Construction and Verification of Software

2021 - 2022

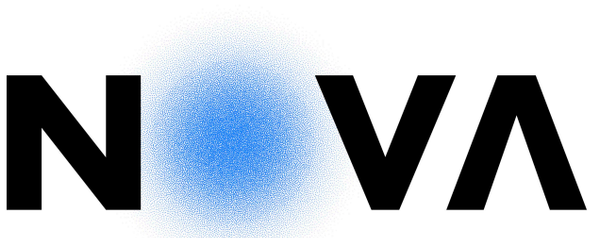
MIEI - Integrated Master in Computer Science and Informatics

Consolidation block

Lecture 10 - Resource Sharing

Bernardo Toninho (btoninho@fct.unl.pt)

based on previous editions by **João Seco** and **Luís Caires**



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Outline

- Linear Resource Usage
- Fractional permissions
- Shared access to locks using fractional permissions
- More concurrency: N readers - 1 writer

Part I

Resource Sharing

Resource sharing

- (Conservative) exclusive access to resources is sound in Concurrent ADTs, i.e. it avoids any kind of interference.
- However, not all interferences and sharing situations are harmful.
- One crucial example where aliasing and sharing is safe, and many times necessary, is when reading information from memory.
- Recall the example from the previous lecture: `Account`.

Account ADT (Java + Verifast)

RECAP

```
public class Account {

  int balance;

  /*@
  predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;
  @*/

  public Account()
  //@ requires true;
  //@ ensures AccountInv(0);
  {
    balance = 0;
  }

  ...

}
```

Account ADT (Java + Verifast)

RECAP

```
public class Account {

  int balance;


  ...
  void deposit(int v)
  //@ requires AccountInv(?b) &*& v>=0;
  //@ ensures AccountInv(b+v);
  {
    balance += v;
  }

  void withdraw(int v)
  //@ requires AccountInv(?b) &*& b >=v;
  //@ ensures AccountInv(b-v);
  {
    balance -= v;
  }
  ...
}
```

Account ADT (Java + Verifast)

RECAP

```
public class Account {

  int balance;


  ...
  int getBalance()
  //@ requires AccountInv(?b);
  //@ ensures AccountInv(b) &*& result==b &*& b >= 0;
  {
    return balance;
  }

  ...

}
```

Account ADT (Java + Verifast)

```
public class Account {

  int balance;
  ...
  static void test2(Account a1, Account a2)
  //@ requires a1.AccountInv(?b1) &*& a2.AccountInv(_) ;
  //@ ensures a1.AccountInv(b1) &*& a2.AccountInv(_) ;
  {
    int v1;
    v1 = a1.getBalance();
    a2.deposit(v1);
  }
  ...
}
```

- Separation logic can prove that `a1` does not change
- The frame principle of SL and the separating conjunction supports local reasoning. Only focusing on the exact footprint of the program fragment.

Account ADT (Java + Verifast)

```
public class Account {

  int balance;
  ...
  static void test2(AccountS a1, AccountS a2)
  //@ requires a1.AccountInv(?b1) &*& a2.AccountInv(_) ;
  //@ ensures a1.AccountInv(b1) &*& a2.AccountInv(_) ;
  {
    int v1;
    //@ assert a1.AccountInv(?b1) &*& a2.AccountInv(_)
    //@ assert a1.balance |-> ?b1 &*& b1>=0 &*& a2.balance |-> ?b2 &*& b2>=0;
    //@ assert a1.AccountInv(b1) &*& a2.balance |-> ?b2 &*& b2>=0;
    v1 = a1.getBalance();

    //@ assert a1.AccountInv(b1) &*& v1>=0 &*& a2.balance |-> ?b2 &*& b2>=0;
    //@ assert a1.AccountInv(b1) &*& v1>=0 &*& a2.AccountInv(b2);

    a2.deposit(v1);
    //@ assert a1.AccountInv(b1) &*& v1>=0 &*& a2.AccountInv(b2+v1);
  }
  ...
}
```

Account ADT (Java + Verifast)

```
public class Account {

  int balance;
  ...
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    b1.deposit(10);
    //@ assert b1.Account(?b) &*& b2.Account(_);
    test2(b1,b2);
    //@ assert b1.Account(b) &*& b2.Account(_);
  }
  ...
}
```

- The precondition of test2 holds: we are sure that b1 and b2 are not aliases (separated memory heap chunks)

Account ADT (Java + Verifast)

```
public class Account {

  int balance;
  ...
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    AccountS b1 = new Account();
    AccountS b2 = new Account();
    b1.deposit(10);
    //@ assert b1.Account(?b) &*& b2.Account(_);
    test2(b1,b1); // ERROR, the precondition of test2 does not hold;

  }
  ...
}
```

- Precondition of test2 is not valid now, we cannot show that b1.Account(_) and b2.Account(_) are separated

Separation and Sharing

- The use of A*B solves a limitation of Hoare Logic:
  it helps to keep track of aliases and memory footprints, as needed to check programs in C or Java.
- However, the use of A*B in Separation Logic forces all usages of memory references to be used “linearly”.
- Each usage of a memory cell requires a “permission” c.N |-> v, and there is only one permission around for each memory cell.
- Thus memory cells cannot be shared or aliased, unless in trivial contexts (where the reference is passed around but not really used for reading or writing).
- However, there are of course situations in which sharing and aliasing is safe and necessary.

Account ADT (Java + Verifast)

```
public class Account {

  static int sum(AccountS a1, AccountS a2)
  //@ requires ??;
  //@ ensures ??;
  {
    return a1.getBalance()+a2.getBalance();
  }

  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b2);
  }
}
```

Which contract should we assign to method sum ?

Account ADT (Java + Verifast)

```
public class Account {

  static int sum(AccountS a1, AccountS a2)
  //@ requires a1.Account(?b1) &*& a2.Account(?b2);
  //@ ensures a1.Account(b1) &*& a2.Account(b2);;
  {
    return a1.getBalance()+a2.getBalance();
  }

  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b2);
  }
}
```

The precondition of sum holds, so everything is ok

Account ADT (Java + Verifast)

```
public class Account {

  static int sum(AccountS a1, AccountS a2)
  //@ requires a1.Account(?b1) &*& a2.Account(?b2);
  //@ ensures a1.Account(b1) &*& a2.Account(b2);;
  {
    return a1.getBalance()+a2.getBalance();
  }

  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b1);
  }
}
```

The precondition does not hold now, but “intuitively” the contract is OK. It works even if a1 and a2 are aliases !

Part II

Fractional Permissions

Representing Sharing in SL

- In “Classical” Logic, we have the law

$$A \Leftrightarrow A \wedge A$$

  This is useful to reason about “pure” values

- In Separation logic, we do not have in general

$$A \not\Leftrightarrow A * A$$

We have seen that the separation principle is key to control aliasing, and support frame (local) reasoning

Representing Sharing in SL

- In “Classical” Logic, we have the law

$$A \Leftrightarrow A \wedge A$$

  This is useful to reason about “pure” values

- In Separation logic, we do not have in general

$$A \not\Leftrightarrow A * A$$

  We have seen that this is useful to control aliasing

- In Separation logic, we would like to “sometimes” allow

$$A \Leftrightarrow A * A$$

  (E.g., if the various usages of A do not actually interfere)

Fractional Permissions [Boyland]

- Answer: we may try to extract “partial views” of the whole permission, using “fractional permissions”, as follows:

$$[f]A \Leftrightarrow [f_1]A * [f_2]A \quad \text{where} \quad 0 \leq f \leq 1 \wedge f = f_1 + f_2$$

- fractional permissions are indexed by rational numbers f such that $0 \leq f \leq 1$
- A permission [1]A means the “whole of A”, so [1]A ⇔ A
- Writing requires full (i.e., 1) permission, unlike reading.
- We now show how Separation Logic rules take fractional permissions into account.

Assignment and Lookup Rules (SL)

- The assignment rule in separation logic is

$$\{\ [1]x \mapsto V\ \}\ x := E\ \{\ [1]x \mapsto E\ \}$$

This means the basic rule:

$$\{\ x \mapsto V\ \}\ x := E\ \{\ x \mapsto E\ \}$$

- The memory lookup rule in SL is now

$$\{\ [f]L \mapsto V\ \}\ y := L\ \{\ [f]L \mapsto V \wedge y = V\}$$

Here, we can allow f<1. But always f>0 !
The frame rule still works !

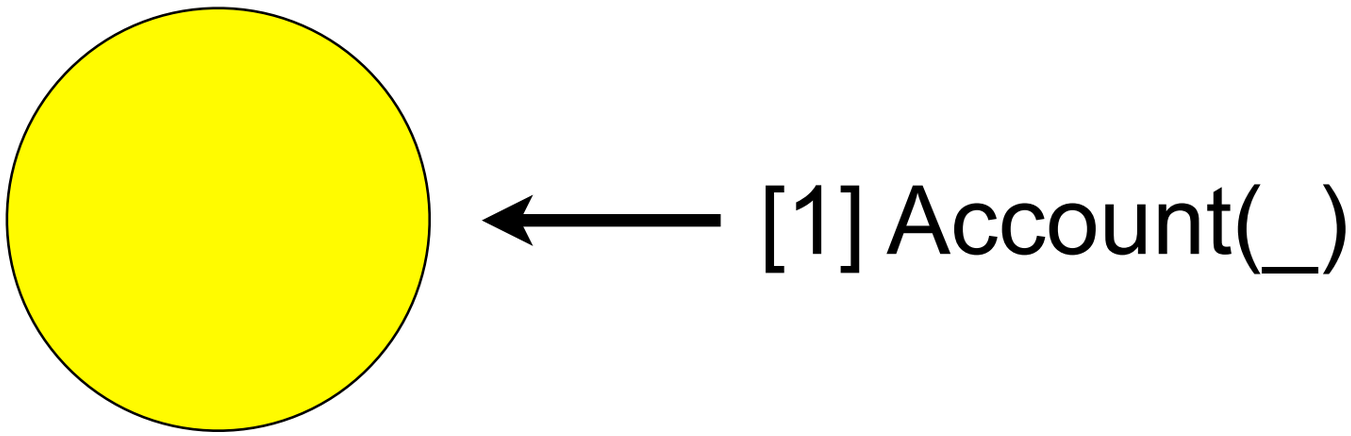
Fractional Permissions

- The key principle to keep in mind is

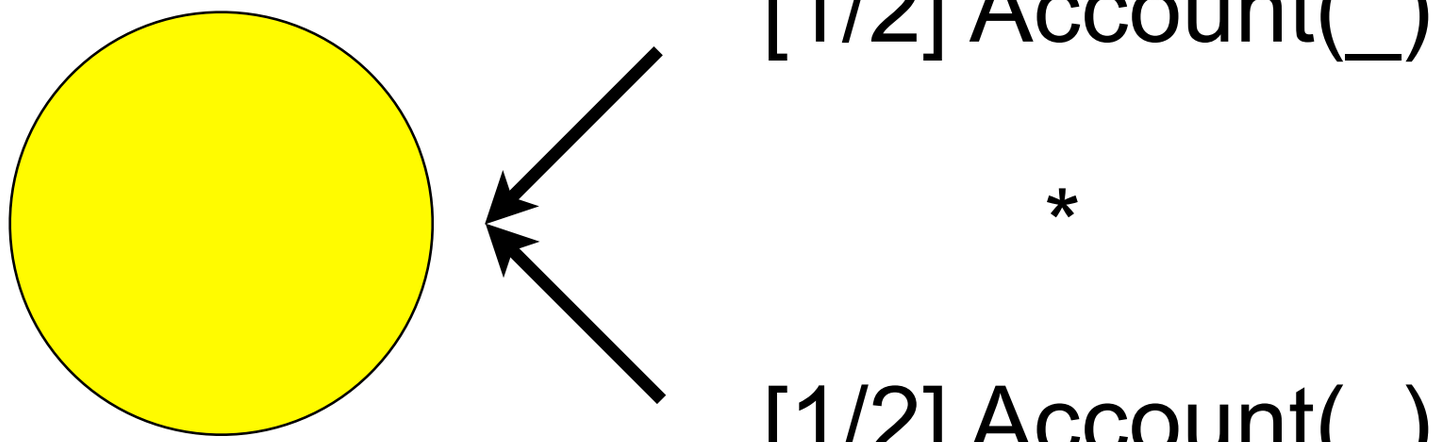
$$[f]A \Leftrightarrow [f_1]A * [f_2]A \quad \text{where} \quad 0 \leq f \leq 1 \wedge f = f_1 + f_2$$

- Using this splitting rule, one can “duplicate” references and heap chunks, allowing sharing or aliasing safely.
- We may recombine fractions later on, eventually regaining the full permission (uniqueness) again.
- Mutable objects may also be safely shared using fractional permissions, if protected by invariants.
- This is the case for concurrent abstract data types, where shared state use is coordinated by locks

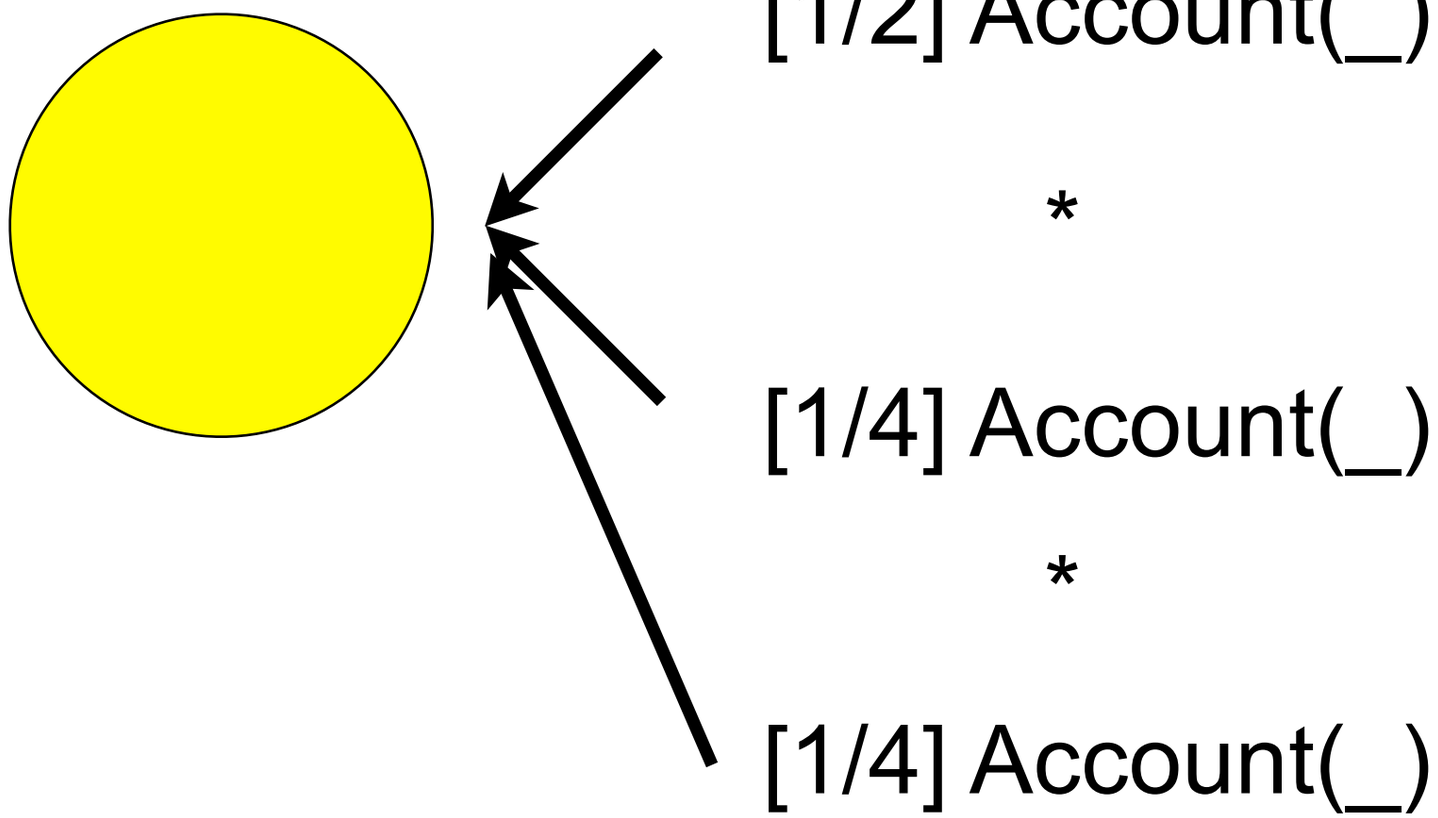
Fractional Permissions



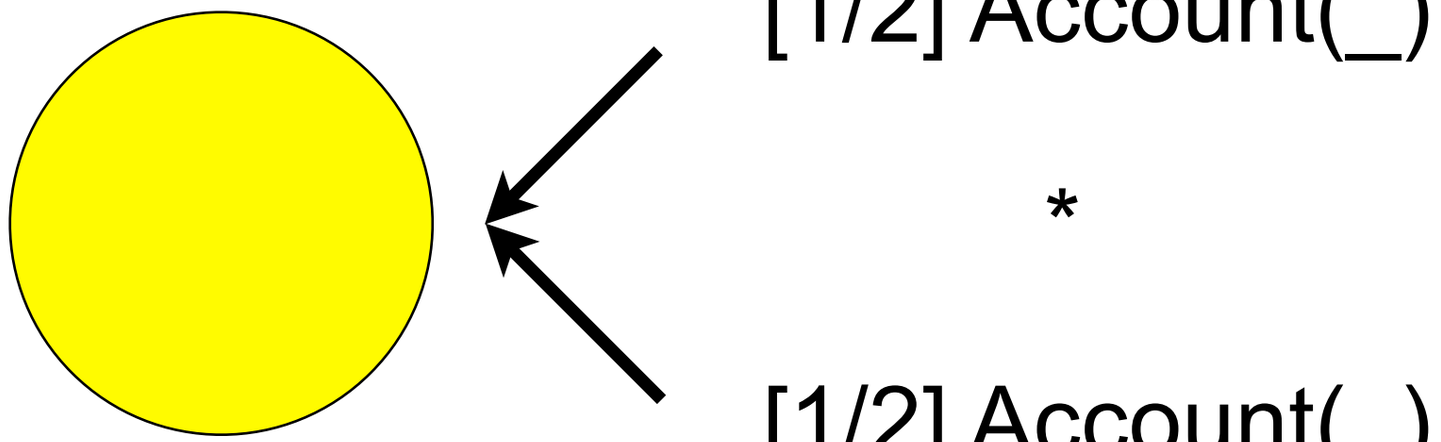
Fractional Permissions



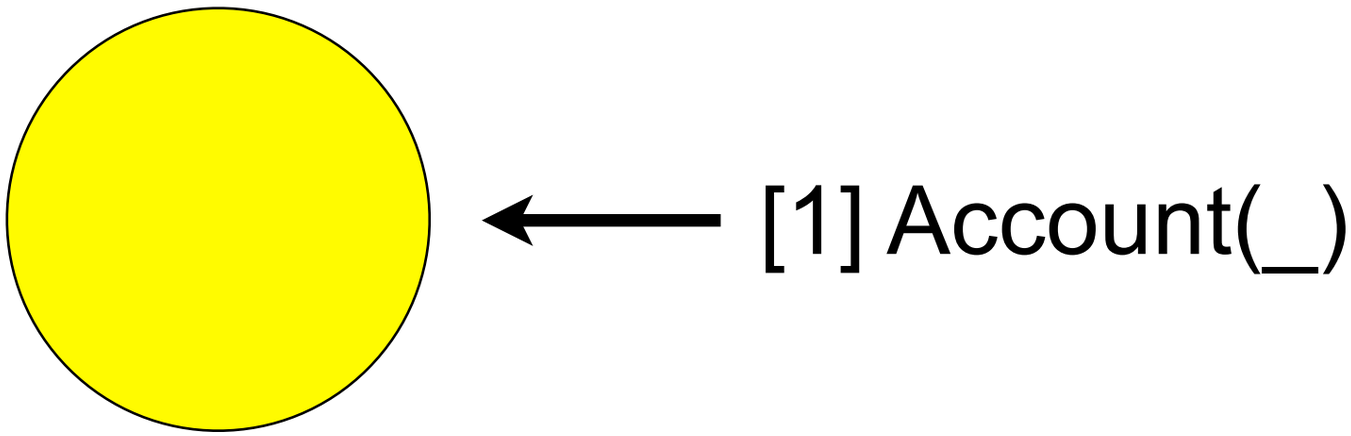
Fractional Permissions



Fractional Permissions



Fractional Permissions



Account ADT (Java + Verifast)

```
public class Account {

  static int sum(AccountS a1, AccountS a2)
  //@ requires a1.Account(?b1) &*& a2.Account(?b2);
  //@ ensures a1.Account(b1) &*& a2.Account(b2);;
  {
    return a1.getBalance()+a2.getBalance();
  }

  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b1);
  }
}
```

- The precondition does not hold now, but “intuitively” the contract is OK. It works even if a1 and a2 are aliases !

Account ADT (Java + Verifast)

```
public class Account {

  static int sum(AccountS a1, AccountS a2)
  //@ requires [?f1]a1.Account(?b1) &*& [?f2]a2.Account(?b2);
  //@ ensures [f1]a1.Account(b1) &*& [f2]a2.Account(b2);;
  {
    return a1.getBalance()+a2.getBalance();
  }

  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b1);
  }
}
```

- The precondition of the sum call holds (a1 and a2 are aliases). This is fine because getBalance only reads!

Account ADT (Java + Verifast)

```
public class Account {

  int balance;


  ...
  int getBalance()
  //@ requires [?f]AccountInv(?b);
  //@ ensures [f]AccountInv(b) &*& result==b &*& b >= 0;
  {
    return balance;
  }

  ...

}
```

- We make that explicit in getBalance contract, using a fractional permission for the invariant AccountInv(_)

Account ADT (Java + Verifast)

```
public class Account {

  int balance;


  ...
  int getBalance()
  //@ requires AccountInv(?b);
  //@ ensures AccountInv(b) &*& result==b &*& b >= 0;
  {
    balance = balance - 1;
    balance = balance + 1;
    return balance;
  }
  ...
}
```

- Note that the code above proof checks.

Account ADT (Java + Verifast)

```
public class Account {

  int balance;


  ...
  int getBalance()
  //@ requires [?f]AccountInv(?b);
  //@ ensures [f]AccountInv(b) &*& result==b &*& b >= 0;
  {
    balance = balance - 1;
    balance = balance + 1;
    return balance;
  }
  ...
}
```

- Note: the code above does not proof check, and in fact it is not correct (e.g, a “data race” may occur).

Part III

Fractional Permissions and Locks

Java Monitors Revisited

```
package java.util.concurrent.locks;

/*@
predicate lck(ReentrantLock s; int p, predicate() inv);

predicate cond(Condition c; predicate() inv, predicate() p);

predicate enter_lck(int p, predicate() inv) = (p == 0 ? emp : inv()) ;

predicate set_cond(predicate() inv, predicate() p) = true;

@*/
```

enter_lock: to associate Representation Invariant to monitor

set_cond: to associate logical assertion to Condition object

Java Monitors Revisited

```
public class ReentrantLock {

    public ReentrantLock();
    //@ requires enter_lck(1,?inv);
    //@ ensures lck(this, 1, inv);

    public void lock();
    //@ requires [?f]lck(?t, 1, ?inv);
    //@ ensures [f]lck(t, 0, inv) &*& inv();

    public void unlock();
    //@ requires [?f]lck(?t, 0, ?inv) &*& inv();
    //@ ensures [f]lck(t, 1, inv);

    public Condition newCondition();
    //@ requires lck(?t, 1, ?inv) &*& set_cond(inv, ?pred);
    //@ ensures lck(t, 1, inv) &*& result != null &*& cond(result,inv,pred);
}
```

the lock and unlock operations are to be used in the “concurrent” context (`[?f]lck…`), and give exclusive access to the ADT — `[1]inv()`

Java Monitors Revisited

```
package java.util.concurrent.locks;

public interface Condition {

    public void await();
        //@ requires cond(this,?inv,?acond) &*& inv();
        //@ ensures cond(this,inv, acond) &*& acond();

    public void signal();
        //@ requires cond(this,?inv,?acond) &*& acond();
        //@ ensures cond(this,inv,acond) &*& inv();

}
```

Concurrent Counter ADT Revisited

```
/*@

predicate_ctor BCounter_shared_state (BCounter c) () =
   c.N |-> ?v &*& v >= 0 &*& c.MAX |-> ?m &*& m > 0 &*& v <= m;

predicate_ctor BCounter_nonzero (BCounter c) () =
   c.N |-> ?v &*& c.MAX |-> ?m &*& v > 0 &*& m > 0 &*& v <= m;

predicate_ctor BCounter_nonmax (BCounter c) () =
   c.N |-> ?v &*& c.MAX |-> ?m &*& v < m &*& m > 0 &*& v >= 0;
```

```
predicate BCounterInv(BCounter c;) =
        c.mon |-> ?l
    &*& l != null
    &*& lck(l,1, BCounter_shared_state(c))
    &*& c.notzero |-> ?cc
    &*& cc !=null
    &*& cond(cc, BCounter_shared_state(c), BCounter_nonzero(c))
    &*& c.notmax |-> ?cm
    &*& cm !=null
    &*& cond(cm, BCounter_shared_state(c), BCounter_nonmax(c));
@*/
```

Concurrent Counter ADT Revisited

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  BCounter(int max)
  //@ requires 0 < max;
  //@ ensures BCounterInv(this);
  {
      N = 0 ;
      MAX = max;
      //@ close BCounter_shared_state(this)();
      //@ close enter_lck(1,BCounter_shared_state(this));
      mon = new ReentrantLock();
      //@ close set_cond(BCounter_shared_state(this),BCounter_nonzero(this));
      notzero = mon.newCondition();
      //@ close set_cond(BCounter_shared_state(this),BCounter_nonmax(this));
      notmax = mon.newCondition();
      //@ close BCounterInv(this);
  }
  ...
}
```

Concurrent Counter ADT Revisited

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void inc()
  //@ requires [?f]BCounterInv(this);
  //@ ensures [f]BCounterInv(this);
  {
      //@ open [f]BCounterInv(this);
      mon.lock();
      //@ open BCounter_shared_state(this)();
      if( N == MAX ) {
        //@ close BCounter_shared_state(this)();
        notmax.await();
        //@ open BCounter_nonmax(this)();
      }
      N++;
      //@ close BCounter_nonzero(this)();
      notzero.signal();
      mon.unlock();
      //@ close [f]BCounterInv(this);
  }
  ...
}
```

Concurrent Counter ADT Revisited

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void dec()
  //@ requires [?f]BCounterInv(this);
  //@ ensures [f]BCounterInv(this);
  {
    //@ open [f]BCounterInv(this);
    mon.lock();
    //@ open BCounter_shared_state(this)();
    if ( N == 0 )
    {
      //@ close BCounter_shared_state(this)();
      notzero.await();
      //@ open BCounter_nonzero(this)();
    }
    N--;
    //@ close BCounter_nonmax(this)();
    notmax.signal();
    mon.unlock(); // release ownership of the shared state
    //@ close [f]BCounterInv(this);
  }
```

Counter ADT (Java + Verifast)

```
public class Main {

    public void doMain ()
    //@ requires true;
    //@ ensures true;
    {
      BCounter ccount = new BCounter();
      //@ assert BCounterInv(ccount);
      Inc_thread it = new Inc_thread(ccount);
      (new Thread(it)).start();
      Dec_thread dt = new Dec_thread(ccount);
      (new Thread(dt)).start();
    }
}
```

Counter ADT (Java + Verifast)

```
public class Main {

    public void doMain ()
    //@ requires true;
    //@ ensures true;
    {
      BCounter ccount = new BCounter();
      //@ assert BCounterInv(ccount);
      Inc_thread it = new Inc_thread(ccount);
      //@ assert [?f]BCounterInv(ccount) &*& f < 1.0;
      (new Thread(it)).start();
      Dec_thread dt = new Dec_thread(ccount);
      new Thread(dt).start();
    }
}
```

Verifast Interface for Threads

```
public interface Runnable {
    //@ predicate pre();                    -- to be redefined in sub-class
    //@ predicate post();                   -- to be redefined in sub-class
    public void run();
        //@ requires pre();
        //@ ensures post();
}
public class Thread {
    static final int MAX_PRIORITY = 10;
    //@ predicate Thread(Runnable r, boolean started);
    public Thread(Runnable r);
        //@ requires true;
        //@ ensures Thread(r, false);
    void start();
        //@ requires Thread(?r, false) &*& r.pre();
        //@ ensures Thread(r, true);
    void setPriority(int newPriority);
        //@ requires Thread(?r, false);
        //@ ensures Thread(r, false);
}
```

Counter ADT (Java + Verifast)

```
class Inc_thread implements Runnable {
  public BCounter loc_cc;
  //@ predicate pre() = Inc_threadInv(this);
  //@ predicate post() = true;

  public Inc_thread(BCounter cc)
  //@ requires cc != null &*& [1/2]CCounterInv(cc);
  //@ ensures Inc_threadInv(this);
  {
      loc_cc = cc;
  }
  public void run()
        //@ requires pre();
        //@ ensures post();
  {
      while(true)
      //@ invariant Inc_threadInv(this);
          { loc_cc.inc();}
  }
}
```

Exercise

- Verify a client function for a shared Counter that launches 100 threads with a maximum value of 10.

```
int MAX = 30;
BCounter c = new BCounter(MAX);

for(int i = 0; i < 100; i++)
{
    new Thread(new Inc(c)).start();
    new Thread(new Dec(c)).start();
}
```

Part IV

More Concurrency

More Concurrency

- The basic monitor scheme does not allow more than 1 thread to ever touch the shared state
- However, it should be ok to let more than 1 thread access the shared state simultaneously, if they do not interfere in “unsafe” ways (e.g., they read only)
- This amounts to controlling sharing granularity, and there are many, more refined, ways to do this.
- A simpler to use pattern is the N readers & 1 writer monitor idiom, which is already very useful
- The key idea is to protect the concurrent object with a more refined monitor wrapper, that coordinates read / write access to the shared state.

N Readers & 1 Writer (pseudo code)

```
class NR1W {
  int readercount; // number of readers inside the CADT
  Bool busy;       // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  RW() { readercount = 0; busy = false; }
  void  StartRead ()
  {
      mon.enter();
      if (busy) await(OKtoRead);
      readercount = readercount + 1;
      signal(OktoRead);
      mon.leave();
  }
  void EndRead ()
  {
      mon.enter();
      readercount = readercount - 1;
      if (readercount == 0) signal(OKtoWrite);
      mon.leave();
  }
  …
}
```

N Readers & 1 Writer

```
class NR1W {
  int readercount;
  Bool busy;
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  void  StartWrite ()
  {
    mon.enter();
    if ((readercount > 0) || busy) await(OKtoWrite);
    busy = true;
    mon.leave();
  }
  void EndWrite ()
  {
    mon.enter();
    busy = false;
    signal(OKtoRead);
    mon.leave();
  }
}
```

Note: StartRead() / EndRead()
and
StartWrite() / EndWrite()
will “guard” access to the shared data!

N Readers & 1 Writer

```
class NR1W {
  int readercount;
  Bool busy;
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && ¬ busy }
  // { OktoWrite = INV && ¬ busy && (readercount == 0) }

  NRW1() {
      readercount = 0;
      busy = false;
      // { INV }
  }
}
```

N Readers & 1 Writer

```
class NR1W {
  int readercount; Bool busy;
  ReentrantLock mon;
  Condition OKtoread, OKtowrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && ¬ busy }
  // { OktoWrite = INV && ¬ busy && (readercount == 0) }

  void  StartRead () // POST: (readercount > 0)
  { mon.enter();
    { INV }
    if (busy) { INV & busy } await(OKtoRead);
    { INV && ¬ busy }
    signal(OktoRead);
    { INV }
    readercount = readercount + 1;
    { INV && (readercount > 0) }
    mon.leave();
  }
}
```

N Readers & 1 Writer

```
class NR1W {
  int readercount; Bool busy;
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && ¬ busy }
  // { OktoWrite = INV && ¬ busy && (readercount == 0) }

  void EndRead ()
  // {PRE: readercount > 0 }
  { mon.enter();
    { INV && (readercount > 0) }
    { INV && (readercount > 0) && ¬ busy }
    readercount = readercount - 1;
    { INV && (readercount >= 0) && ¬ busy }
    if (readercount == 0)
      { INV && (readercount == 0) && ¬ busy }
      signal(OKtoWrite)
      { INV }
    { INV }
    mon.leave();
  }
}
```

N Readers & 1 Writer

```
class NR1W {
  int readercount; Bool busy;
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && ¬ busy }
  // { OktoWrite = INV && ¬ busy && (readercount == 0) }

  void  StartWrite ()
  // { POST: (readercount = 0 && busy) }
  {
    mon.enter();
    { INV }
    if ((readercount != 0) || busy) {
      { INV & (readercount != 0 || busy) }
      await(OKtoWrite);
      { INV && ¬ busy && (readercount == 0) }
    }
    { INV && ¬ busy && (readercount == 0) }
    busy = true
    { INV && busy && (readercount == 0) }
    mon.leave();
  }
```

N Readers & 1 Writer

```
class NR1W {
  int readercount; Bool busy;
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && ¬ busy }
  // { OktoWrite = INV && ¬ busy && (readercount == 0) }
  void { PRE: busy } EndWrite ()
  {
    mon.enter();
    { INV }
    busy = false;
    { INV && ¬ busy }
    signal(OKtoRead);
    { INV }
    mon.leave();
  }
}
```

(Example) Concurrent Dictionary

```
class ConcurrentDictionary {
  int readercount;
  Bool busy;
  Dict myDict;

  void StartRead () {POST: (readercount > 0) && ¬ busy}
  void {PRE: (readercount > 0) } EndRead ()
  void StartWrite () {POST: busy && (readercount == 0) }
  void {PRE: busy } EndWrite () {POST: ¬ busy && (readercount = 0) }

  Val Find(Key k) {
    StartRead(); { (readercount > 0) && ¬ busy } /* retrieve value from D */;
    v = myDict.find(k);
    EndRead();
    return v;
  }
  void Insert(Key k, Val v) {
    StartWrite(); { busy && (readercount == 0) } /* insert value in D */ ;
    myDict.insert(k,v);
    EndWrite();
  }
}
```

(Example) Concurrent Dictionary

```
class ConcurrentDictionary {
  int readercount;
  Bool busy;
  Dict myDict;

  void StartRead () {POST: (readercount > 0) && ¬ busy}
  void {PRE: (readercount > 0) } EndRead ()
  void StartWrite () {POST: busy && (readercount == 0) }
  void {PRE: busy } EndWrite () {POST: ¬ busy && (readercount = 0) }

  Val Find(Key k) {
    StartRead(); { (readercount > 0) && ¬ busy } /* retrieve value from D */;
    v = myDict.find(k);
    EndRead();
    return v;
  }
  void Insert(Key k, Val v) {
    StartWrite(); { busy && (readercount == 0) } /* insert value in D */ ;
    myDict.insert(k,v);
    EndWrite();
  }
}
```

StartRead() / EndRead()
and
StartWrite() / EndWrite()
“guard” access to the shared data!

(Example) Concurrent Dictionary

```
class ConcurrentDictionary {
  int readercount;
  Bool busy;
  Dict myDict;

  void StartRead () {POST: (readercount > 0) && ¬ busy}
  void {PRE: (readercount > 0) } EndRead ()
  void StartWrite () {POST: busy && (readercount == 0) }
  void {PRE: busy } EndWrite () {POST: ¬ busy && (readercount = 0) }

  Val Find(Key k) {
    StartRead(); { (readercount > 0) && ¬ busy } /* retrieve value from D */;
    v = myDict.find(k);
    EndRead();
    return v;
  }
  void Insert(Key k, Val v) {
    StartWrite(); { busy && (readercount == 0) } /* insert value in D */ ;
    myDict.insert(k,v);
    EndWrite();
  }
}
```

StartRead() / EndRead()
and
StartWrite() / EndWrite()
“guard” access to the shared data!

More Concurrency

- We may consider even more refined patterns. The general reasoning principles still apply:
- Unique ownership of shared state
- Shared state may be split in different disjoint chunks:

  Each “piece” of shared state must be protected by a monitor. You must statically know what piece of shared state (and invariant) each monitor is protecting.

  To access a piece of shared state, a thread must call mon.enter() (a.k.a. mon.lock)

  Example: an open hash table with a different monitor protecting each collision list.

More Concurrency

- We may consider even more refined patterns. The general reasoning principles still apply:
- Replace pre-conditions by monitor conditions inside critical regions

  A thread cannot check the shared state safely if it does not own the shared state (which can only happen after mon.enter)

  If a pre-condition C does not hold either await for C, or release the shared state and try later (this should be exceptional behaviour)

More Concurrency

- We may consider even more refined patterns. The general reasoning principles still apply:
- multiple ownership of shared state

  In some cases, you may allow several threads to own a piece of shared state simultaneously

  To tackle that, use the N-readers + 1 writer pattern

  This works whenever the contents of the shared state is concurrently used safely, not just when the shared state is only read (this is just a special case)

  Example: shared state contains “atomic” variables, or concurrent ADTs, and invariants are not broken.

More Concurrency

- We may consider even more refined patterns (for the daring). The general reasoning principles still apply:
- Synchronisation free programming

  There are several other interesting ways of writing concurrent programs, introduced more recently
- Transactional Memory

  Use higher level concept of transaction, to ensure ACID properties.
- Lock-free (racy) programming

  Writing this kind of code correctly is very challenging, and only used in high performance or low level code.

  This is an active research area.

More Concurrency (Revisited)

Transactional Memory

Essentially, one reasons about a transaction in the same way as a we would for permission based access to shared state (as if bounded by enter/leave)

To each transaction, we associate a resource bundle rb describing the shared state and its invariants

$$\frac{\{A * rb.INV\}\ \ S\ \ \{B * rb.INV\}}{\{A\}\ \text{atomic}\ S\ \{B\}}$$

We may reason about S as if no interference occurs (isolation/atomicity), but invariants must be preserved across borders. Caveat: there are no ways to check preconditions (cf. wait / signal).

Summary

- Exclusive access to monitor is essential to preserve consistency of an ADT
- More flexible mechanisms can be used in the case of safe interference and aliasing.
- Safe sharing (for reading) can be checked with fractional permissions and disciplined with monitors anyway.
- Monitors implementing N-readers and 1-Writer can be used to extend sequential ADTs to concurrent ones.
- Other, flexible, concurrency control mechanisms are possible, but more challenging. Verification of such mechanisms is an open research area.