Microsoft Research

A Foundation for Verifying Concurrent Programs

K. Rustan M. Leino
RiSE, Microsoft Research, Redmond

joint work with Peter Müller and Jan Smans

Lecture 0
1 September 2009
FOSAD 2009, Bertinoro, Italy

Program verification

- Prove program correctness
  for all possible inputs and behaviors

Modular verification

- Prove parts of a program separately
- Correctness of every part
  implies
  correctness of whole program

Specifications

- Record programmer design decisions
- Describe usage of program constructs
- Provide redundancy
- Enable modular verification

Specification style

- Specification and verification methodology
- Describes properties of the heap
- Active area of research
  - Ownership
    - Spec#, Java+JML, vcc, type systems, …
  - Dynamic frames
    - VeriCool, Dafny
  - Permissions (capabilities)
    - Effect systems, separation logic, VeriCool 3, Chalice, …

Microsoft Research

Concurrent programs

- Interleaving of thread executions
- Unbounded number of: threads, locks, …
- We need some basis for doing the reasoning
  - A way of thinking!

These lectures

- Concurrent programs
  - Features like: threads, monitors, abstraction as well as: objects, methods, loops, …
  - Avoid errors like: race conditions, deadlocks
- Specifications with permissions
- Building a program verifier

demo

Square

Pre- and postconditions

demo

Cube

Loop invariants

demo

ISqrt

Chalice

Specifications at run time

- Helps testing find bugs more quickly
- Optional, they can be treated as ghosts
- If they are to be ghosted, specifications must have no side effects (on non-ghost state)

Microsoft Research

Dealing with memory (the heap)

- Access to a memory location requires permission
- Permissions are held by activation records
- Syntax for talking about permission to y: acc(y)

demo

Inc

Permissions

Transfer of permissions

```
method Main()
{
   var c := new Counter;
   call c.Inc();
}
```

acc(c.y)

```
method Inc()
   requires acc(y)
   ensures acc(y)
{
   y := y + 1;
}
```

Well-formed specifications

- A specification expression can mention a memory location only if it also entails the permission to that location
- acc(y) && y < 100 ✓
- y < 100 ✗
- acc(x) && y < 100 ✗
- acc(o.y) && p.y < 100 ✗
- o == p && acc(o.y) && p.y < 100 ✓
- x / y < 20 ✗
- y ≠ 0 && x / y < 20 ✓

Microsoft Research

Loop invariants and permissions

- A loop iteration is like its own activation record

```
Before;
while (B) invariant J { S; }
After;
```

is like

```
Before;
call MyLoop(…);
After;
```

```
method MyLoop(…)
  requires J
  ensures J
{
  if (B) {
    S;
    call MyLoop(…);
  }
}
```

Loop invariant: example

```
method M()
   requires acc(x) && acc(y) && x <= 100 && y <= 100
{
   while (y < 100)
      invariant acc(y) && y <= 100
   {
      y := y + 1;
      x := x + 1;  // error: no permission to access x
   }
   assert x <= y;
}
```

Loop invariant: example

```
method M()
   requires acc(x) && acc(y) && x <= 100 && y <= 100
{
   while (y < 100)
      invariant acc(y) && y <= 100
   {
      y := y + 1;

   }
   assert x <= y;
}
```

Microsoft Research

demo

ISqrt with fields

Loop invariants with permissions

Threads

- Threads run concurrently
- A new thread of control is started with the fork statement
- A thread can wait for another to complete with the join statement
- Permissions are transferred to and from a thread via the starting method's pre- and postconditions

demo

ForkInc

Fork and join

The two halves of a call

- call == fork + join

```
call x,y := o.M(E, F);
```

is semantically like

```
fork tk := o.M(E, F);
join x,y := tk;
```

… but is implemented more efficiently

Microsoft Research

demo

TwoSqrts

Parallel computation

Well-formed revisited

- Recall:
  A specification expression can mention a memory location only if it also entails some permission to that location

- Example: acc(y) && y < 100 ✓

- Without any permission to y, other threads may change y, and then y would not be stable

Read permissions

- acc(y)   write permission to y
- rd(y)   read permission to y

- At any one time, at most one thread can have write permission to a location

demo

VideoRental

Parallel reads

Fractional permissions

- acc(y)   100% permission to y
- acc(y, p)   p% permission to y
- rd(y)   read permission to y
- Write access requires 100%
- Read access requires >0%
- acc(y) = acc(y,69) + acc(y,31)
- rd(y) ≈ acc(y, ε)

Microsoft Research

Implicit dynamic frames

- method M() requires acc(y) ensures acc(y) can change y
- Can
  method P() requires rd(y) ensures rd(y) change y?
- That is, can we prove:

```
method Q()
  requires rd(y) && y == 5
{
  call P();
  assert y == 5;
}
```

Demo: NoPerm

Microsoft Research

Shared state

- What if two threads want write access to the same location?

```
class Fib {
  var y: int;
  method Main()
  {
    var c := new Fib;
    fork c.A();
    fork c.B();
  }
}
```

acc(c.y)

```
method A() …
{
   y := y + 21;
}
```

```
method B() …
{
   y := y + 34;
}
```

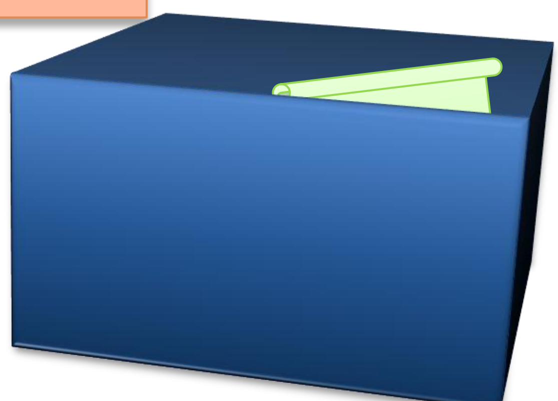
Monitors

```
class Fib {
  var y: int;
  invariant acc(y);
  method Main()
  {
    var c := new
    share c;
    fork c.A();
    fork c.B();
  }
}
```

acc(c.y)

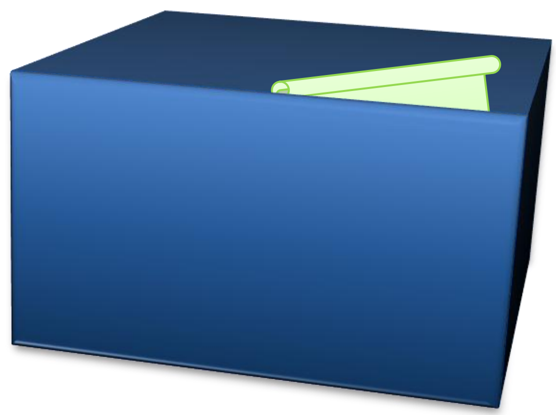
```
method A() …
{
   acquire this;
   y := y + 21;
   release this;
}
```

```
method B() …
{
   acquire this;
   y := y + 34;
   release this;
}
```



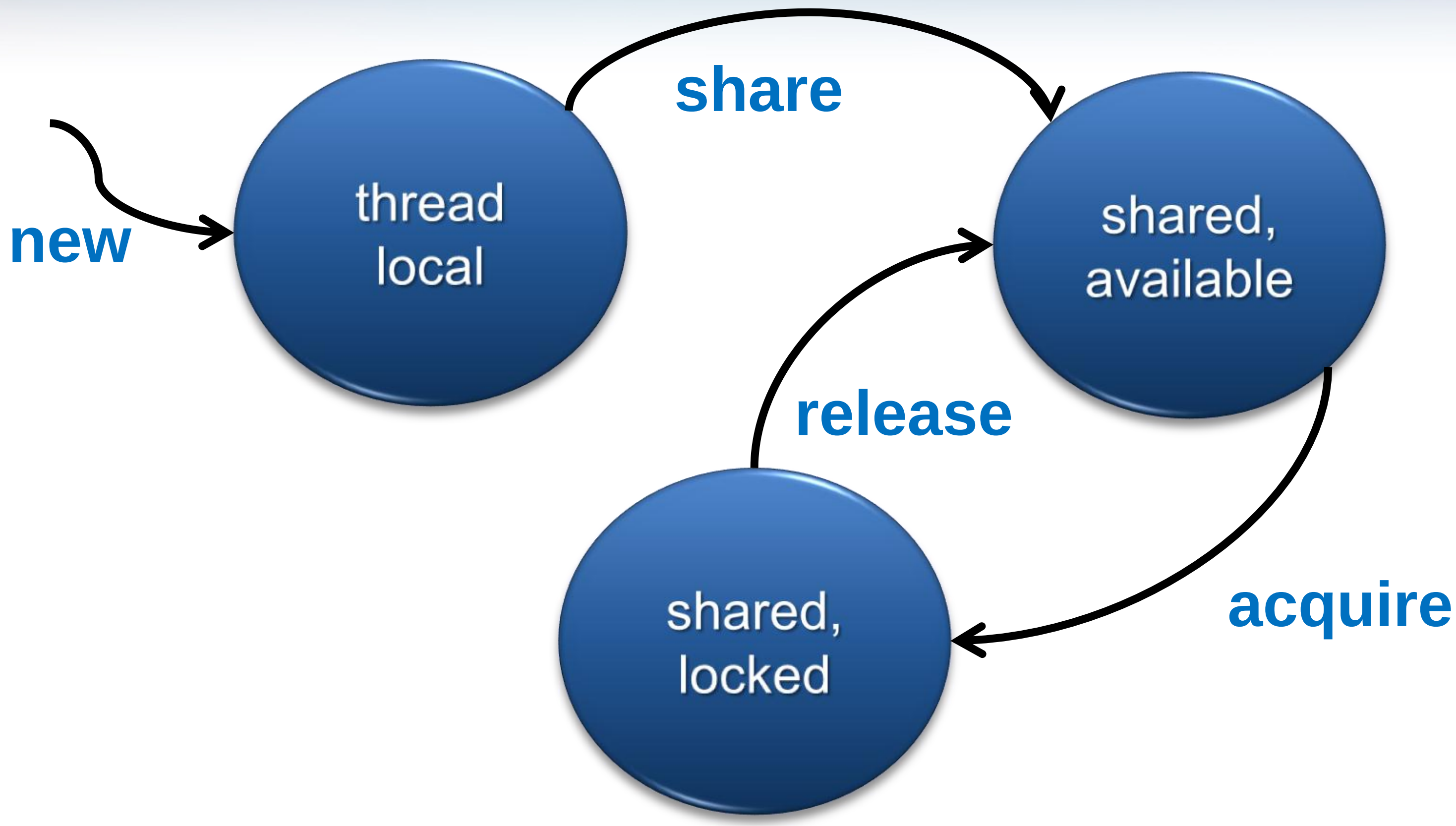
Monitor invariants

- Like other specifications, can hold both permissions and conditions
- Example: invariant acc(y) && 0 <= y



Microsoft Research

Object life cycle



Microsoft Research

demo

SharedCounter

Monitors

Locks and permissions

- The concepts
  - holding a lock, and
  - having permissions

  are orthogonal to one another
- In particular:
  - Holding a lock does not imply any right to read or modify shared variables
- Their connection is:
  - Acquiring a lock obtains some permissions
  - Releasing a lock gives up some permissions

Thread-safe libraries

- Server-side locking
  - "safer" (requires less thinking)

```
invariant acc(y);
method M()
   requires true
{
   acquire this;  y := …;  release this;
}
```

- Client-side locking
  - more efficient

```
method M()
   requires acc(y)
{
   y := …;
}
```

Microsoft Research