_{CorrectAnswersLockB} \tag{12}$$

$$H_1 : \mu_{CorrectAnswersLockA} > \mu_{CorrectAnswersLockB} \tag{13}$$

2) Design, Instrumentation and Subjects: For this empirical experiment, we have chosen two metrics: time to answer a question and number of correct answers.

In order to prevent *bias*, we needed to control a few factors during the experiment execution. The first factor was the selection of subjects to participate on this experiment, as different background knowledge could potentially influence chosen metrics. The second factor we had to control was the complexity of programs that each subject. Complexity we define as the amount of files in the program, number of threads and number of locks to analyze; as we assumed that easier programs could have little or no benefit from deadlock exceptions, we wanted to have one program that we considered easy to identify the problem and another that was way more complex, spread in more files and classes, thus reflecting a more realistic case. We provided implementations of each program using either *LockA* or *LockB*: the two possible treatments that we want to compare.

We decided to use Latin Square Design to control these two factors mentioned earlier: subjects and program complexity factors. Since we had N subjects, 2 programs and 2 possible treatments, we disposed subjects in rows and programs in columns of latin squares, randomly assigning in each cell of the square a treatment that could be *LockA* or *LockB*, but also guaranteeing that for any given row or column in this square, each treatment appears only once (see Table 1). Consequently, we have replication, local control and randomization which are the three principles of experiment design.

Table 1. Latin Square design

	Program 1	Program 2
Subject 1	LockA	LockB
Subject 2	LockB	LockA

We wrote two programs with different complexity which were presented in the same order for all subjects. The first program, known as *Bank*, contained 4 classes spread in 4 files, 3 threads, 3 explicit locks, and 82 lines of code in average. The second program, known as *Eclipse* had 15 classes spread in 11 files, 4 threads, 5 explicit locks, and 40 lines of code in average. We expected the first program to be easier to identify the deadlock because it contained fewer classes and files. Each program could use either *LockA* or *LockB* but we randomly assigned a group to each student so that if they fall into group A, they would start with *LockA* in the first question, but change to *LockB* on the second question; or if they fall in group B, they they would start with *LockB* and switch to *LockA* in the second question. We randomly paired subjects in tuples composed of one subject in group A and another subject of group B, then we created latin squares for each one of these pairs, where any remainders were discarded.

We have repeated this experiment for two groups of students with different backgrounds. The first group consisted of undergraduate students attending Programming Language Paradigms course. They had classes about concurrent programming, including exercises in Java using ReentrantLock where deadlocks and other concurrent bugs should be avoided; however, these students were not experienced in this area. The second group consisted of graduate students enrolled in master's degree or PhD program attending Parallel Programming course where they had classes about advanced concepts of parallel programming and had a lot of practical exercises, including implementing their own lock; thus, they were expected to have a lot of experience. We did a survey with the second group to understand their background even further (see charts below) at the end of the experiment.

3) Metrics Collection: Each one should start the experiment with the first question containing *Program 1* and once they finish to provide an answer, they should request for the second question. At that point, we collect and place a timestamp in their answer. Once they finish the second question containing *Program 2*, then they should again give us a notice so we can leave a new timestamp. We have used these timestamps to measure how long they took to finish each question. We have started this experiment with a time limit for each question of 60 minutes each. However, during the test we realized it could not be sufficient for all students so we expanded to 90 minutes each.

The timestamp was written by students conducting the experiment based on a counter we projected on the laboratory wall in real time. In a few circunstances the subject could write the timestamp when they finish, but we have double checked the value at the time we collected their answer, overwriting in case they did any mistake.

3.3 Experiment Operation

We executed this experiment in two different days. In the first day we did it with undergraduate students in replacement of their default exam, so their participation was obligatory but we disclaimed they could optionally leave a comment if they did not want to take part in this research, so we would not use their data. Fortunately no one chose to not participate. In the second day, we did it with graduate students after the last class of Parallel Programming course and it was optional. In total, 31 students participated on the first day and 16 students participated on the second day, but we had to discard 2 students data because they arrived late and they had to leave early.

On the first day we started with a time frame of 2 hours for the whole experiment, so we decided to set a deadline for each question and put a time limit of 1 hour each. Later we expanded the time limit to 1 hour 30 minutes for each question. On the second day we decided to stick with 1 hour each because there was no demand to extend it.

3.4 Experiment Results

We can split the experiment analysis in two parts:

1) Time Analysis:

2) Accuracy Analysis: Applying Fisher's exact test we can see that undergraduate students did receive an improvement on accuracy (see Table 2), while

Table 2. Graduate students ANOVA results

	Df	Sum Sq	Mean Sq	F value	Pr(¿F)	
replica	6	5413347	902225	5.1011	0.0080593	**
program	1	19032	19032	0.1076	0.7485379	
lock	1	4150300	4150300	23.4656	**0.0004022**	***
replica:student	7	4614230	659176	3.7270	0.0223149	*
Residuals	12	2122405	176867			

graduate students did not have any significant difference for the accuracy (see Table 3).

Table 3. Undergraduate students results presented a two-tailed P value equals 0.0004. The association between rows (groups) and columns (outcomes) is considered to be extremely statistically significant

	Correct	Incorrect
LockA	29	2
LockB	16	15

Table 4. Graduate students results presented a two-tailed P value equals 0.3259. The association between rows (groups) and columns (outcomes) is considered to be not statistically significant

	Correct	Incorrect
LockA	13	1
LockB	10	4

3.5 Results Interpretation

We can see that accuracy was indeed improved for unexperienced subjects but may not have any big impact in more experienced subjects, like we expected. For the time metric, however, it's still unclear whether it was indeed improved for unexperienced subjects, but we see significant improvement for more experienced subjects.

We believe that our imposed time limit have limited more drastically the time ranges on the first group because they spent more time on each question. Also the fact it was an exam for them may have delayed the time to answer because they were more careful. We have observed during the experiment that many students wrote their answers but they were reluctant to ask for the next question because they still have plenty of time left and they wanted to make sure it was correct. We did not observe such behavior with the second group of students and we believe it is because they did not have the same pressure to deliver correct results.

3.6 Threats To Validity

...

3.7 Attachments

Instructions in R to evaluate graduate students results

```
> exp1.dat = read.table(file="/Users/rafaelbrandao/r_input.dat", header = T)
> attach(exp1.dat)
>
> replica = factor(replica.)
> student = factor(student.)
> program = factor(program.)
> lock = factor(lock.)
> plot(time~lock,col="gray",xlab="Lock",ylab="Time(seconds)")
> anova.ql<-aov(time~replica+student:replica+program+lock)
> library(MASS)
> bc <- boxcox(anova.ql,lambda = seq(-3, 5, 1/10))
> lambda <- bc$x[which.max(bc$y)]
> lambda
[1] 0.959596
> TukeyNADD.QL.REP<-function(objeto1)
+ {
+ y1<-NULL
+ y2<-NULL
+ y1<- fitted(objeto1)
+ y2<- y1^2
+ objeto2<- aov(y2 ~ objeto1[13]$model[,2] +
+ objeto1[13]$model[,3]:objeto1[13]$model[,2]
+ + objeto1[13]$model[,4]+ objeto1[13]$model[,5])
+ ynew <- resid(objeto1)
+ xnew <- resid(objeto2)
+ objeto3 <- lm(ynew ~ xnew)
+ M <- anova(objeto3)
+ MSN <- M[1,3]
+ MSErr <- M[2,2]/(objeto1[8]$df.residual-1)
+
+ F0 <- MSN/MSErr
+ p.val <- 1 - pf(F0, 1,objeto1[8]$df.residual-1)
+ p.val
+ }
> TukeyNADD.QL.REP(anova.ql)
[1] 0.6863746
> anova(anova.ql)
```

References

...