# Lecture 5

# Towards a Verifying Compiler: Multithreading

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Race Conditions, Locks,
Deadlocks, Invariants, Locklevels
Access Sets

---

Joint work with Rustan Leino, Mike Barnett, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and Peter Müller (ETH), Bart Jacobs (KU Leuven) and Bor-Yuh Evan Chung (Berkley) .

## Review: Pure Methods and Model Fields

Data abstraction is crucial to express functional correctness properties

- Verification methodology for model fields
  - *Model fields are reduced to ordinary fields with automatic updates*
- Verification challenges for model fields and pure methods
  - *Consistency*
  - *Weak purity*
  - *Heap dependence (and frame properties)*

## Multi-threading

- **Data race prevention**
- Invariants and ownership trees
- Deadlock prevention

# Multithreading

Multiple threads running in parallel, reading and writing shared data

A *data race* occurs when a shared variable is written by one thread and concurrently read or written by another thread

How to guarantee that there are no data races?

```
class Counter {
  int dangerous;
  void Inc() {
    int tmp = dangerous;
    dangerous = tmp + 1; }
}

Counter ct = new Counter();
new Thread(ct.Inc).Start();
new Thread(ct.Inc).Start();
//  What is the value of
//   ct.dangerous after both
//   threads have terminated?
```

## Mutexes: Avoiding Races

- *Mutual exclusion* for shared objects is provided *via locks*

- Locks can be obtained using a *lock block*. A thread may enter a lock (o) block only if no other thread is executing inside a lock (o) block; else, the thread waits

- When *a thread holds a lock on object o*, *C#/Java*
  - do prevent other threads from locking o but
  - *do not prevent other threads from accessing o's fields*

# Program Method for Avoiding Races

Our program rules enforce that
*a thread* t *can only access a field of object* o *if* o *is either thread local or* t *has locked* o

We model accessibility using *access sets*:

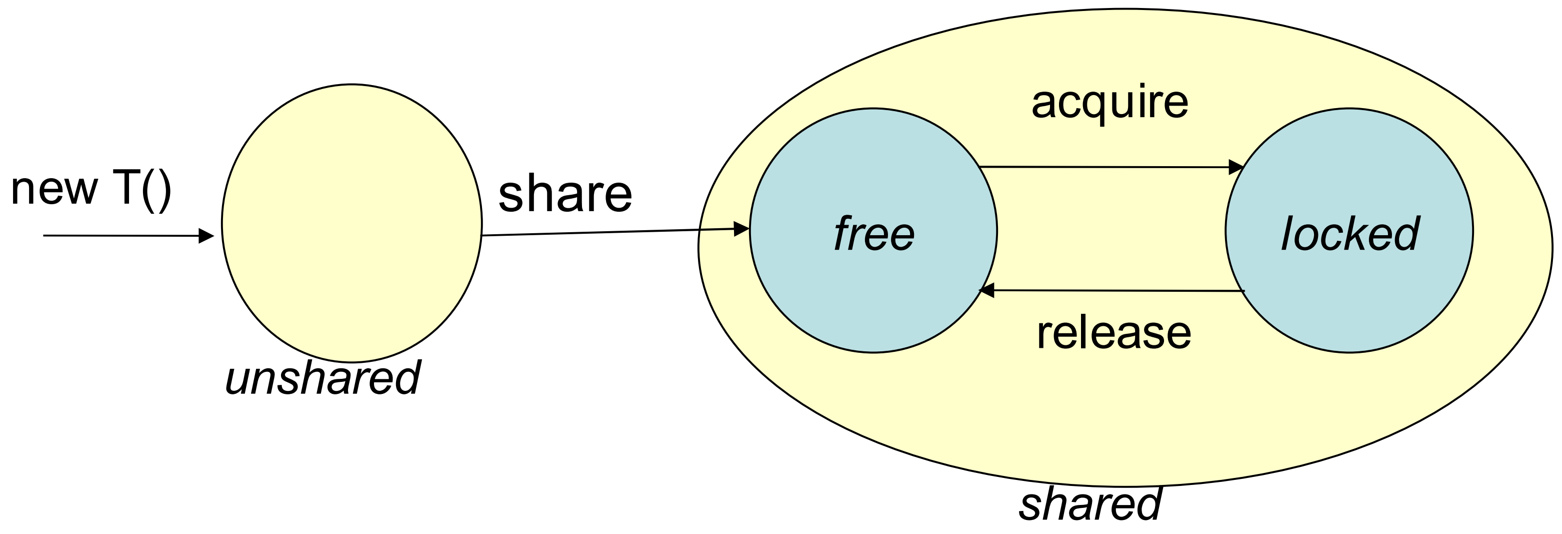
- A thread's access set consists of all objects it has created but not shared yet or whose lock it holds.
- Threads are only allowed to access fields of objects in their corresponding access set

Our program rules prevent data races by ensuring that *access sets of different threads never intersect*.

## Annotations Needed to Avoid Races

- Threads have access sets
  - t.A is a new ghost field in each thread t describing the set of accessible objects
- Objects can be shared
  - o.shared is a new boolean ghost field in each object o
  - share(o) is a new operation that shares an unshared o
- Fields can be declared to be shared
  - Shared fields can only be assigned shared objects.

# Object Life Cycle



## Verification via Access Sets

```
Tr[[o = new C();]] = …
  o.shared:= false;
  tid.A[o]:= true

Tr[[x = o.f;]] = …
  assert tid.A[o];
  x :=o.f;

Tr[[o.f = x;]] = …
  assert tid.A[o];
  if (f is declared shared)
    assert x.shared;
  o.f :=x;
```

```
Tr[[share(o)]] = …
  assert tid.A[o];
  assert ! o.shared;
  o.shared :=true;
  tid.A[o] :=false;

Tr[[lock (o) S  ]] = …
  assert ! tid.A[o];
  assert o.shared;
  havoc o.*;
  tid.A[o]:=true;
 Tr[[S]];
  tid.A[o]:= false
```

# A Note on havoc in the Lock Rule

When a thread (re) acquires o, o might have been changed by another thread.

```
int x;
lock (o) {
  x = o.f;
}
lock (o) {
  assert x == o.f;   // fails
}
```

So we have to “forget all knowledge about o’s fields”. We do so by assigning an arbitrary value to all of o’s field, expressed as

havoc o.*

## Example for Data Race Freedom

```
Counter ct = new Counter();
share(ct);
new Thread(delegate () { lock (ct) ct.Inc(); }).Start();
new Thread(delegate () { lock (ct) ct.Inc(); }).Start();
```

# Example for Data Race Freedom

```
// thread t0
  Counter ct = new Counter();
  share(ct);
  Session s1 =new Session(ct,1);
  Session s2 =new Session(ct,2);
  // transfers s1 to t1
     t1 = new Thread(s1.Run);
  // transfers s2 to t2
     t2 = new Thread(s2.Run);
  t1.Start();
  t2.Start();
```

```
class Session {
  shared Counter ct ;
  int id;

  Session(Counter ct , int id)
    requires ct.shared;
    ensures tid.A[this] ∧ ! this.shared;
  { this.ct=ct; this.id=id; }

  void Run()
     requires tid.A[this];
  { for (; ; )
       lock (this.ct)
          this.ct.Inc();
  }
}
```

## Soundness

Theorem

- $\forall$ threads t1,t2 :: $t1 \neq t2 \Rightarrow t1.A \cap t2.A = \emptyset$
- $\forall$ object o, thread t :: o.shared && $o \in t.A \Rightarrow$ t holds o’s lock

- Proof sketch for Theorem
  - new
  - share (o)
  - Entry into lock (o)
  - Exit from lock (o)

Corollary

- Valid programs don’t have data races

## Multi-threading

- Data race prevention
- **Invariants and ownership trees**
- Deadlock prevention

## Invariants and Concurrency

*Invariants*, whether over a single object or over an ownership tree, can be *protected via a single lock* (coarse grained locking)

For sharing and locking

- require an object *o to be valid when o becomes free*
- ensures o's invariant on entry to its locked state

For owned objects

- require that commited objects are unaccessable, but
  - unpack(o) adds o's owned objects to the thread's access set
  - pack(o) deletes o's owned objects from the thread's access set

# Verifying Multi-threaded Pack/Unpack

```
Tr[[unpack o;]] =
    assert tid.A[o];
    assert o.inv;
    foreach (c | c.owner = o)
      { tid.A[c] := true; }
    o.inv := false;
```

```
Tr[[ pack o;]] =
    assert tid.A[o];
    assert ! o.inv;
    assert $\forall$c: c.owner = o $\Rightarrow$
         tid.A[c] $\wedge$ c.inv;
    foreach (c | c.owner = o)
        { tid.A[c] := false; }
    assert Inv( o );
    o.inv := true;
```

## Ownership: Verifying Lock Blocks

Finally, when locking we also have to “forget the knowledge about” owned objects

```
Tr[[lock (o) S;  ]] =
        assert o.shared;
        assert ! tid.A[o];
        foreach (p | !tid.A[p])  havoc p.*;
        tid.A[o]:=true;
        Tr[[S]] ;
        tid.A[o]:= false;
```

## Outline of the talk

- Data race prevention
- Invariants and ownership trees
- **Deadlock prevention**

## Multi-threading

- Data race prevention
- **Invariants and ownership trees**
- Deadlock prevention

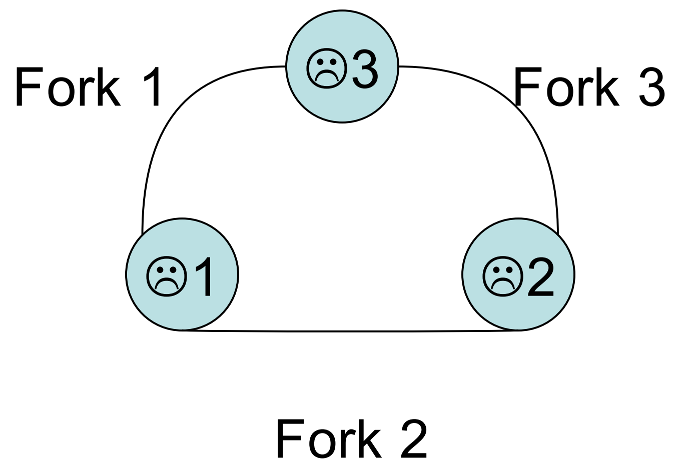
# Concurrency: Deadlocks

A *deadlock* occurs when a set of threads each wait for a mutex (i.e shared object) that another thread holds

Methodology:

- *partial order over all shared objects*
- in each thread, *acquire shared objects in descending order*

Dining Philosophers



☹1 has F1, waits for F2
☹2 has F2, waits for F3
☹3 has F3, waits for F1

## Annotations Needed to Avoid Deadlocks

*We construct a partial order on shared objects, denoted by .*

- When o is shared, we add edges to the partial order as specified in the share command's where clause.

  (Specified lower bounds have to be less than specified upper bounds)

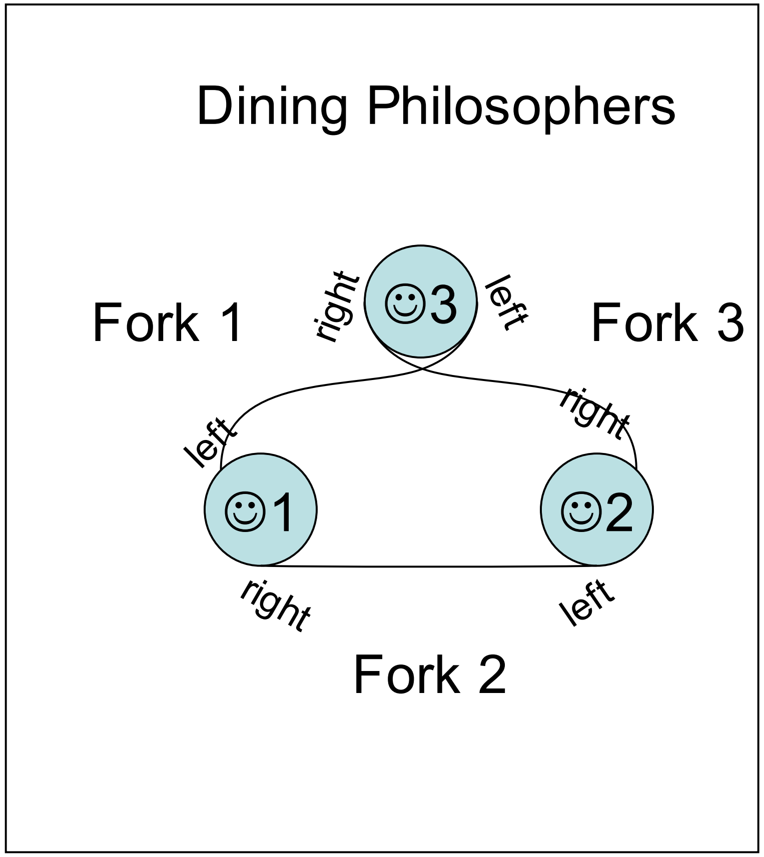
- Each thread has a new ghost field *lockstack*, holding the set of acquired locks

# Verification via Lock Ordering and Lockstacks

```
Tr[[share o
      where p     o && o     q;]] =
  assert o ∈ tid.A;
  assert ! o.shared;
  tid.A[o] := false;
  o.shared := true;
  assert p     q;
  assume p     o && o     q;
```

```
Tr[[lock (o) S  ]] =
 assert o.shared;
 assert tid.lockstack != empty ⇒
    o     tid.lockstack.top();
  tid.lockStack.push(o);
  foreach (p | !tid.A[p])  havoc p.*;
  tid.A[o]:=true;
  Tr[[S]] ;
  tid.A[o]:= false;
  tid.lockstack.pop(o);
```

# Example: Deadlock Avoidance (contd.)



```
f1 = new Fork(); share f1;
f2 = new Fork(); share f2 where f1    f2;
f3 = new Fork(); share f3 where f2    f3 ;

P1 = new Thread( delegate() {
     lock (f2) { lock (f1) { /*eat*/ }}});
    P1.Start();
P2 = new Thread( delegate() {
    lock (f3) { lock (f2) {/*eat*/ }}}); P2.Start();
P3 = new Thread( delegate() {
    lock (f3) { lock (f1) {/*eat*/ }}}); P3.Start();
```

## Conclusion

- Clients can reason entirely as if world was single-threaded for non-shared objects
- Supports caller-side locking and callee-side locking
- Deadlocks are prevented by partially ordering shared objects

## The End

(for now)

Thank you!

http://research.micsoft.com/specsharp

# Lecture 5

# Towards a Verifying Compiler: Multithreading

Wolfram Schulte
Microsoft Research
Formal Methods 2006

Race Conditions, Locks,
Deadlocks, Invariants, Locklevels
Access Sets

Joint work with Rustan Leino, Mike Barnett, Manuel Fähndrich, Herman Venter, Rob DeLine, Wolfram Schulte (all MSR), and Peter Müller (ETH), Bart Jacobs (KU Leuven) and Bor-Yuh Evan Chung (Berkley) .

## Review: Pure Methods and Model Fields

Data abstraction is crucial to express functional correctness properties

- Verification methodology for model fields
  - *Model fields are reduced to ordinary fields with automatic updates*
- Verification challenges for model fields and pure methods
  - *Consistency*
  - *Weak purity*
  - *Heap dependence (and frame properties)*

## Multi-threading

- **Data race prevention**
- Invariants and ownership trees
- Deadlock prevention

## Multithreading

Multiple threads running in parallel, reading and writing shared data

A *data race* occurs when a shared variable is written by one thread and concurrently read or written by another thread

How to guarantee that there are no data races?

```
class Counter {
  int dangerous;
  void Inc() {
    int tmp = dangerous;
    dangerous = tmp + 1; }
}

Counter ct = new Counter();
new Thread(ct.Inc).Start();
new Thread(ct.Inc).Start();
//  What is the value of
//   ct.dangerous after both
//   threads have terminated?
```

## Mutexes: Avoiding Races

- *Mutual exclusion* for shared objects is provided *via locks*

- Locks can be obtained using a *lock block*. A thread may enter a lock (o) block only if no other thread is executing inside a lock (o) block; else, the thread waits

- When *a thread holds a lock on object o, C#/Java*
  - do prevent other threads from locking o but
  - *do not prevent other threads from accessing o's fields*

## Program Method for Avoiding Races

Our program rules enforce that
*a thread* t *can only access a field of object* o *if* o *is either thread local or* t *has locked* o

We model accessibility using *access sets*:

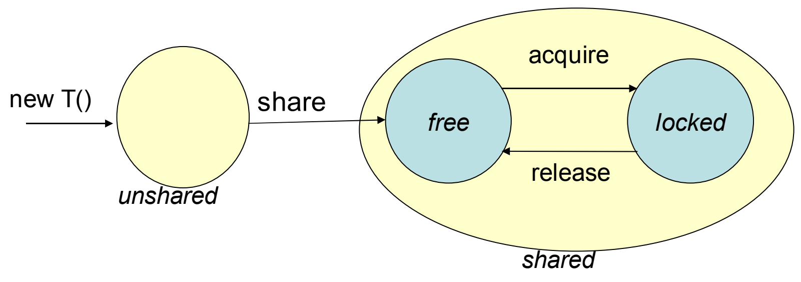
- A thread's access set consists of all objects it has created but not shared yet or whose lock it holds.
- Threads are only allowed to access fields of objects in their corresponding access set

Our program rules prevent data races by ensuring that *access sets of different threads never intersect*.

## Annotations Needed to Avoid Races

- Threads have access sets
  - t.A is a new ghost field in each thread t describing the set of accessible objects
- Objects can be shared
  - o.shared is a new boolean ghost field in each object o
  - share(o) is a new operation that shares an unshared o
- Fields can be declared to be shared
  - Shared fields can only be assigned shared objects.

## Object Life Cycle



## Verification via Access Sets

```
Tr[[o = new C();]] = …
  o.shared:= false;
  tid.A[o]:= true

Tr[[x = o.f;]] = …
  assert tid.A[o];
  x :=o.f;

Tr[[o.f = x;]] = …
  assert tid.A[o];
  if (f is declared shared)
    assert x.shared;
  o.f :=x;
```

```
Tr[[share(o)]] = …
  assert tid.A[o];
  assert ! o.shared;
  o.shared :=true;
  tid.A[o] :=false;


Tr[[lock (o) S  ]] = …
  assert ! tid.A[o];
  assert o.shared;
  havoc o.*;
  tid.A[o]:=true;
 Tr[[S]];
  tid.A[o]:= false
```

## A Note on havoc in the Lock Rule

When a thread (re) acquires o, o might have been changed by another thread.

```
int x;
lock (o) {
  x = o.f;
}
lock (o) {
  assert x == o.f;   // fails
}
```

So we have to “forget all knowledge about o’s fields”. We do so by assigning an arbitrary value to all of o’s field, expressed as

havoc o.*

## Example for Data Race Freedom

```
Counter ct = new Counter();
share(ct);
new Thread(delegate () { lock (ct) ct.Inc(); }).Start();
new Thread(delegate () { lock (ct) ct.Inc(); }).Start();
```

# Example for Data Race Freedom

```
// thread t0
  Counter ct = new Counter();
  share(ct);
  Session s1 =new Session(ct,1);
  Session s2 =new Session(ct,2);
  // transfers s1 to t1
     t1 = new Thread(s1.Run);
  // transfers s2 to t2
     t2 = new Thread(s2.Run);
  t1.Start();
  t2.Start();
```

```
class Session {
  shared Counter ct ;
  int id;

  Session(Counter ct , int id)
    requires ct.shared;
    ensures tid.A[this] ∧ ! this.shared;
  { this.ct=ct; this.id=id; }

  void Run()
     requires tid.A[this];
  { for (; ; )
      lock (this.ct)
         this.ct.Inc();
  }
}
```

## Soundness

Theorem

- ∀ threads t1,t2 :: $t1 \neq t2 \Rightarrow t1.A \cap t2.A = \varnothing$
- ∀ object o, thread t :: o.shared && $o \in t.A \Rightarrow$ t holds o's lock

- Proof sketch for Theorem
  - new
  - share (o)
  - Entry into lock (o)
  - Exit from lock (o)

Corollary

- Valid programs don't have data races

## Multi-threading

- Data race prevention
- **Invariants and ownership trees**
- Deadlock prevention

## Invariants and Concurrency

*Invariants*, whether over a single object or over an ownership tree, can be *protected via a single lock* (coarse grained locking)

For sharing and locking

- require an object *o to be valid when o becomes free*
- ensures o’s invariant on entry to its locked state

For owned objects

- require that commited objects are unaccessable, but
  - unpack(o) adds o’s owned objects to the thread’s access set
  - pack(o) deletes o’s owned objects from the thread’s access set

## Verifying Multi-threaded Pack/Unpack

```
Tr[[unpack o;]] =
    assert tid.A[o];
    assert o.inv;
    foreach (c | c.owner = o)
      { tid.A[c] := true; }
   o.inv := false;
```

```
Tr[[ pack o;]] =
    assert tid.A[o];
    assert ! o.inv;
    assert ∀c: c.owner = o ⇒
         tid.A[c] ∧ c.inv;
    foreach (c | c.owner = o)
        { tid.A[c] := false; }
     assert Inv( o );
     o.inv := true;
```

## Ownership: Verifying Lock Blocks

Finally, when locking we also have to “forget the knowledge about” owned objects

```
Tr[[lock (o) S;  ]] =
      assert o.shared;
      assert ! tid.A[o];
      foreach (p | !tid.A[p])  havoc p.*;
      tid.A[o]:=true;
      Tr[[S]] ;
      tid.A[o]:= false;
```

## Outline of the talk

- Data race prevention
- Invariants and ownership trees
- **Deadlock prevention**

## Multi-threading

- Data race prevention
- **Invariants and ownership trees**
- Deadlock prevention

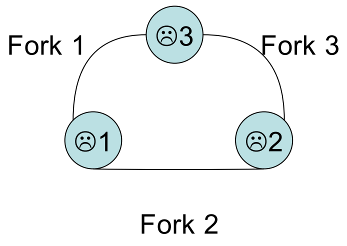
## Concurrency: Deadlocks

A *deadlock* occurs when a set of threads each wait for a mutex (i.e shared object) that another thread holds

Methodology:

- *partial order over all shared objects*
- in each thread, *acquire shared objects in descending order*

Dining Philosophers



☹1 has F1, waits for F2
☹2 has F2, waits for F3
☹3 has F3, waits for F1

## Annotations Needed to Avoid Deadlocks

*We construct a partial order on shared objects, denoted by .*

- When o is shared, we add edges to the partial order as specified in the share command's where clause.

  (Specified lower bounds have to be less than specified upper bounds)

- Each thread has a new ghost field *lockstack*, holding the set of acquired locks

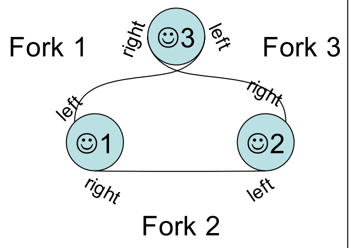
# Verification via Lock Ordering and Lockstacks

```
Tr[[share o
       where p    o && o    q;]] =
  assert o ∈ tid.A;
  assert ! o.shared;
  tid.A[o] := false;
  o.shared := true;
  assert p    q;
  assume p    o && o    q;
```

```
Tr[[lock (o) S  ]] =
 assert o.shared;
 assert tid.lockstack != empty ⇒
    o    tid.lockstack.top();
  tid.lockStack.push(o);
  foreach (p | !tid.A[p])  havoc p.*;
  tid.A[o]:=true;
  Tr[[S]] ;
  tid.A[o]:= false;
  tid.lockstack.pop(o);
```

## Example: Deadlock Avoidance (contd.)

Dining Philosophers



```
f1 = new Fork(); share f1;
f2 = new Fork(); share f2 where f1     f2;
f3 = new Fork(); share f3 where f2     f3 ;

P1 = new Thread( delegate() {
      lock (f2) { lock (f1) { /*eat*/ }}});
     P1.Start();
P2 = new Thread( delegate() {
     lock (f3) { lock (f2) {/*eat*/ }}}); P2.Start();
P3 = new Thread( delegate() {
     lock (f3) { lock (f1) {/*eat*/ }}}); P3.Start();
```

## Conclusion

- Clients can reason entirely as if world was single-threaded for non-shared objects
- Supports caller-side locking and callee-side locking
- Deadlocks are prevented by partially ordering shared objects

## The End
(for now)

Thank you!

http://research.micsoft.com/specsharp