A Simple Sequential Reasoning Approach for Sound Modular Verification of Mainstream Multithreaded Programs

B. Jacobs Jr. J. Smans F. Piessens W. Schulte

May 11, 2007

Introduction

1. Preventing data races
   - Programming Model
   - Rules Encoding
2. Invariants and Ownership
   - Annotations
   - Rules Encoding
3. Deadlock prevention
   - Principle
   - Practically
   - Rules Encoding
4. Immutable Objects
5. Limitations

Introduction

Writing correct multithreaded programs software is:

1. *difficult* (Flanagan and Qadeer).
2. *notoriously difficult* (Jacobs et al.).
3. *notoriously tricky* (Peyton Jones et al.).

Introduction

Writing correct multithreaded programs software is:

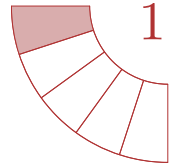
1. *difficult* (Flanagan and Qadeer).
2. *notoriously difficult* (Jacobs et al.).
3. *notoriously tricky* (Peyton Jones et al.).

Goal of this work

- Verify concurrent object-oriented programs statically.
- In a modular way.
- To facilitate verification: a programming model is imposed.

Preventing data races

Programming Model



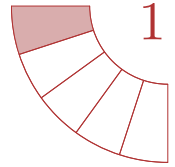
Data race

A *data race* occurs when 2 threads access the same location simultaneously and one of them writes.

- Each thread `t` has an *access set* `t.A`.
- To write to an object, it must be in the current thread's access set. (1)

Preventing data races

Programming Model



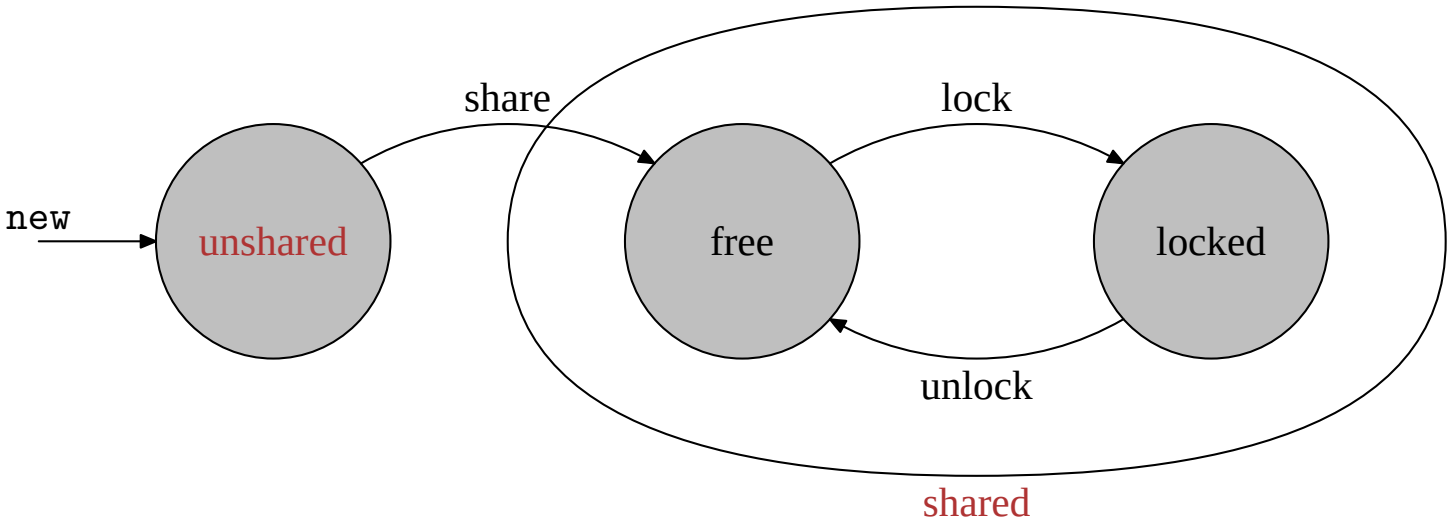
Data race

A *data race* occurs when 2 threads access the same location simultaneously and one of them writes.

- Each thread `t` has an *access set* `t.A`.
- To write to an object, it must be in the current thread's access set. (1)
- An object `o` is in `t`'s access set `t.A` if:
  - It has been created by `t` and it is *unshared*. (2)
  - It is *shared* and locked by `t`. (3)

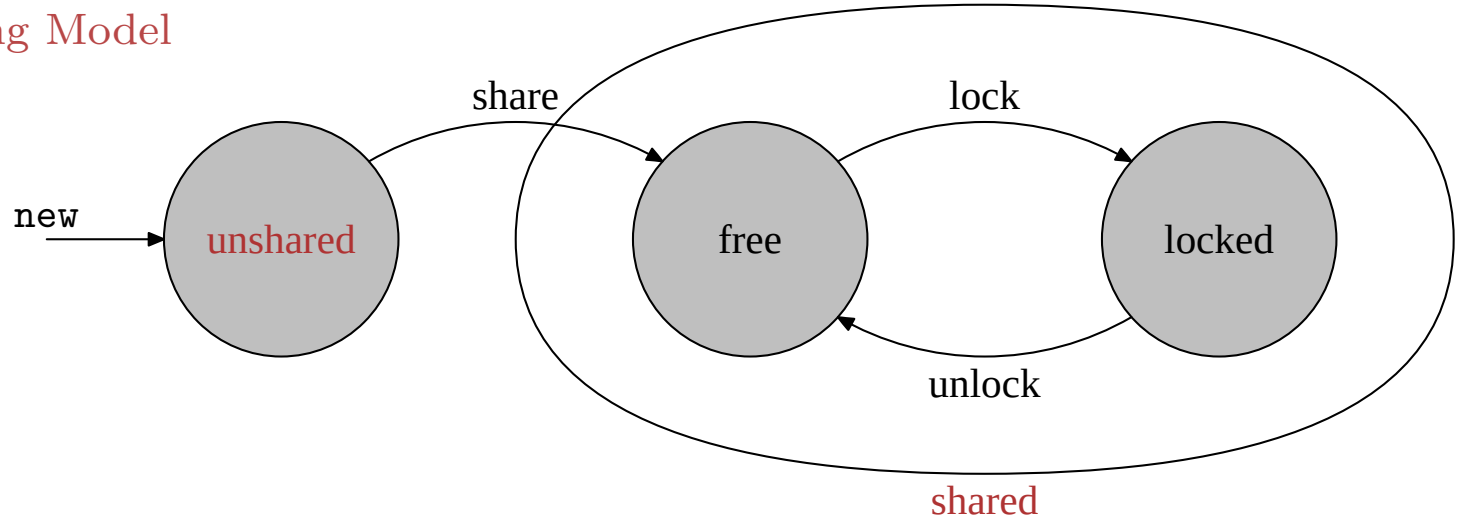
Preventing data races

Programming Model



Preventing data races

Programming Model




```
class Point{
  int x,y;

  requires this ∈ tid.A ∧
           other ∈ this.A;
  void add(Point other){
      ...
  }
}
```

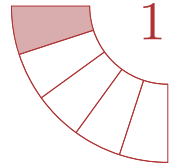
```
void main(){
  Point p = new Point();
  share p;
  new Thread(p);

  synchronized(p){
    ...
  }
}
```

forward

Preventing data races

Rules Encoding




```
o.f := x ≜                                    (1)
  assert o ∈ tid.A;
  if (f is shared)
    assert x ∈ S;
  o.f ← x;

o := new C ≜                                  (2)
  o ← new C;
  assume o ∉ S;
  tid.A ← tid.A ∪ {o};
```

```
synchronized (o) B ≜                          (3)
  assert o ∈ S ∧ o ∉ A;
  havoc o.*;
  tid.A ← tid.A ∪ {o};
    B
  tid.A ← tid.A \ {o};
```

Correct programs ensure access sets of different threads do not intersect.

Invariant

An invariant may depend on:

- Fields of the current class.
- `rep` objects.

Invariants and Ownership

Annotations

Invariant

An invariant may depend on:

- Fields of the current class.
- `rep` objects.

- State of an object = its fields + fields of its `rep` objects,
- and so on recursively.

Invariants and Ownership

Annotations


```
class Point{
  int x,y;

  requires this ∈ tid.A ∧ this.inv;
  ensures  this ∈ tid.A ∧ this.inv;
  void move(int dx,int dy){ ... }
}
```

```
class Rectangle{
  rep Point ul, lr;
  invariant ul.x ≤ lr.x ∧
            ul.y ≥ lr.y;

  requires this ∈ tid.A ∧ this.inv;
  ensures  this ∈ tid.A ∧ this.inv;
  void move(int dx,int dy){
    unpack this;
    ul.move(dx,dy);
    lr.move(dx,dy);
    pack this;
  }
}
```

- Lock-free access to `ul` and `lr`.
- `synchronized` access through their owner (`Rectangle`).

Invariants and Ownership

Rules Encoding

```
unpack o ≜
  assert o ∈ tid.A;
  assert o.inv;
  o.inv ← false;
  foreach(p ∈ repobjects(o)){
    tid.A ← tid.A ∪ {p};
    assume p ∉ S;
  }
```

```
pack o ≜
  assert o ∈ tid.A ∧ ¬ o.inv;
  assert (∀ p ∈ repobjects(o).
     p ∈ tid.A ∧ p ∉ S ∧ p.inv);
  assert Inv(o);
  o.inv ← true;
  foreach(p ∈ repobjects(o))
    tid.A ← tid.A \ {p};
```

Program invariant:

$$\forall o : T, o.inv \Longrightarrow Inv_T(o)$$

Deadlock prevention

Principle

Deadlock

- Thread t has lock l_1 and waits for lock l_2.
- Thread s has lock l_2 and waits for lock l_1.

Deadlock prevention

Principle

Deadlock

- Thread t has lock l_1 and waits for lock l_2.
- Thread s has lock l_2 and waits for lock l_1.
- Both t and s stuck forever.

Deadlock prevention

Principle

Deadlock

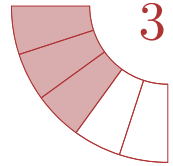
- Thread t has lock l_1 and waits for lock l_2.
- Thread s has lock l_2 and waits for lock l_1.
- Both t and s stuck forever.

Solution:

- Partial order between locks, such as $l_1 < l_2$.
- Threads can only acquire locks in decreasing order.
  - `lock(`l_2`); lock(`l_1`)`
  - `lock(`l_1`); ~~lock(`l_2`)~~`

Deadlock prevention

Practically




```
class Philosopher extends Thread{
   shared Fork fork1, fork2;

   requires fork1.lockLevel < fork2.lockLevel
   Philosopher( shared Fork fork1, shared Fork fork2){ ... }
}

void main(){
    locklevel level1 := between({},{});
    locklevel level2 := between({level1},{});
    locklevel level3 := between({level2},{});
    Fork fork1 = new Fork();
    share(fork1,level1);
    Fork fork2 = new Fork();
    share(fork2,level2);
    Fork fork3 = new Fork();
    share(fork3,level3);
    new Philosopher(fork1,fork2).start();
    new Philosopher(fork2,fork3).start();
    new Philosopher(fork1,fork3).start();
}
```

Deadlock prevention

Rules Encoding


```
share(o,l) ≡
  assert o ∈ tid.A ∧ o ∉ S;
  assert o.inv;
  o.lockLevel ← l;
  tid.A ← tid.A \ {o};
  S ← S ∪ {o};
```

```
synchronized(o) B ≡
  assert o ∈ S;
  assert o.lockLevel <
           tid.lockStack.top();
  tid.lockStack.push(o);
  havoc o.*;
  tid.A ← tid.A ∪ {o};
    B
  tid.A ← tid.A \ {o};
  tid.lockStack.pop();
```

Immutable Objects

Immutable object

- An object that is never written (i.e. only read) after its initialization.
- Therefore, immutable objects can be accessed without any synchronization.

Immutable Objects

Immutable object

- An object that is never written (i.e. only read) after its initialization.
- Therefore, immutable objects can be accessed without any synchronization.

- Access set split into *read set* and *write set*.
- `share` replaced by `share_immutable` and `share_lockprotected`.
- When an immutable object is shared, its invariant must hold therefore it holds at all times.

Limitations

1. Once shared, an object can never revert to the unshared state (problematic with fork/join patterns).
2. Lock reentrancy (default Java's `synchronized` behavior) forbidden.
3. Protection by locking only provided by `this`.
4. Truly concurrent objects (i.e. objects where multiple threads can execute simultaneously) forbidden.

back