See http://deadlockempire.github.io/#2-flags

Verification & Testing
Dynamic Algorithms for Concurrency Problems

Roderick Bloem

Sources:

- Savage, Burrows, Nelson, Sobalvarro, Anderson, Eraser: A Dynamic Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems 15, 1997
- Visser et al, Model Checking Programs, Model Checking Programs, Automated Software Engineering 10, 2003

Deadlocks & Race Conditions

Deadlocks show themselves when a program hangs

Race conditions cause unexpected results

- Hard to find because they often occur only with a specific scheduling.
- Often not found during testing but as low-frequency (high-impact) bugs at client site. Hard to reproduce.
- **Today**: Algorithms that find these problems without looking at all schedulings.

Dynamic Tools for Concurrency Problems

What we want:

- *better than testing*
- *works for any program we can run!*
- *We can sacrifice precision: unnecessary warnings, undiscovered bugs are OK*

Subject: **dynamic methods** to find concurrency errors – deadlocks and race conditions

Dynamic methods:

- Result depends on exact run (inputs and scheduling)
- Try to minimize dependence on scheduling

Locking Example

```
int available = 0;

thread 1:
public synchronized int get() {
  while (!available) {
    try { wait(); }
    catch (InterruptedException e) { }
  }
  available = false;
  notifyAll();
  return contents;  //still locked!
}

thread 2:
public synchronized void put(int value) {
  while (available) {
    try { wait(); }
    catch (InterruptedException e) { }
  }
  contents = value;
  available = true;
  notifyAll();
}
```

Explicit Locks

```
ReentrantLock l = new ReentrantLock();
l.lock();
…
l.unlock();
```

Note: synchronized locks are just locks on “this”

Deadlock

A deadlock is a circular wait

For locks, this is called *lock reversal*:

- Thread 1 holds lock A, waits for B
- Thread 2 holds lock B, waits for A

or with three threads:

- Thread 1 holds lock A, waits for B
- Thread 2 holds lock B, waits for C
- Thread 3 holds lock C, waits for A

Deadlock Example

```
ReentrantLock ALock =
    new ReentrantLock;
ReentrantLock BLock =
    new ReentrantLock;

class Alice{
  void hug(){
    ALock.lock();
      Block.lock();
        work…
      Block.unlock()
    ALock.unlock();
}}

class Bob{
  void hug(){
    BLock.lock();
      Alock.lock();
        work…
      Alock.unlock();
    BLock.unlock();
}}
```

Thread 1 calls Alice.hug()
Thread 1 calls ALock.lock()
[T1 holds AlLock]
Thread 2 calls Bob.hug
Thread 2 calls Block.lock();
[T1 holds AlLock, T2 holds BLock]
Thread 1 calls Block.lock()
[T1 holds ALock waits for BLock, T2 holds BLock]
Thread 2 calls Alock.lock()
[T1 holds ALock waits for BLock,
T2 holds BLock, waits for ALock]

(deadly embrace)

Gate Locks

A **gate lock** prevents a deadlock by protecting the areas with lock reversal

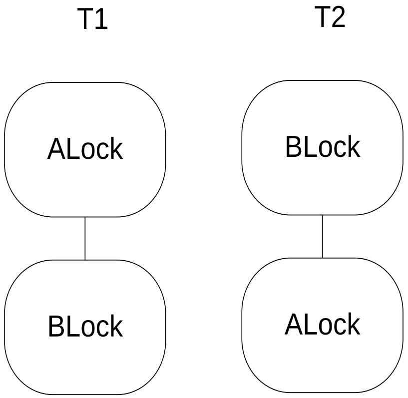
```
ReentrantLock gateLock;
class Alice{
  void hug(){
    gateLock.lock();
      ALock.lock();
        Block.lock();
        Block.unlock()
      ALock.unlock();
    gateLock.unlock();
}}

class Bob{
  void hug(){
    gateLock.lock();
      BLock.lock();
        Alock.lock();
        Alock.unlock()
      BLock.unlock();
    gateLock.unlock();
}}
```

Lock Tree Algorithm

Dynamic algorithm to find deadlocks

- **Lock reversal:** only for deadlocks with two threads
- **Dynamic:** may miss deadlocks (statements not executed at all or not in every possible order)
- **False warnings:** other mechanisms may prevent deadlock (e.g., shared variable)

In a tree, keep track the order in which locks are acquired and released; see if there are reversals



Lock Tree Algorithm

Build trees during runtime

- each tree has a current node
- If lock acquired create new child and move to it
- If node released, move up one level

After termination, analyze trees. Possible deadlock if

1. T1 contains a node Li with ancestor Lj
2. T2 tree contains a node Lj with ancestor Li
3. There is no gate lock: node Lk which is an ancestor of Li in T1 and Lj in T2

A gate lock is a lock that is

1. an ancestor of Li and Lj in T1 and
2. an ancestor of Li and Lj in T2

Limitations

- Works for deadlocks involving two threads only
- Works only for properly nested locks

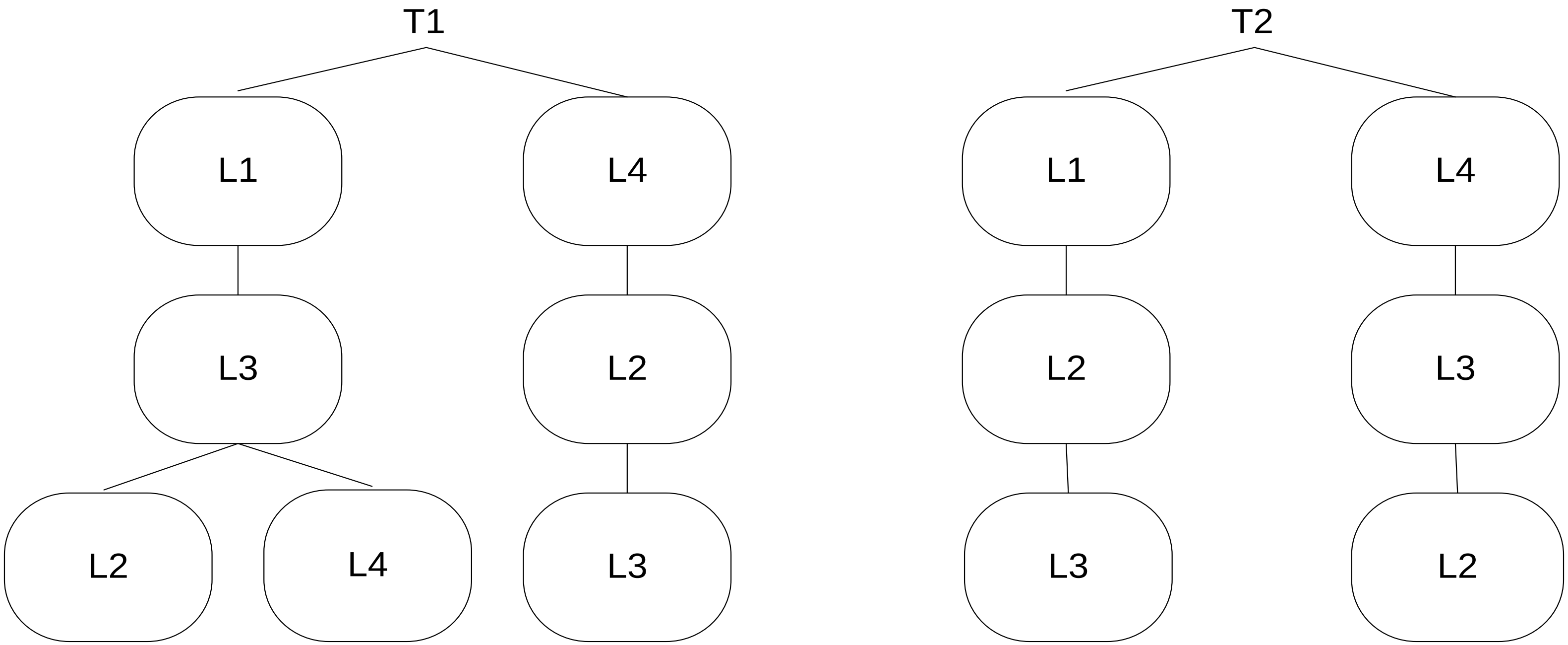
Lock Tree

```
Thread 1:

L1.lock();
  L3.lock();
    L2.lock();
    L2.unlock();
    L4.lock();
    L4.unlock();
  L3.unlock()
L1.unlock();
L4.lock();
  L2.lock();
    L3.lock();
    L3.unlock()
  L2.unlock();
L4.unlock();
```

```
Thread 2:

L1.lock();
  L2.lock();
    L3.lock();
    L3.unlock();
  L2.unlock
L1.unlock();
L4.lock();
  L3.lock();
    L2.lock();
    L2.unlock();
  L3.unluck();
L4.unlock();
```

Let’s draw lock tree by executing T1 first and then T2

Lock Tree



Where are the potential deadlocks?

Deadlocks

Potential deadlocks in the last example,

- L3L4 left versus L4L3 right is a problem
- L3L2 left versus L2L3 right is not: protected by L1
- L2L3 left versus L3L2 right is not: protected by L4

To get deadlock:

1. Execute T2, stop when L4 acquired,
2. Execute T1 until deadlock.

Note: executing T1 first then T2 will not give deadlock.

By executing one scheduling we found a problem in a different scheduling!

Limitations

1. Dependence on execution: If suspicious code is never executed, we do not find deadlock
2. Deadlocks do not have to be due to locks
3. Deadlocks can be prevented without using locks

(trick for 2,3: build your own lock.)

Limitations:
LockTree detects False Deadlock

```
class Lock{
Lock lock;

int a = 0; // the gate lock

class Alice{
  ReentrantLock ALock = …;
  void hug(){
    synchronize(lock){
      while(a==0) lock.wait();
    }
    ALock.lock();
      Block.lock();
      Block.unlock();
    ALock.unlock();
    a = 0;
    synchronize(lock){
      lock.notifyAll();
    }
  }
}
```

```
class Bob{
  ReentrantLock Block = …;
  void hug(){
    synchronize(lock){
      while(a==1) lock.wait();
    }
    Block.lock();
      Alock.lock();
      Alock.unlock();
    Block.unlock();
    a = 1;
    synchronize(lock){
      lock.notifyAll()
    }
  }
}
```

Limitations: An undetected Deadlock

```
class Lock{}
Lock lock;
int a = 0, b = 0;
```

```
class Alice{
  void hug(){
    synchronize(lock){
      while(a==0) lock.wait();
    }
    a = 0;
    b = 1;
    synchronize(lock){
      lock.notifyAll;
    }
  }
}
```

```
class Bob{
  void hug(){
    synchronize(lock){
      while(b==0) lock.wait();
    }
    b = 0;
    a = 1;
    synchronize(lock){
      lock.notifyAll();
    }
  }
}
```

Data Races

Data Race

A **data race** exists when:

1. Two threads access variable concurrently
2. At least one access is write
3. Nothing prevents simultaneous access

When data race occurs, result depends on the interleaving

Not *necessarily* bad

- Thermometer writes to int temp, GUI reads: no locks needed

But be careful:

- Writes to ints are atomic, so this works
- if temp is a long or a structure, you need locking

How do you usually prevent race conditions?

Eraser

- Check locking behavior
- For any shared data, is some lock always held on access?
- This condition is sufficient but not necessary for correctness
- Dynamic algorithm
  - Computes locks held during one run
  - May not find all problems
  - May warn when no problem exists
  - What it finds depends on the run!

Bank Account

(Grandma’s Disappearing Money)

```
class Acct{
  private long balance;
  private long acctNr;

  Acct(){
    acctNr = Acct.getNewNr();
    balance = 0;
  }

  long getAcctNr(){
    return acctNr;
  }
```

```
long getBalance(){
    return balance;
  }


void deposit(long amount){
    long current;

    current = balance;
    current += amount;
    balance = current;
  }
}
```

Data Race

```
void deposit(long amount){
  long current;

  current = this.balance;
  current += depositAmount;
  this.balance = current;
}
```

Initial balance is 0, deposit 100 twice. Final balance: 100 instead of 200.

Thread 1 (You):	**Thread 2 (Grandma):**
`account1.deposit(100)`	
	`account1.deposit(100)`
`current = balance; (0)`	
	`current = balance; (0)`
`current += amount; (100)`	
	`current += amount; (100)`
`balance = current; (100)`	
	`balance = current; (100)`

Where did Grandma’s money go??

- Same problem occurs if you use `balance +=amount`.

Eraser – Simple Version

At any point in time, a thread *t* holds a set of locks: *locks(t)*
Associate with each variable *v* a set of **lock candidates**, *C(v)*

```
For each variable v {
  C(v) = all_locks;
}

// called when thread t reads variable v
read(t,v){
  C(v) := C(v) ∩ locks(t);
  if C(v) = ∅ then issue warning;
}
// same for write(t,v)
```

Note: minimal dependence on order of scheduling!
Results only depends on execution paths taken (which may in turn depend on scheduler)

Example

Thread 1	Thread 2	locks(T1)	locks(T2)	C(v)
		∅	∅	{l1, l2}
`l1.lock();`		{l1}		
`v := 1;`				{l1}
`l1. unlock()`		∅		
	`l2.lock()`		{l2}	
	`v := v + 1;`			∅: warning!
	`l2.unlock()`		∅	

Bank Account, 2

```
class Acct{
  private long balance;
  private long acctNr;
  private ReentrantLock l;

  Acct(){
    acctNr = Acct.getNewNr();
    balance = 0;
    l = new Lock();
  }

  long getAcctNr(){
    return acctNr;
  }
```

```
  long getBalance(){
      long currentBalance;

      l.lock();
      currentBalance = balance;
      l.unlock();
      return currentBalace;
    }


  void deposit(long amount){
      long current;

      l.lock();
      current = balance;
      current += amount;
      balance = current;
      l.unlock();
    } }
```

Does this solve our problem?

Remaining Problems

Program is now correct but Eraser does not understand:

1. Initialization is not protected
    - But initialization is never simultaneous with anything else!
2. Account number not protected

Also, an efficiency problem:

- Two threads reading account data have to wait for each other.
    - We should exclude simultaneous read/writes, but simultaneous reads are OK.

We will solve problem 1 & 2 first

Initialization & Read-Shared

Virgin: new data

Exclusive: only one thread has access (initialization mode)

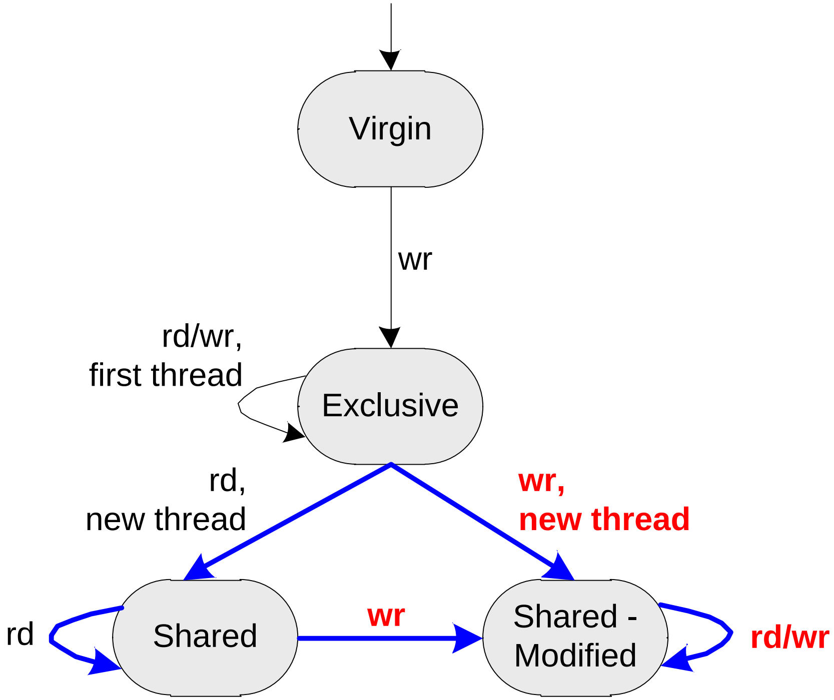
Shared: read-only, after initialization finished

shared-modified: at least one writer and one reader

Start computing lock sets when second thread accesses variable

Report warnings when moving to shared-modified & lock set empty

Side effect: increased dependency on scheduler. (When do we leave Exclusive?)



Example

Thread 1	Thread 2	locks(T1)	locks(T2)	state(v)	C(v)
		∅	∅	VIRGIN	{l1, l2}
`l1.lock();`		{l1}			
`v := 1;`				EXCLUSIVE	
`v := v + 1`					
`l1. unlock()`		∅			
	`l2.lock()`		{l2}		
	`l := v + 1;`			SHARED	{l2}
	`l2.unlock()`		∅		
`l1.lock();`		{l1}			
`l := v + 1;`					∅
`v = l;`				SHARED-MODIFIED	WARNING
`l1. unlock()`		∅			

Eraser, version II

```
//called when thread t reads var v
read(t,v){
  case state(v) of{
    VIRGIN: read before write!;
    EXCLUSIVE:
      if( t != threadid(v) ){
        state(v) = SHARED;
        locks(v) = locks(t); }
    SHARED:
      locks(v) = locks(v) ∩ locks(t);
    SHARED-MODIFIED:
      locks(v) = locks(v) ∩ locks(t);
      if(locks(v) = ∅) emit warning;
  endcase
}
```

Per variable keep:

- state
- when exclusive: thread id
- when shared: lock set

```
//called when thread t writes var v
write(t,v){
  case state(v) of{
    VIRGIN:
      state(v) = EXCLUSIVE;
      threadid(v) = t;
    EXCLUSIVE:
      if(t != threadid(v)){
        state(v) = SHARED-MODIFIED;
        locks(v) = locks(t);
       if(locks(v) = ∅) emit warning;
      }
    SHARED:
      state(v) = SHARED-MODIFED;
      locks(v) = locks(v) ∩ locks(t);
       if(locks(v) = ∅) emit warning;
    SHARED-MODIFIED:
      locks(v) = locks(v) ∩ locks(t);
      if(locks(v) = ∅) emit warning;
  endcase
}
```

Note for C programmers: who needs breaks?

Problem 2: Read/Write Locks

Let’s solve problem 2: simultaneous reads should be allowed

Read-write locks allow for
- multiple simultaneous readers,
- a write is never simultaneous with another read or write.

Useful if you have many reads, regular writes. (Tricky to implement: prevention of starvation for writers)

```
Lock l = new ReentrantReadWriteLock();
// acquire/release l in read mode
l.readLock().lock();
l.readLock().unlock();

// acquire/release l in write mode
l.writeLock().lock();
l.writeLock().unlock();
```

Bank Account, 3

```
class Acct{
  private long balance;
  private long acctNr;
  private ReentrantReadWriteLock l;

  Acct(){
    acctNr = Acct.getNewNr();
    balance = 0;
    l = new ReentrantReadWriteLock();
  }

  long getAcctNr(){
    return acctNr;
  }
```

```
long getBalance(){
    long currentBalance;

    l.readLock().lock();
    currentBalance = balance;
    l.readLock().unlock();
    return currentBalace;
  }


void deposit(long amount){
    long current;

    l.writeLock().lock();
    current = balance;
    current += depositAmount;
    balance = current;
    l.writeLock().unlock();
  } }
```

Problem

Lockset does not work properly

Bank account is correct, but

- write lock is not always held and
- always holding read lock is not enough (a write with just a read lock would be a problem)

Lockset for Read/Write Locks

Let *locks(t)* be the set of locks held by *t*
Let *write_locks(t)* be the set of write locks held by *t*

```
For each variable v {
  C(v) = all_locks;
}

read(t,v){
  C(v) := C(v) ∩ locks(t);
  if C(v) = ∅ then issue warning;
}

wite(t,v){
  C(v) := C(v) ∩ write_locks(t);
  if C(v) = ∅ then issue warning;
}
```

Example

Thread 1	**rlocks**	**wlocks**	**Thread 2**	**rlocks**	**wlocks**	**C(v)**
	∅	∅		∅	∅	all locks
`l.rdl.lk()`	{l}					
			`l.rdl.lk()`	{l}		
`read v`						{l}
			`read v`			{l}
`l.rdl.ulk()`	∅					
			`l.rdl.ulk()`	∅		
`l.wl.lk()`		{l}				
`write v`						{l}
`l.wl.ulk()`		∅				
`l.rl.lk()`	{l}					
`write v`						**∅: warning!**

Remaining False Alarms

- Memory reuse: a private memory manager may use a location for one purpose first, then for another purpose. Locks will be different
- Private locks.
- Benign races

Solution: annotations

- `EraserReuse()`
- `Eraser{Read/Write}{Lock/Unlock}()`
- `EraserIgnore{On/Off}()`

Conclusions

Dynamic algorithms

- May give false alarms
- May not find all problems

Locktree finds possible deadlocks

Eraser finds possible race conditions

Little dependence on scheduling: Can find bug in one scheduling by executing another one: *better than testing.*