By John Mount, Dr. Dobb's Journal
Dec 01, 2000
URL:http://www.ddj.com/architect/184404351

Automatic Detection of Potential Deadlock

Work performed while at @TheMoment Inc. John Mount can be contacted at jmount@mzlabs.com.

Real-time e-commerce applications that process thousands of user requests every second are challenging projects to develop and maintain. The challenges include compressed implementation/release cycles, remote applications, distributed applications, and interaction between third-party software components.

For example, the Dynamic Trading Suite from @TheMoment (the company I work for) is a web-based live business exchange allowing users to place offers to buy and sell, view summary data, and watch real-time trading animations from @TheMoment's central exchange. The system supports a throughput and immediacy not achievable in traditional database-backed web sites. In developing the system, our primary challenge was to develop a reliable high-performance system. In this article, I'll examine a fundamental challenge -- developing a multithreaded solution that is both correct and delivers high performance. We ensure the correctness of our system by ensuring that "Mutexs" (Mutual Exclusion locks) guard data accessed by multiple threads (to ensure that no two transactions ever try to overwrite the same piece of data at the same time). One path to high performance is to use "fine-grain" locking. Fine-grain locking is a strategy where a large number of locks each guard a small amount of data, in the hope that threads rarely collide when attempting to acquire a lock. However, using locks itself potentially exposes a dangerous situation called "deadlock."

Deadlock occurs when a number of consumers (typically threads) access a set of resources (Mutexs or locks) in an inconsistent pattern. The large numbers of locks used in a fine-grain locking solution produce a large number of lock-to-lock interactions, any of which could conceal a deadlock. My strategy is to analyze the lock-to-lock interactions so that the developer can remove any bad interactions. A simple example of deadlock can be given with two threads named "*Thread1*" and "*Thread2*" and two locks named "*LockA*" and "*LockB*," respectively. Each thread has an intended sequence of operations. *Thread1* intends to acquire *LockA* and then acquire *LockB*. *Thread2* intends to acquire *LockB* and then *LockA*; see Figure 1. Deadlock occurs when, due to scheduling circumstances, both threads make it half way into their respective plans. In this situation, *Thread1* has acquired *LockA* and not yet tried to acquire *LockB*, while *Thread2* has acquired *LockB* and not yet tried to acquire *LockA*. At this point, the system is deadlocked. Neither thread can progress any further until the other thread releases the lock it holds, but neither thread is prepared to release its lock. Without intervention, the system is frozen forever. There are systems that automatically resolve deadlocks at run time by aborting one thread or the other. However, such run-time intervention is not always practical or desirable. I advocate avoiding deadlock by design. For instance, you could remove the potential deadlock from this example by requiring that *Thread2* acquire *LockA* first so that *Thread2* now has an access pattern consistent with *Thread1*'s access pattern.

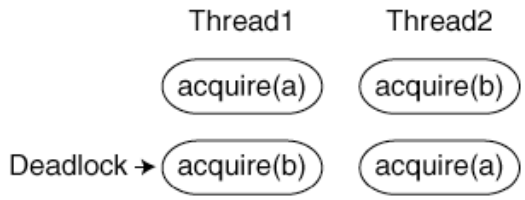

Figure 1: Each thread has an intended sequence of operations.

A concrete deadlock following the mentioned example could arise in many places in an online trading system. An example is if *Thread1* requests a list of all of a user's outstanding offers to sell in the system at the same time that *Thread2* tries to look up the owner of a given offer to sell that happens to belong to this same user. This locking strategy is misdesigned because *Thread1*'s transaction holds a lock on the user's biography structure and is attempting to acquire a lock for one of the offers for sale at the same time that *Thread2*'s transaction holds the offer for sale lock and is attempting to acquire the same user biography lock. These interactions are hard to anticipate, as different programmers may implement each of the queries. It is inconvenient to preassign an ordering to all the locks in the system as users are continually entering and leaving the system, and many small lockable objects are continually being created and destroyed. Finally, the very nature of deadlock makes it hard to detect during quality-assurance runs -- it depends on the precise timing of simultaneous events. Obtaining anything near complete coverage of all simultaneous run-time situations is impossible.

Imagine you are responsible for a large multithreaded system that is known to be largely correct -- no memory errors, no errors in lock use, and doesn't seem to deadlock on debugging data. In this situation, what guarantee do you have that a potentially embarrassing "rare" deadlock is not lurking in the system? In this case, rare would be defined as "doesn't show up in the debugging workload" but "does appear intermittently in the released application." Deadlocks depend on the timing of multiple interfering events, so this situation is not uncommon. What you need to know is that it would take more than just a change in lock timing and event ordering for users to expose a deadlock not found in debugging.

The solution I settled on is a run-time (as opposed to a compile-time or static) lock analysis. While there are a number of excellent commercial tools (notably Sun's lockLint and the KL Group's Jprobe Threadalizer), I opted for a roll-your-own solution that could be tightly integrated into our system and our remote logging facility. The run-time analyzer analyzes all transactions and determines if there is any possible ordering of the locks, such that all transactions never pick up a lock out of order. If such an ordering exists, then no rerun of the application that differs only in lock timing or event ordering can deadlock. In this case, we say the observed access pattern is "potential deadlock free." When a potential deadlock is detected we can report to the user (using code not given here) the locks involved and where they were acquired (using information captured from C++'s preprocessor), yielding a fast debug cycle.

With the aforementioned use in mind, I present my system for automatically analyzing (at run time) the lock activity that detects potential deadlock or certifies the observed run as potential deadlock free. Again, potential deadlock free is a powerful guarantee (much better than not observing any deadlocks during debugging) but is a technical term. An application that has been certified potential deadlock free means that no situation that could lead to deadlock under any variation of lock scheduling/timing was observed during testing runs. The application could, of course, still have a lurking deadlock hidden on a code path not observed during debugging/certification; see

Figure 2. Combining this run-time technique with traditional code-coverage techniques decreases the odds of having such a lurking path. This analysis scheme is very powerful -- a successful certification of even a single-thread version of an application can certify that a multithreaded version could not deadlock on the same set of transactions. In fact, checking if an application is potential deadlock free is checking if the application obeys an ordered or hierarchical lock discipline. While there are correct deadlock free programs that do not use such a discipline, hierarchical locking is a good design practice and is common in actual use. In ordered or hierarchical locking, an order is assigned to all the locks in the system, and any thread that acquires multiple locks must acquire them in the hierarchical order. As all threads are obeying the same discipline, they cannot deadlock. An obvious obstacle to automatically checking if a system is obeying such a discipline is the requirement of explicitly specifying the lock hierarchy to the checking software. My system doesn't require any such specification.

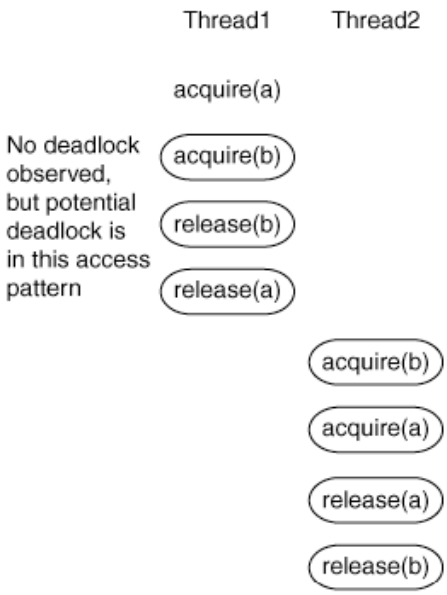

Figure 2: An application that has a potential deadlock.

I include here my C++ implementation of the basic analyzer using the Linux version of Posix threads (though I regularly use this system using Windows and Microsoft Visual C++). In this example, I have set up a lock class called *Synchronizer* (see Listing One). Source code is modified so all locking goes through this locking class (instead of using native methods). Such source-code modification is not strictly necessary; in fact, John Robins's *Debugging Applications* (Microsoft Press, 2000) has excellent advice on instrumenting Windows programs at run time. However, it is still good practice to route all of your lock access through a class like *Synchronizer* (this helps with porting and debugging). Listing One defines the *Synchronizer* class and the *instrumentedSynchronizer* class (derived from *Synchronizer*), which adds code to log the lock activity. This is also a good place to add any other debugging instrumentation. Things to check for include deleting locked locks, threads terminating while owning locks, and threads attempting to acquire nonrecursive locks multiple times. This example implementation does have run-time consequences. There is the overhead of lock tracking and the fact that we are using a single lock to access the logging data structures. The single lock serializes all of the lock's access (which can affect application timing). The serialization can be avoided by using per-thread logging and merging the logs offline.

The *instrumentedSynchronizer* implementation in Listing Two demonstrates what must be tracked when looking

for potential deadlock. For each lock, we create a record called a "node." Every time a thread tries to acquire a lock not currently owned by the thread, we draw an arrow from each and every lock node already owned by the thread to the lock node that the thread is trying to acquire. The tracking procedure is implemented entirely by adding nodes and edges to a single graph (see Figure 3). I do not count a thread attempting to acquire a lock it already owns as an actual lock acquisition because the thread can't block another thread during such a call and no other thread can detect such an event has occurred. A thread can block an attempt to acquire a lock it already owns (for example Posix "fast" style locks), but if this is the case then such an attempt is really a lock error (and can be detected without complicated analysis). On the other hand, Windows "critical sections" allow multiple acquisitions of a given lock. The graph can be inspected at any time to see if a potential deadlock has arisen. However, because the log is cumulative, I can make do with a single analysis at the end of the run (this is especially useful when verifying systems thought to be correct, as it cuts down on analysis overhead). Potential deadlock tracking differs from traditional deadlock tracking in that no modification is made to the directed lock graph when a lock is released.

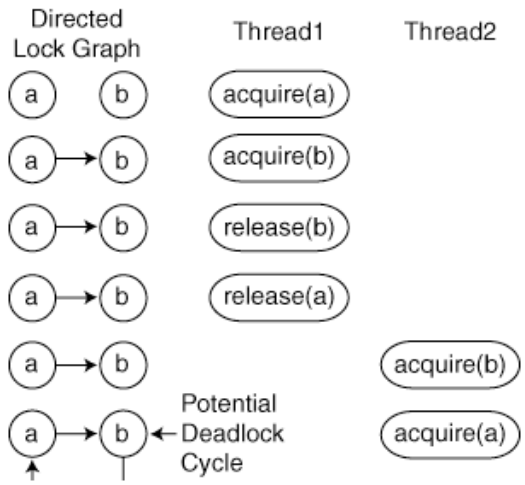

Figure 3: The tracking procedure is implemented entirely by adding nodes and edges to a single graph.

When a lock is deleted we are tempted to remove it from the log (remember that the system under analysis might create and destroy objects in addition to locking them). This would be a weaker analysis as the results would depend on the timing of an object's creation and destruction. Instead, as in the example implementation given here, we can leave the destroyed lock's node in our graph or we can remove the node using a "transitive closure" operation. To remove a node we no longer need without losing its information we must first add arrows to our graph. The rule is: For each and every node pointing to the node we want to delete, we direct an arrow to each and every node pointed to by the node we want to delete (see Figure 4). Such a replacement should only be made if space is at a premium. There are some potential problems -- annotations from the missing node may be lost, the data structure size may actually grow, and some nodes may now point to themselves (indicating they were in a potential deadlock cycle with nodes that have since been deleted).

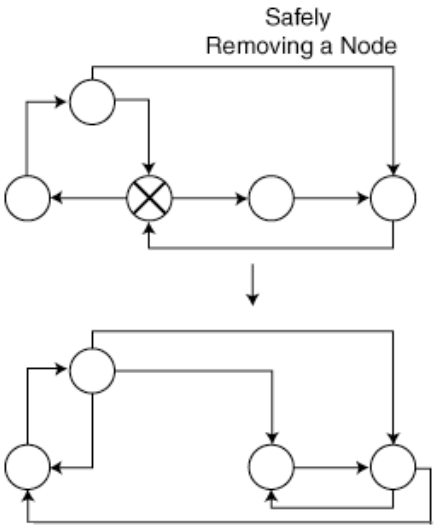

Figure 4: Directing arrows to each and every node pointed to by the node we want to delete.

The *LockMonitor* class in Listing Two efficiently maintains the directed lock graph. For brevity and safety, I use templates and the C++ Standard Template Library (see *Generic Programming and the STL*, by Matthew H. Austern, Addison-Wesley, 1998). The ease of maintenance far outweighs the speed and space that would be saved by a more direct pointer implementation. The first class *oneSidedMap* (available electronically; see "Resource Center," page 5) supports a map from items of type "*a*" to sets of items of type "*b*." An obvious operation that this class does not support is the automatic removal of a *b*-item from all sets. The class *incidenceMap* implements efficient removal by using two *oneSidedMap*s -- one that maps *a* to *b* and one that maps *b* to *a*. Each of these two maps contains exactly the information needed to support an efficient delete on the other map. I use *incidenceMap* to track which locks are owned by each thread. Because each lock can, at most, be owned by one thread, we see that the *incidenceMap* class is slightly more powerful than what we strictly need. This small negative is outweighed by the large positive that we can reuse the *incidenceMap* class to quickly (and efficiently) implement the directed lock graph. The class *directedGraph* implements a directed graph structure that provides arrow insertion and node deletion (the two operations we need). The representation is that each node is split into a left and a right. The left portion of a node is a list of nodes that the node points to and the right portion of a node is a list of nodes pointing to the node.

The *directedGraph* class implements the cycle detection needed to check if we have a potential deadlock. The essential observation (Donald Knuth traces this back to E. Szpilrajn in 1930) is that we have a potential deadlock (that is, locks that cannot be ordered) if and only if there is a cycle in the directed lock graph. In a similar vein, Donald Knuth demonstrates (*The Art of Computer Programming*, Vol. 2, by Donald E. Knuth, Addison-Wesley, 1973) a definitive cycle-detection algorithm. It is based on the observation that an acyclic graph (that is, a graph without a directed cycle) always has a node with no outgoing arrows (similarly, it must have a node with no incoming arrows). The cycle-checking algorithm is then a repetition of the following two-step operation. First, if all nodes have outgoing arrows we know the graph has a cycle and we are done. Second, if there are nodes without outgoing arrows we remove all such nodes (and all arrows pointing to them). The operations are repeated until we decide the graph has a cycle or we have no nodes left -- which means the original graph was acyclic (see Figure 5). This is very efficient: The number of set accesses is proportional to the number of nodes and arrows in the directed lock graph (which is typically much smaller than the number of nodes squared). The final graph, if

nonempty, definitely contains at least one cycle. However, the graph itself may still contain extra irrelevant edges and nodes (one example graph that contains multiple cycles, but cannot be further reduced by the mentioned methods, is two disjoint cycles connected by a single edge). Fortunately, in the final graph it is easy to find a cycle: Just follow outgoing edges until we see a node-label repeat. However, in my actual industrial implementation I use a breadth-first search to find the shortest cycle in the reduced graph (see *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, MIT Press, 1990). I have also found it helpful to decorate the cycle data with the line of source code at which each lock dependency was observed (this is easily captured by defining an *acquire()* macro that incorporates the C++ preprocessor's __FILE__ and __LINE__ directives).

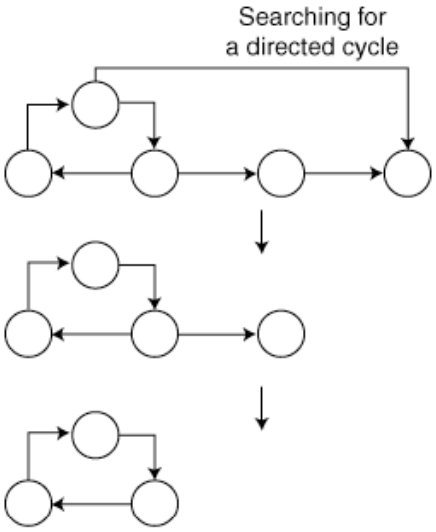

Figure 5: Operations are repeated until either we decide the graph has a cycle or we have no nodes left.

This technique is useful and can be extended in many ways. One extension I mentioned is the optional debugging of lock logic (detecting locks that are deleted while locked, threads that terminate owning locks). Another extension is to generate reports of lock utilization. Furthermore, the algorithm I used to look for potential deadlock is called "topological sort" and can be used to generate a report of allowable lock orders. Such a report could then be incorporated into design and specification documents so that the lock hierarchy discovered by the software can become part of the design plan and documentation. The system has proven efficient enough in practice to analyze a reasonably large transaction system through tens of thousands of transactions.

Listing Three is a simple use of the system. If compiled and run with no arguments, it runs and analyzes a safe example. If run with an argument, it runs an unsafe example (a lock pattern that could deadlock if the length of the *sleep()* intervals were changed). The output of these runs is given in Example 1 and Example 2, respectively.

```
R1 Thread 1026 try a->acquire();
R1 Thread 1026 try b->acquire();
R2 Thread 2051 try b->acquire();
```

```
R3 Thread 3076 try a->acquire();
R1 Thread 1026 try a->release();
R1 Thread 1026 try b->release();
R2 Thread 2051 try c->acquire();
R2 Thread 2051 try c->release();
R2 Thread 2051 try b->release();
R3 Thread 3076 try c->acquire();
R3 Thread 3076 try c->release();
R3 Thread 3076 try a->release();
do not see potential deadlock
```

Example 1: Output without potential deadlock.

```
R1 Thread 1026 try a->acquire();
R1 Thread 1026 try b->acquire();
R2 Thread 2051 try b->acquire();
R1 Thread 1026 try a->release();
R1 Thread 1026 try b->release();
R2 Thread 2051 try c->acquire();
R2 Thread 2051 try c->release();
R2 Thread 2051 try b->release();
R4 Thread 3076 try c->acquire();
R4 Thread 3076 try a->acquire();
R4 Thread 3076 try c->release();
R4 Thread 3076 try a->release();
see potential deadlock
```

Example 2: Output with potential deadlock.

The methods shown here are not the last word. Much more can be done in tracking and reporting. Sun's lockLint, for instance, can identify variables that are accessed from multiple threads without any unique lock being held. The KL Group's Jprobe Threadalizer has "lock covers" that allow it to recognize nonhierarchical (but correct) lock acquisition sequences. One such situation is when a thread acquires (in order) *LockA*, *LockB*, and *LockC* without ever releasing LockA until it is done acquiring locks. Another thread, at another time, could acquire (in order) *LockA*, *LockC*, and *LockB* without ever releasing *LockA* until it is done acquiring locks. These two threads look like a potential deadlock (they acquire *LockB* and *LockC* in different orders), but can never deadlock against each other for the simple reason that their use of *LockA* prevents them from ever running at the same time. This does not contradict the potential deadlock indication, as potential deadlock is really a statement that, if both events ever happened at the same time, they could deadlock. Such sophisticated reporting requires more detailed data structures (remembering which locks have been held continuously) and also requires more complicated search/reporting strategies, because the underlying problem of finding cycles with fairly arbitrary edge-label requirements is NP hard. However, the actual configurations arising from real programs are probably fairly amenable to a good breadth-first search algorithm.

Conclusion

I have found that the potential deadlock analyzer has had a positive impact on @TheMoment's software-development cycle. The majority of the access patterns that it red-flagged were indeed questionable lock-access patterns. False alarms, seemingly inconsistent access patterns that could never happen at the same time (because both are guarded by a common lock or due to limited visibility of locks), did occur. However, they were in the

minority and easily coded around. The lesson learned was that most deadlocks involved code from two programmers. A typical case was when code written by programmer A called code written by programmer B, which then called back some programmer A code. The error was often that programmer A was holding a lock that they never expected any other programmer to know about. Often this lock caused a deadlock during the callback. Thus, a lock defends a thread against other threads, but cannot defend a thread against itself.

DDJ

Listing One

```
//Synchronizer.h

#ifndef Synchronizer_h_included
#define Synchronizer_h_included

typedef pthread_t threadId;
typedef int synchronizerId;

class Synchronizer {
public:
  Synchronizer();
  virtual ~Synchronizer();
  virtual void acquire();
  virtual void release();
private:
  pthread_mutex_t M;
  // deliberately not implemented
  Synchronizer(const Synchronizer &);
  Synchronizer &operator=(const Synchronizer &);
};
class instrumentedSynchronizer : public Synchronizer {
public:
  instrumentedSynchronizer();
  ~instrumentedSynchronizer();
  void acquire();
  void release();
private:
  synchronizerId myid;
  // deliberately not implemented
  instrumentedSynchronizer(const instrumentedSynchronizer &);
  instrumentedSynchronizer &operator=(const instrumentedSynchronizer &);
};
extern threadId CurThreadId();
extern bool havePotentialDeadlock();
#endif
```

Back to Article

Listing Two

```
//Synchronizer.cpp

#include <map>
```

```
#include <set>
#include <vector>
using std::map;
using std::set;
using std::vector;

#include <iostream.h>
#include <pthread.h>
#include <sys/types.h>

#include "incidenceMap.h"
#include "Synchronizer.h"

Synchronizer::Synchronizer() {
  pthread_mutex_init(&M,NULL);
}
Synchronizer::~Synchronizer() {
  pthread_mutex_destroy(&M);
}
void Synchronizer::acquire() {
  pthread_mutex_lock(&M);
}
void Synchronizer::release() {
  pthread_mutex_unlock(&M);
}
threadId CurThreadId() {
  pthread_self();
}
class LockMonitor : private Synchronizer {
public:
  LockMonitor() {
    nextId = 1;
  };
  bool isAcyclic() {
    acquire();
    bool rv = priorSyncMap.isAcyclic();
    release();
    return rv;
  };
  int assignId() {
    acquire();
    int rv = nextId++;
    release();
    return rv;
  };
  void observeAcquireAttempt(synchronizerId sid, threadId t) {
    acquire();
    if(!syncOwnedBy.isPair(sid,t)) {
      vector<synchronizerId> *u = syncOwnedBy.left(t);
      if(u!=NULL) {
        for(unsigned int i = 0;i<u->size();++i) {
          priorSyncMap.insertPair(sid,(*u)[i]);
        }
        delete u;
      }
      syncOwnedBy.insertPair(sid,t);
    }
    release();
  };
  void observeRelease(synchronizerId sid, threadId t) {
    acquire();
    syncOwnedBy.erasePair(sid,t);
    release();
```

```
  };
private:
  directedGraph<synchronizerId> priorSyncMap;
  synchronizerId nextId;
  incidenceMap<synchronizerId,threadId> syncOwnedBy;
  // deliberately not implemented
  LockMonitor(const LockMonitor &);
  const LockMonitor &operator=(const LockMonitor &);
};
// Singleton pattern (guarantees initialization order). Assumes this is called
// before we have gone multithreaded, so there is no race condition on new-
// command. Deliberately leak monitor on heap, so it outlasts non-heap objects
static LockMonitor *GlobalLockMonitor() {
  static LockMonitor *GLM = NULL;
  if(GLM==NULL) {
    GLM = new LockMonitor;
  }
  return GLM;
}
bool havePotentialDeadlock() {
  return !GlobalLockMonitor()->isAcyclic();
}
instrumentedSynchronizer::instrumentedSynchronizer() {
  myid = GlobalLockMonitor()->assignId();
}
instrumentedSynchronizer::~instrumentedSynchronizer() {
}
void instrumentedSynchronizer::acquire() {
  threadId t = CurThreadId();
  GlobalLockMonitor()->observeAcquireAttempt(myid,t);
  Synchronizer::acquire();
}
void instrumentedSynchronizer::release() {
  threadId t = CurThreadId();
  GlobalLockMonitor()->observeRelease(myid,t);
  Synchronizer::release();
}
```

Back to Article

Listing Three

```
//Main.cpp

#include <iostream.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include "Synchronizer.h"

static void dumpstat() {
  if(havePotentialDeadlock()) {
    cout << "see potential deadlock\n";
  } else {
    cout << "do not see potential deadlock\n";
  }
}
// These are created before main is entered, guaranteeing that
// the global monitor is built before we create new threads.
static instrumentedSynchronizer a,b,c;
static void *R1(void *) {
  cout << "R1 Thread " << CurThreadId() << " try a.acquire();\n";
```

```
  a.acquire();
  cout << "R1 Thread " << CurThreadId() << " try b.acquire();\n";
  b.acquire();
  sleep(2);
  cout << "R1 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  cout << "R1 Thread " << CurThreadId() << " try b.release();\n";
  b.release();
  return NULL;
}
static void *R2(void *) {
  sleep(1);
  cout << "R2 Thread " << CurThreadId() << " try b.acquire();\n";
  b.acquire();
  cout << "R2 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R2 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R2 Thread " << CurThreadId() << " try b.release();\n";
  b.release();
  return NULL;
}
static void *R3(void *) {
  sleep(1);
  cout << "R3 Thread " << CurThreadId() << " try a.acquire();\n";
  a.acquire();
  cout << "R3 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R3 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R3 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  return NULL;
}
static void *R4(void *) {
  sleep(3);
  cout << "R4 Thread " << CurThreadId() << " try c.acquire();\n";
  c.acquire();
  cout << "R4 Thread " << CurThreadId() << " try a.acquire();\n";
  a.acquire();
  cout << "R4 Thread " << CurThreadId() << " try c.release();\n";
  c.release();
  cout << "R4 Thread " << CurThreadId() << " try a.release();\n";
  a.release();
  return NULL;
}
int main(int argc, char *argv[]) {
  bool fail = (argc>1);
  pthread_t T1,T2,TX;
  pthread_create(&T1,NULL,R1,NULL);
  pthread_create(&T2,NULL,R2,NULL);
  if(!fail) {
    pthread_create(&TX,NULL,R3,NULL);
  } else {
    pthread_create(&TX,NULL,R4,NULL);
  }
  sleep(5);
  dumpstat();
  return 0;
}
```