Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables

Rahul Agarwal
Joint work with Prof. Scott D. Stoller
Deadlock

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose.

A common kind of concurrency error is a deadlock.

Informally, a deadlock occurs when some threads are permanently blocked.
Synchronization Mechanisms

- **Locks** (block structured and non block structured)
  - Operations: acquire(), release()

- **Semaphores**
  - Operations: up(), down()

- **Condition variables**
  - A lock is associated with each condition variable.
  - Operations: wait(), notify(), notifyAll()

- **Thread synchronization**
  - Operations: start(), join()

Available in Java 5 concurrency library and POSIX pthread library for C.
Blocking Event

An event is one step in the execution of a program.
- Consider synchronization events and access to shared variables.

A trace is a sequence of events in an execution.

An event $e$ in a trace $tr$ is called a blocking event if one of the following holds:
- $e$ is an acquire of a lock $l$ by thread $t$, and $l$ is currently held by another thread.
- $e$ is a wait on a condition variable.
- $e$ is down on a semaphore whose value is 0.
Deadlock

A trace $tr$ deadlocks if a set $T$ of threads in $tr$ exists, such that the last event for each thread in $T$ is a blocking event, and all threads in $tr$ not in $T$ have terminated.
Feasible Permutation of a Trace

A feasible permutation of a trace $t$ is a permutation of $t$ that preserves the original order of events from each thread and is consistent with constraints imposed by synchronization events.

- Constraint imposed by locks is that no lock is held by multiple threads at same time.
- Constraints due to other synchronization mechanisms are expressed as happens-before orderings.
Happens-before

Happens-before is a partial order on the events in an execution.

If event $e_1$ happens before event $e_2$, then $e_1$ must precede $e_2$ in all feasible permutations of the trace.
Happens-before Ordering Due to Thread Synchronization

When a thread \( t_1 \) calls \( t_2 . \text{start}() \) to start another thread \( t_2 \), then the thread start event in \( t_1 \) happens before the first event of \( t_2 \).

When thread \( t_1 \) calls \( t_2 . \text{join}() \) to wait for another thread \( t_2 \) to terminate, last event of \( t_2 \) happens before join event in \( t_1 \).
Potential for Deadlock

A trace has potential for deadlock if some feasible permutation of the trace deadlocks.

This definition considers all synchronization mechanisms together.
Potential for Deadlock Due to Locks

- A trace has potential for deadlock due to locks if some feasible permutation of trace restricted to operations on locks and operations on threads deadlocks.

- A simpler condition below can be checked efficiently.
  - Ignores gate locks and thread operations.

- A program has potential for deadlock due to locks ignoring gate locks (PDL-IGL) if there exist distinct threads $t_0, ..., t_{m-1}$ and locks $l_0, ..., l_{m-1}$ in a given trace s.t. for all $i = 0, ..., m-1$, $t_i$ holds lock $l_i$ while acquiring lock $l_{i+1 \mod m}$.
Lock Trees

- Root labeled by name of the thread.
- One child for each lock acquired by thread t.
- Each of those nodes is labeled with name $l$ of one of those locks and has child labeled with a lock $l'$ iff t acquired $l'$ while holding $l$. Note: Height at most 2.
Lock Trees

Root labeled by name of the thread.
One child for each lock acquired by thread t.
Each of those nodes is labeled with name \( l \) of one of those locks and has child labeled with a lock \( l' \) iff \( t \) acquired \( l' \) while holding \( l \). Note: Height at most 2.
Lock Trees

- Root labeled by name of the thread.
- One child for each lock acquired by thread $t$.
- Each of those nodes is labeled with name $l$ of one of those locks and has child labeled with a lock $l'$ iff $t$ acquired $l'$ while holding $l$. Note: Height at most 2.
Lock Trees

- Root labeled by name of the thread.
- One child for each lock acquired by thread t.
- Each of those nodes is labeled with name $l$ of one of those locks and has child labeled with a lock $l'$ iff $t$ acquired $l'$ while holding $l$. Note: Height at most 2.
Lock Trees

- Root labeled by name of the thread.
- One child for each lock acquired by thread t.
- Each of those nodes is labeled with name $l$ of one of those locks and has child labeled with a lock $l'$ iff t acquired $l'$ while holding $l$. Note: Height at most 2.

```
 acquire(L1)
 acquire(L2)
 acquire(L3)
 release(L2)
```
Lock Trees

- Root labeled by name of the thread.
- One child for each lock acquired by thread \( t \).
- Each of those nodes is labeled with name \( l \) of one of those locks and has child labeled with a lock \( l' \) iff \( t \) acquired \( l' \) while holding \( l \). Note: Height at most 2.

```
Thread 1
   /|
  / |\
L1 L2
   |  |
  |  |
L2 L3
   |  |
  |  |
L2 L3
```

acquire(L1)
acquire(L2)
acquire(L3)
release(L2)
release(L3)
release(L1)
Detecting Potential for Deadlock Due to Locks Ignoring Gate Locks (PDL-IGL)

Construct a run-time lock graph which contains

- **tree edges**: the directed (from parent to child) edges in each of the run-time lock trees, and

- **inter edges**: bidirectional edges between nodes that are labeled with the same lock and that are in different run-time lock trees.

PDL-IGL holds iff the graph contains a *valid cycle*.

Valid cycle is a cycle that

- does not contain consecutive inter edges, and

- nodes from each thread appear as at most one consecutive subsequence in the cycle.

Related Work: GoodLock Algorithm [Havelund, 2000]
Example

acquire(l1);
acquire(l2);
release(l1);
release(l2);
acquire(l3);
acquire(l4);
release(l4);
release(l3);
acquire(l2);
acquire(l3);
release(l3);
release(l2);
acquire(l4);
acquire(l1);
release(l4);
release(l1);

no valid cycles: no deadlocks
Example

Valid cycle is a cycle that

- does not contain consecutive inter edges, and
- nodes from each thread appear as at most one consecutive subsequence in the cycle.

no valid cycles: no deadlocks
Example

Valid cycle is a cycle that

- does not contain consecutive inter edges, and
- nodes from each thread appear as at most one consecutive subsequence in the cycle.

no valid cycles: no deadlocks
Example

acquire(l1);
acquire(l2);
release(l1);
release(l2);

acquire(l2);
acquire(l3);
release(l3);
release(l2);

acquire(l3);
acquire(l1);
release(l3);
release(l1);

Warning: Potential Deadlock
Example

Valid cycle: Potential Deadlock

acquire(l1);
acquire(l2);
release(l1);
release(l2);

acquire(l1);
acquire(l2);
release(l1);
release(l2);

acquire(l2);
acquire(l3);
release(l3);
release(l2);

acquire(l3);
acquire(l1);
release(l3);
release(l1);
Detecting Valid Cycles

- Use a modified depth-first search algorithm.
- Presented optimizations to the algorithm [Agarwal+, 2005].
- Better complexity than checking for all feasible permutations of the trace.
Detecting Potential for Deadlock Due to Locks

- A cycle in the lock graph may be protected by a *gate lock*, a common lock acquired by at least two threads involved in the cycle.
- For every valid cycle generated, we check whether there is a gate lock preventing the deadlock.
- Details in the paper.
Potential for Deadlock Due to Semaphores

An execution trace has potential for deadlock due to semaphores if some feasible permutation of trace restricted to operations on semaphores and operations on threads deadlocks.
Dual Nature of Semaphores

Semaphores can be used to provide **mutual exclusion** or condition synchronization.

We **classify** a semaphore `sem` as used for mutual exclusion if $(\exists \text{ thread } t. \ t:\text{sem}.\text{down}() \ t:\text{sem}.\text{up}()^* (\exists \text{ thread } t. \ t:\text{sem}.\text{down}()))$?

Semaphores used for **mutual exclusion** are **analyzed exactly like locks** with `down` treated as **acquire** and `up` treated as **release**.
Happens-before Ordering Due to Semaphores

Semaphores not used for mutual exclusion are analyzed based on their happens-before-before ordering.

An up event $e_u$ that unblocks a thread blocked on a down event $e_d$ happens before $\text{succ}(e_d)$ where $\text{succ}(e)$ is the event immediately following $e$ on the same thread.

![Diagram of up and down events](image-url)
Cigarette Smokers Problem

Initially tob=0, pap=0, match=0, order = 1

**Smoker 1:** while(1) {tob.down(); pap.down(); order.up();}

**Smoker 2:** while(1) {pap.down(); match.down(); order.up();}

**Smoker 3:** while(1) {match.down(); tob.down(); order.up();}

**Agent:** while(1) {order.down();

up on one of tob, paper, or match at random;

up on one of three at random but not one above;}

July, 2006

Rahul Agarwal, Stony Brook University 28
Deadlock Free Trace

**Smoker1**
- t.down
- p.down
- o.up

**Smoker2**
- p.down
- m.down
- o.up

**Agent**
- o.down
- t.up
- p.up
- o.down
- p.up
- m.up
Happens-before Ordering for the Trace

Smoker1: t.down p.down o.up

Smoker2: p.down m.down o.up

Agent: o.down t.up p.up o.down p.up m.up
Feasible Permutation that Deadlocks

The original trace has potential for deadlock.
Lost Notifies

A notify is lost if it occurs before the thread it should wake actually calls wait.

As a result, the notify has no effect, and when that thread does call wait, it may wait forever.
Potential for Lost Notify

A n execution trace has potential for lost notify if it contains a notify or notifyall event \( e \) such that there is a feasible permutation of \( tr \) in which \( e \) wakes up fewer threads than it does in \( tr \).

This is possible when the wait event of one of the threads woken in \( tr \) is not constrained to happen before \( e \).
Example

class EventHandler extends .... {
    public void handleEvent(Event e) {
        switch(e.type) {
            case update: data.update();
                sync(computeThread){computeThread.notify();}
            break;
        }
    }
}

class ComputeThread extends Thread {
    public void run () {
        while (true) {
            sync(this) {this.wait(); compute();} }
        }
}
Happens-before Ordering Due to Condition Variables

For each notify or notifyAll event $e_n$ and each wait event $e_w$ that is notified by $e_n$, $e_n$ happens-before $\text{succ}(e_w)$, where $\text{succ}(e)$ is the event immediately after $e$ on the same thread.

Since, the same lock $l$ must be held when $e_w$ and $e_n$ occur and only one thread can hold a given lock at a time, this is equivalent to saying that the release of $l$ after $e_n$ happens-before $\text{succ}(e_w)$.

\begin{itemize}
  \item acq(a) rel(a)
  \item notify
  \item wait
  \item acq(a) rel(a)
\end{itemize}
Trace without Lost Notify

EventHandler

ComputeThread

acq notify rel

acq wait rel acq
Happens-before Ordering for the Trace
Feasible Permutation with Lost Notify

EventHandler: acq notify rel

ComputeThread: acq wait rel acq
Happens-before Ordering Due to Condition Variables

Orderings due to control dependencies on access to shared variables which is useful for condition sync.

A read event $e_r$ on some shared variable that occurs in boolean condition of if or while statement happens after the previous write event $e_w$ ($e_w$ happens before $e_r$) and happens-before the next write event to that variable.
public synchronized int get() {
    while (available == false) {
        wait();
    }
    available = false;
    notifyAll();
    return contents;
}

public synchronized void put(int value) {
    while (available == true) {
        wait();
    }
    contents = value;
    available = true;
    notifyAll();
}
Trace for Producer Consumer

Producer

Consumer
Happens-Before Ordering for the Trace

Producer

Consumer

acq  rd(av)  wr(av)  rel

acq  rd(av)  wait  rel  acq

notify
Happens-before Ordering for the Trace

Producer

Consumer

No lost notifies!!
Detection of Potential Deadlocks Due to Locks, Condition Variables, and Semaphores

- Classify semaphore as used for mutual exclusion or used as condition variable.
- All ordering constraints so far are imposed.
- Check if a feasible permutation of the trace deadlocks.
Example

public sync doWait(B b) { /*this: Class A object.*/
    compute();
    sync(b) {b.wait();}
}

public doNotify(B b) {
    sem.down();
    sync(b) {b.notify();}
}

public sync doCompute() { /*this: Class A object*/
    compute();
    sem.up();
}
Deadlock Free Trace

doWait

\[ \text{acq(a) acq(b) rel(b)} \]

\[ \text{wait} \]

\[ \text{acq(b)} \]

doNotify

\[ \text{sem.down acq(b) notify} \]

\[ \text{rel(b)} \]

doCompute

\[ \text{acq(a) sem.up rel(a)} \]
Partial Order for the Trace

doWait

acq(a) acq(b) rel(b)

wait

acq(b)

doNotify

sem.down acq(b) notify

rel(b)

doCompute

acq(a) sem.up rel(a)
Feasible Permutation that Deadlocks

doWait

- acq(a)
- acq(b)
- rel(b)
- wait

doNotify

- sem.down

doCompute

- acq(a)
Feasible Permutation that has Lost Notifies

\begin{center}
\begin{tabular}{c}
\textbf{doWait} \\
\hline
acq(a) & acq(b) & rel(b) \\
\hline
\text{wait} \\
\textbf{doNotify} \\
\hline
\text{sem.down} & \text{notify} \\
\hline
acq(b) & rel(b) \\
\textbf{doCompute} \\
\hline
acq(a) & \text{sem.up} & rel(a)
\end{tabular}
\end{center}
Related Work

Run-time analysis
- GoodLock Algorithm [Havelund, 2000]
- MultiThread GoodLock Algorithm [Bensalem+, 2005, Agarwal+, 2005]
- ConTest [Edelstein+, 2003]
- JMPaX [Sen+, 2005]

Static Analysis
- SafeJava [Boyapati+, 2002]
- RacerX [Engler+, 2003]
- Williams et. al., 2005