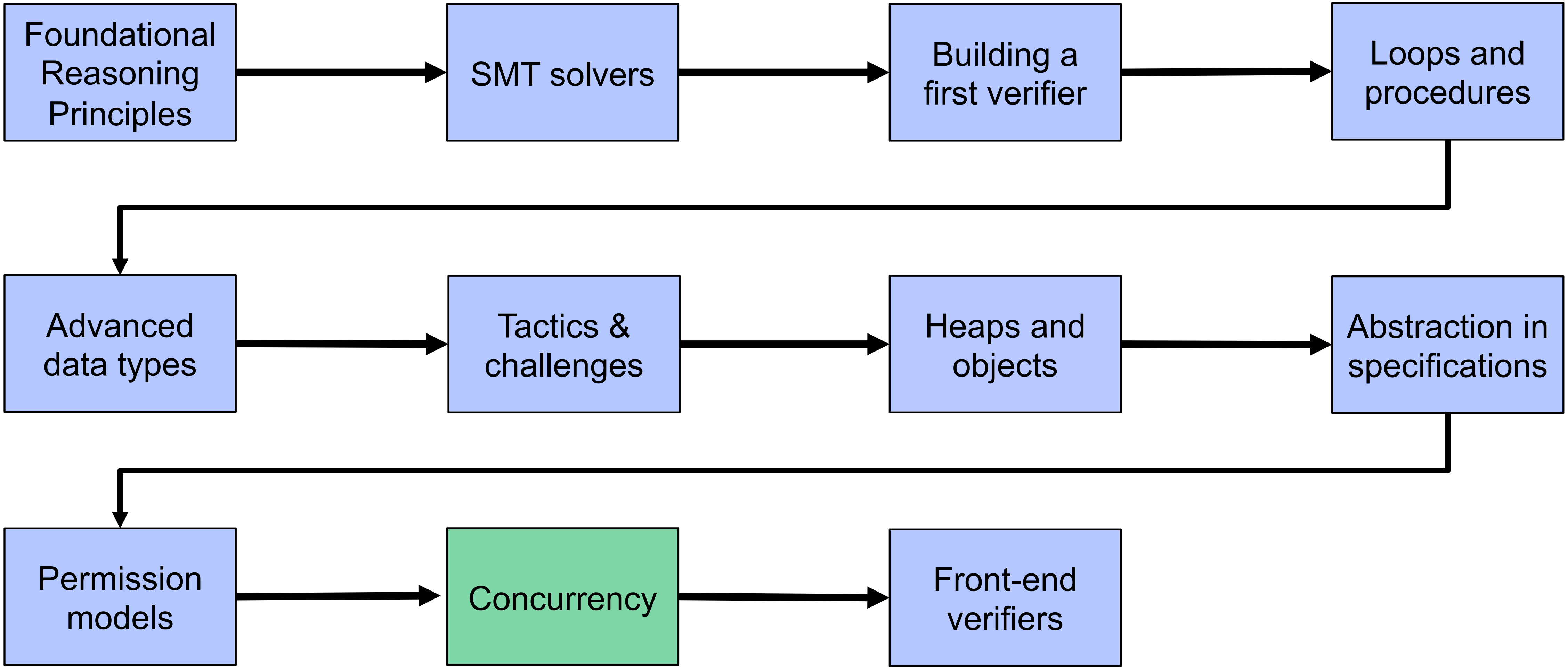
02245 – Module 10

CONCURRENCY

Tentative course outline



But first: the permission story

- Who may hold permissions and how are they transferred?

But first: the permission story

- Who may hold permissions and how are they transferred?

```
method foo(this: Ref)
  requires acc(this.d)
  ensures acc(this.d)
{
   this.d := 17
}
```

method executions

```
while (0 < i)
  invariant acc(this.d)
{
  this.d := this.d + i
  i := i - 1
}
```

loop iterations

```
predicate list(this: Ref)
{ acc(this.next) && ... }

var x: Ref
x := new(*)
fold(list)
```

predicate instances

```
// gain permission
inhale acc(...)
```

```
// give up permission
exhale acc(...)
```

```
// trade permission
unfold list(x)
fold list(x)
```

➔ Next: method executions may also *run in parallel*

Reasoning about concurrent programs – challenges

```
x.f := x.f + 1   ||   x.f := x.f + 1
```

Data race: 2+ threads access same data, at least one mutates data

```
acquire x            ||   acquire x
x.f := 5             ||   x.f := 0
release x            ||   release x
acquire x
y := 10 / x.f
release x
```

Reasoning about thread interference

```
acquire x      ||   acquire y
acquire y      ||   acquire x
…              ||   …
release x      ||   release x
release y      ||   release y
```

Deadlock

```
x.f := 0
acquire x            ||   acquire x
x.f := x.f + 1       ||   x.f := x.f + 1
release x            ||   release x
acquire x
assert x.f == 2
```

Reasoning about thread cooperation

Thread-modular verification

- All verification techniques introduced so far are procedure-modular
  - Reason about calls in terms of the callee's specification
  - Verification of a method does not consider callers or implementation of callees

```
method create() returns (res: Ref)
  ensures  list(res)
  ensures  content(res) == Seq[Int]()
{
  res := new(*)
  res.next := null
  fold list(res)
}
```

- We will now present techniques that are also thread-modular
  - Reason about a thread execution without knowing which other threads might run concurrently

```
acquire x              ||    acquire x
x.f := 5               ||    x.f := 0
release x              ||    release x
acquire x              ||
y := 10 / x.f          ||
release x              ||
```

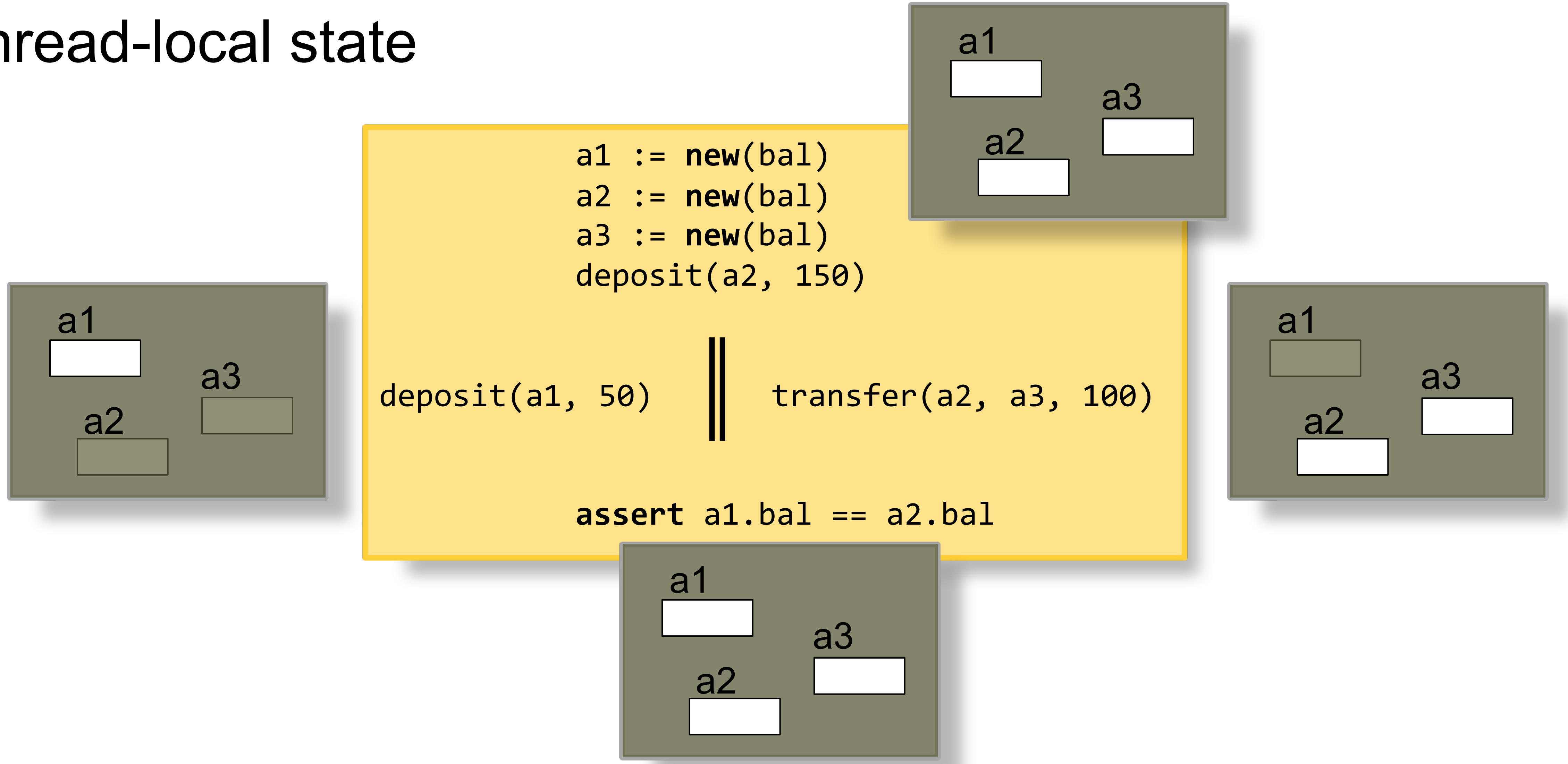
- Both forms of modularity are crucial for verification to scale

Concurrency

1. Concurrency with thread-local state

2. Shared state and synchronization

Thread-local state



Thread-local state: parallel branches operate on disjoint memory

➔ data races are not possible

Structured parallelism

- Permissions and separating conjunction lead to a simple proof rule

```
{ P1 } S1 { Q1 }      { P2 } S2 { Q2 }
-----------------------------------------
  { P1 * P2 } S1 || S2 { Q1 * Q2 }
```

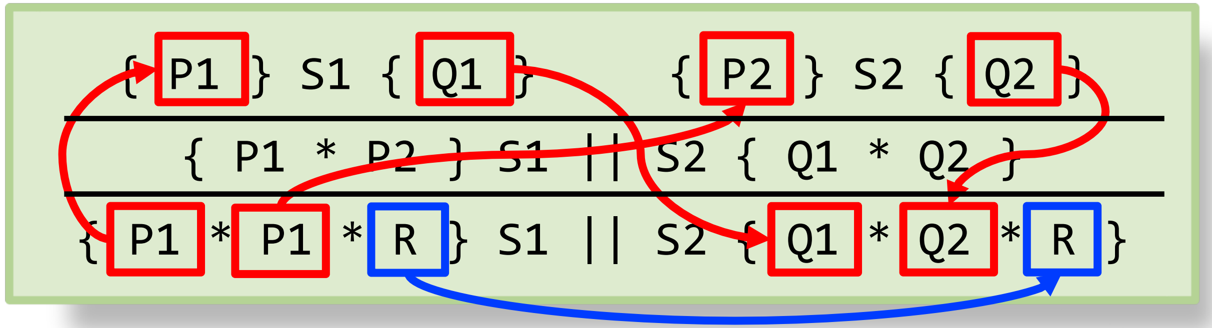
- All shared memory is on the heap
- Separating conjunction prevents interference between the parallel branches

- Programs with data races have an unsatisfiable precondition

```
{ acc(x.f) } x.f := 7 { ... }      { acc(x.f) } y := x.f { ... }
-------------------------------------------------------------------
      { acc(x.f) * acc(x.f) } x.f := 7 || y := x.f { ... }
```

Encoding structured parallelism

- The proof rule employs the familiar permission transfer



- We can encode this proof rule via exhale and inhale operations

```
method left(…) returns (res1: T)
  requires P1
  ensures  Q1
{ // encoding of S1 }
```

Encode left and right branch
as methods with specifications

```
exhale P1[…]
exhale P2[…]
havoc res1, res2
inhale Q1[…]
inhale Q2[…]
```

Encode parallel composition
like two “half method calls”

Example: parallel list search

➔ `00-busy.vpr`

```
method busy(courses: Ref, seminars: Ref, exams: Ref, today: Int) returns (res: Bool)
  requires list(courses) && list(seminars) && list(exams)
  ensures  list(courses) && list(seminars) && list(exams)
  ensures  res == (today in content(courses) ||
                   today in content(seminars) ||
                   today in content(exams))
{
                                            || var rightRes: Bool
                                            || rightRes := contains(seminars, today)
  var leftRes: Bool                         || var res2: Bool
  leftRes := contains(courses, today)       || res2 := contains(exams, today)
                                            || rightRes := rightRes || res2

                    res := leftRes || rightRes
}
```

What have we proved when the Viper encoding verifies?

Example: parallel read access

➔ `01-getStressed.vpr`

- Since `contains` is side-effect free, concurrent executions should be allowed

```
method getStressed(exams: Ref, today: Int) returns (res: Bool)
  requires list(exams)
  ensures  list(exams)
  ensures  res == (today in content(exams) || today + 1 in content(exams))
{

  var leftRes: Bool                        ||  var rightRes: Bool
  leftRes := contains(exams, today)        ||  rightRes := contains(exams, today + 1)

  res := leftRes || rightRes
}
```

- Fractional permissions enable concurrent read access
- but prevent concurrent reads and writes (and, thus, data races)

Parallel branches with side-effects

➔ `02-par-swap.vpr`

```
method client(cell1: Ref, cell2: Ref, cell3: Ref, cell4: Ref)
  requires acc(cell1.f) && acc(cell2.f) && acc(cell3.f) && acc(cell4.f)
{
  cell1.f := 1
  cell2.f := 2

  swap(cell1, cell2)        ||      swap(cell3, cell4)

  assert cell1.f == 2
}
```

- In the encoding presented so far, old-expressions in the postconditions of the left and right branch are interpreted incorrectly
- They should refer to the heap before the parallel composition (not the prestate of the enclosing method, which is unsound)

Labeled old-expressions

➔ `03-par-swap.vpr`

- Viper allows the declaration of labels (at positions where statements may occur)
- Labeled old-expressions are evaluated in the heap at the label
- Encoding of parallel composition uses label to interpret the postconditions of the two branches

```
label branch

// exhale precondition of left block
// exhale precondition of right block

// postcondition of left block
inhale acc(cell1.f) && acc(cell2.f)
inhale cell1.f == old[branch](cell2.f) && cell2.f == old[branch](cell1.f)
// analogous for postcondition of right block
```

Exercise: structured parallelism

➔ `04-array-inc-all.vpr`

a. Implement and encode the method below; it increments all elements of an array
b. Verify memory safety
c. Specify and verify functional correctness

```
method incrementAll(a: Array)
  requires …
  ensures  …
{
  …

  // sequential increment of       ||   // sequential increment of
  // left half of the array        ||   // right half of the array

  …
}
```

Solution: structured parallelism

```
method left(a: Array, mid: Int)
  requires arraySeg(a, 0, mid)
  requires 0 <= mid && mid <= len(a)
  ensures  arraySeg(a, 0, mid)
  ensures  forall j: Int :: 0 <= j && j < mid ==> lookup(a, j) == old(lookup(a, j)) + 1
{
  var i: Int := 0
  while(i < mid)
    invariant arraySeg(a, 0, mid)
    invariant 0 <= i && i <= mid
    invariant forall j: Int :: 0 <= j && j < i ==> lookup(a, j) == old(lookup(a, j)) + 1
    invariant forall j: Int :: i <= j && j < mid ==> lookup(a, j) == old(lookup(a, j))
  {
    update(a, i, lookup(a, i)+1)
    i := i + 1
  }
}
```

- Analogous for right branch

Solution: structured parallelism (cont'd)

```
method incrementAll(a: Array)
  requires array(a)
  ensures  array(a)
  ensures  forall j: Int :: 0 <= j && j < len(a) ==> lookup(a, j) == old(lookup(a, j)) + 1
{
  var mid: Int := len(a) / 2
  label branch
  // precondition of left block
  exhale arraySeg(a, 0, mid) && mid <= len(a)
  // precondition of right block
  exhale arraySeg(a, mid, len(a)) && 0 <= mid

  // postcondition of left block
  inhale arraySeg(a, 0, mid)
  inhale forall j: Int :: 0 <= j && j < mid ==> lookup(a, j) == old[branch](lookup(a, j)) + 1
  // postcondition of right block
  inhale arraySeg(a, mid, len(a))
  inhale forall j: Int :: mid <= j && j < len(a) ==> lookup(a,j) == old[branch](lookup(a,j))+1
}
```

Parallel for-loops

- Some languages and libraries provide parallel for-loops

```
parallel for i: Int from 0 to len(a)
{ update(a, i, lookup(a, i) + 1) }
```

- We can treat such loops as generalized (unbounded) parallel composition

```
body(0) || body(1) || … || body(len(a) – 1)
```

- For this purpose, we specify the loop body with a pre- and postcondition (instead of a loop invariant)

```
parallel for i: Int from 0 to len(a)
  requires acc(loc(a, i).val)
  ensures  acc(loc(a, i).val)
  ensures  lookup(a, i) == old(lookup(a, i)) + 1
{ update(a, i, lookup(a, i) + 1) }
```

old refers to pre-state of the loop

Encoding of parallel for-loops

➔ `05-par-for-loop.vpr`

```
parallel for i: Int from 0 to len(a)
  requires acc(loc(a, i).val)
  ensures  acc(loc(a, i).val)
  ensures  lookup(a, i) ==
            old(lookup(a, i)) + 1
{ update(a, i, lookup(a, i) + 1) }
```

```
method body(i: Int, a: Array)
  requires 0 <= i && i < len(a)
  requires acc(loc(a, i).val)
  ensures  acc(loc(a, i).val)
  ensures  lookup(a, i) ==
             old(lookup(a, i)) + 1
{ update(a, i, lookup(a, i) + 1) }
```

Check that loop body satisfies its specification

Intuition for encoding of loop

```
body(0) || … || body(len(a)-1)
```

```
exhale pre(0) && … && pre(len(a)-1)
inhale post(0) && … && post(len(a)-1)
```

Encoding of loop

```
label l
exhale forall i: Int :: 0 <= i && i < len(a)
         ==> acc(loc(a, i).val)
inhale forall i: Int :: 0 <= i && i < len(a)
         ==> acc(loc(a, i).val)
inhale forall i: Int :: 0 <= i && i < len(a)
         ==> lookup(a, i) == old[l](lookup(a, i)) + 1
```

Unstructured parallelism (threads)

- Most programming languages offer unstructured parallelism via threads

Statements

```
S ::= …
    | x := fork m(Ē)
    | y := join x
```

- Fork executes a method call in a new thread, returning a thread object
- Join waits for thread to terminate and returns the results of the forked method

- Structured parallelism can easily be encoded via fork and join

```
x := left(…) || y := right(…)
```

```
t1 := fork left(…)
t2 := fork right(…)
x := join t1
y := join t2
```

Challenges of encoding join-operations

- Analogously to structured parallelism, a join inhales the postcondition of the forked method (for instance, to re-gain permissions passed to the forked thread)

- Challenge: how to identify the postcondition to inhale?

```
var t: Thread
if(b) { t := fork left(…)  }
else  { t := fork right(…) }
join t
```

```
method m(t: Thread)
{
  join t
}
```

Examples use a source language, not Viper

- We assume a type system that parameterizes type `Thread` with the method that has been forked

```
var t: Thread<left>
if(b) { t := fork left(…)  }
else  { t := fork right(…) }
join t
```

```
method m(t: Thread<left>)
{
  join t
}
```

Challenges of encoding join-operations (cont'd)

- The postcondition of a forked method will in general refer to method parameters

```
method double(p: Int) returns (res: Int)
  ensures res == p + p
```

- For a join, the corresponding fork is not statically known

```
var t: Thread<double>
if(b) { t := fork double(5) }
else  { t := fork double(7) }
y := join t
assert b ==> y == 10
```

```
method m(t: Thread<double>)
{
  y := join t
  assert y == 10
}
```

- **Problem:** we cannot determine statically how to substitute actual arguments for formal parameters when inhaling the postcondition during a join

Simplified encoding of fork and join

```
method m(p: T<a>) returns (r: T<b>)
  requires P
  ensures  Q
```

- Encoding of fork stores method arguments in fields of the thread object

```
t := fork m(5)
```

```
field pArg: T<a>
```

```
t := new(pArg)
t.pArg := 5
exhale P[p/5]
```

- Encoding of join uses these fields to inhale postcondition

```
y := join t
```

```
inhale Q[p/t.pArg, r/y]
```

Example: parallel list search

➔ 06-fork-busy.vpr

```
method busy(courses: Ref, seminars: Ref, today: Int) returns (res: Bool)
  requires list(courses) && list(seminars)
  ensures  list(courses) && list(seminars)
  ensures  res == (today in content(courses) || today in content(seminars))
{
  var r1: Bool; var r2: Bool
  var t1: Thread<contains>; var t2: Thread<contains>

  t1 := fork contains(courses, today)
  t2 := fork contains(seminars, today)

  r1 := join t1
  r2 := join t2

  res := r1 || r2
}
```

Repeated joins

➔ `07-repeated.vpr`

- Since a join inhales permissions, it is unsound to join the same thread twice

```
join t
join t
assert false
```

```
inhale acc(t.aArg.f)
inhale acc(t.aArg.f)
assert false
```

✔

- To prevent repeated joins of the same thread, the join operation requires and consumes a dedicated join-permission

```
field joinable: Ref
```

```
t := fork m(5)
```

```
t := new(pArg)
t.pArg := 5
exhale A[p/5]
inhale acc(t.joinable)
```

```
y := join t
```

```
exhale acc(t.joinable)
inhale B[p/t.pArg, r/y]
```

Reasoning about heap changes

- Analogously to methods and parallel branches, threads may modify the heap

- Therefore, the postcondition of the forked method may contain old-expressions, which can be encoded via labeled old-expressions

fork

```
t := new(pArg)
t.pArg := x
label l
exhale A[p/5]
inhale acc(t.joinable)
```

```
exhale acc(t.joinable)
inhale … old[l](t.pArg.f) …
```

join

- However, this encoding of join requires that the corresponding fork is statically known and in scope

Reasoning about heap changes (cont'd)

- In general, the corresponding fork for a join is not statically known

```
var t: Thread<double>
if(b) { t := fork double(5) }
else  { t := fork double(7) }
y := join t
```

```
method m(t: Thread<double>)
{
  y := join t
}
```

- In simple cases, we could evaluate old-expressions when a method is forked and store their values in the thread object (like method parameters)

```
method swap(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
  ensures  acc(a.f) && acc(b.f)
  ensures  a.f == old(b.f) &&
           b.f == old(a.f)
```

- This is difficult when old-expressions occur under conditionals, contain result variables, or evaluate to unbounded data structures
- We simply omit such postconditions during a join (sound but incomplete)

Exercise: threads

➔ `08-par-tree-depth.vpr`

a. Encode the method on the right; it computes the height of a binary tree (or -1 if the parameter is null)

b. Verify memory safety

c. Specify and verify functional correctness using the depth function from the template

```
method parDepth(this: Ref) returns (res: Int)
  requires …
  ensures  …
{
  if(this == null) { res := -1 }
  else {
    var r1: Int; var r2: Int
    var t1: Thread<parDepth>
    var t2: Thread<parDepth>

    unfold tree(this)
    t1 := fork parDepth(this.left)
    t2 := fork parDepth(this.right)
    r1 := join t1
    r2 := join t2
    res := max(r1, r2) + 1
    fold tree(this)
  }
}
```

Solution: threads

```
method parDepth(this: Ref) returns (res: Int)
  requires this != null ==> tree(this)
  ensures  this != null ==> tree(this)
  ensures  res == depth(this)
{
  if(this == null) { res := -1 }
  else {
    var r1: Int; var r2: Int
    var t1: Ref; var t2: Ref

    unfold tree(this)

    // t1 := fork parDepth(this.left)
    t1 := new(thisArg)
    t1.thisArg := this.left
    label f1  // not used here
    exhale this.left != null ==>
                tree(this.left)
    inhale acc(t1.joinable)
```

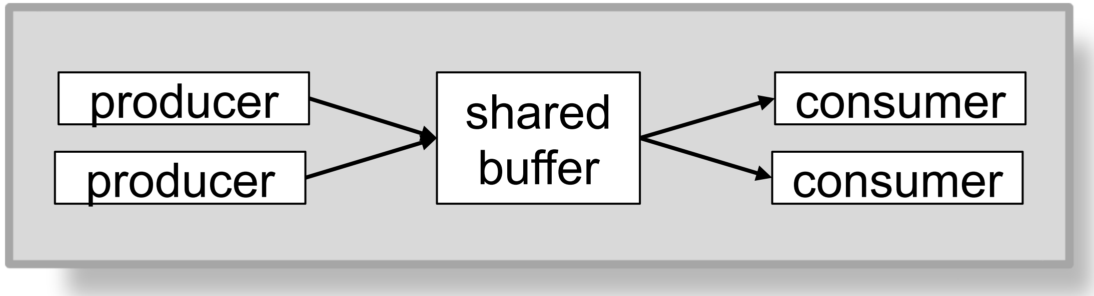
```
    // analogous for:
    // t2 := fork parDepth(this.left)

    // r1 := join t1
    exhale acc(t1.joinable)
    inhale t1.thisArg != null ==>
                tree(t1.thisArg)
    inhale r1 == depth(t1.thisArg)

    // r2 := join t2
    exhale acc(t2.joinable)
    inhale t2.thisArg != null ==>
                tree(t2.thisArg)
    inhale r2 == depth(t2.thisArg)

    res := max(r1, r2) + 1
    fold tree(this)
  }
}
```

Concurrency

1. Concurrency with thread-local state
2. Shared state and synchronization

Shared state

- The solution presented so far supports concurrency with thread-local state
- Threads exchange information upon fork and join, but cannot communicate or collaborate while they are running
- Communication between threads is typically supported by shared state or message passing
- We will focus on shared state, but message passing can also be supported using permissions
- Example: Producer-Consumer



- Concurrent accesses to mutable shared state require synchronization to prevent data races and ensure correctness
- We will focus on locks as a synchronization primitive

Data race freedom

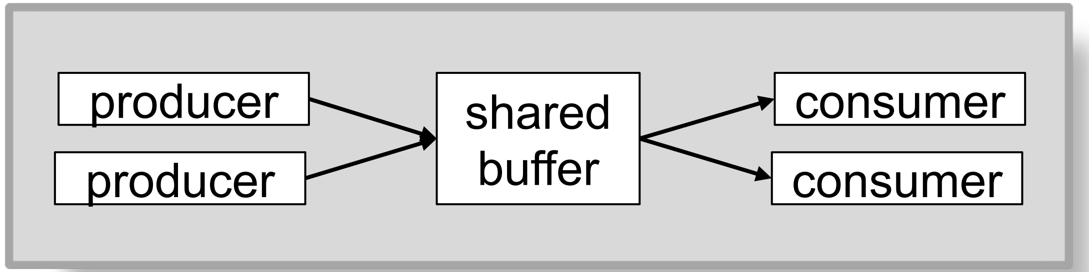
- Concurrent accesses to mutable shared state may lead to data races

```
method produce(buf: Ref)
{
  while(true) {
    if(buf.val == null) {
      buf.val := new()
    }
  }
}
```

```
method consume(buf: Ref)
{
  while(true) {
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
  }
}
```

- In verification, permissions can be used to prove the absence of data races (while permitting concurrent reading)
- In programs, synchronization prevents data races

Synchronization via locks




```
method produce(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val == null) {
      buf.val := new()
    }
    release buf
  }
}
```

```
method consume(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
    release buf
  }
}
```

- Permission to access `buf.val` cannot be obtained via the preconditions (that would prevent concurrent executions)
- Permissions transfer happens when acquiring or releasing a lock

Lock invariants

- A lock guards accesses to certain memory locations

```
class Buffer {
  @GuardedBy("this")
  Product val;
}
```

Java provides annotations to document which locations are guarded by a lock

- We associate each lock with a lock invariant

```
class Buffer {
  lock invariant acc(this.val)
  Product val;
}
```

Permissions in the lock invariant express which locations are guarded by the lock

- **Intuition:** permissions are held by method executions, loop iterations, predicate instances, or locks

Locks and permission transfer

```
class Buffer {
  lock invariant acc(this.val)
  Product val;
}
```


```
method produce(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val == null) {
      buf.val := new()
    }
    release buf
  }
}
```


```
method consume(buf: Ref)
{
  while(true) {
    acquire buf
    if(buf.val != null) {
      // consume buf.val
      buf.val := null
    }
    release buf
  }
}
```

More on lock invariants

- A lock invariant holds whenever the lock is *not* currently held by a thread

- Lock invariants contain arbitrary self-framing assertions

```
acc(this.val) && 0 < this.val
```

```
list(this) && 0 < length(this)
```

```
acc(this.val, 1/2)
```

```
forall x: Ref :: x in s ==> acc(x.val)
```

- Self-framingness is crucial for soundness

```
0 < this.val
```

Methods could violate the invariant without acquiring the lock

Initializing locks

- Before the first acquire, the lock needs to be initialized, to establish the lock invariant and to transfer the permissions to the lock

```
buf := new(val)
acquire buf  // should be rejected
assert false
```

- We introduce a ghost statement `share` that initializes the lock

```
buf := new(val)
share buf
acquire buf  // allowed
```

Simplified encoding of locks

- Locks are encoded as references
- To track whether a lock has been initialized, we use the permission to a ghost field `isLock`

```
share x
```

```
exhale Inv(x)
inhale acc(x.isLock, wildcard)
```

`Inv(x)` denotes the lock invariant

- Some fractional permission for this field is required to acquire the lock
  - Permission is transferred to each thread that accesses the guarded state
- The rule does not prevent sharing a lock twice
  - Multiple inhales of `wildcard` are sound

Simplified encoding of locks (cont'd)

➔ `09-producer-consumer.vpr`

- We model non-reentrant locks (repeated acquire leads to deadlock)
- Therefore, each acquire obtains permissions from the lock

```
acquire x
```

```
assert acc(x.isLock, wildcard)
inhale Inv(x)
```

`wildcard` ensures that the permission to acquire is freely duplicable

```
release x
```

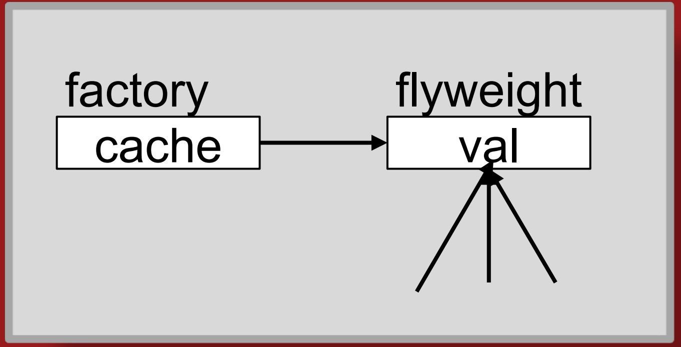
```
exhale Inv(x)
```

- The rule for `acquire` does not prevent deadlock; extra proof obligations can be imposed to ensure that locks are acquired in an order (beyond this course)

Exercise: locking

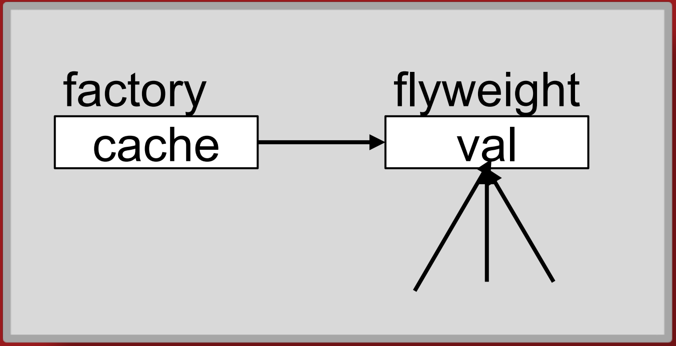
➔ `10-par-flyweight.vpr`

Make our previous implementation of the Flyweight pattern thread-safe, that is, use locks to prevent data races



Solution: locking

Make our previous implementation of the Flyweight pattern thread-safe, that is, use locks to prevent data races




```
// the lock invariant is identical
// to the former factory predicate

define Inv(this) (
  acc(this.cache) &&
  (this.cache != null ==>
        acc(this.cache.val, wildcard))
)
```

```
method get(this: Ref) returns (f: Ref)
  requires acc(this.isLock, wildcard)
  ensures  acc(f.val, wildcard)
{
  acquire(this)
  if(this.cache == null) {
      f := new(val)
      this.cache := f
  }
  f := this.cache
  release(this)
}
```

Client-side vs. server-side locking

Client-side locking

```
method inc(cell: Ref)
  requires acc(cell.val)
  ensures  acc(cell.val)
  ensures  cell.val == old(cell.val) + 1
{
  cell.val := cell.val + 1
}
```

```
acquire cell
cell.val := 0
inc(cell)
assert cell.val == 1
release cell
```

Server-side locking

```
method inc(cell: Ref)
  requires acc(cell.isLock, wildcard)
  ensures  // cannot refer to cell.val
{
  acquire cell
  cell.val := cell.val + 1
  release cell
}
```

```
acquire cell
cell.val := 0
release cell
inc(cell)
acquire cell
assert cell.val == 1
release cell
```

Reasoning about server-side locking

➔ `11-owicki-gries.vpr`

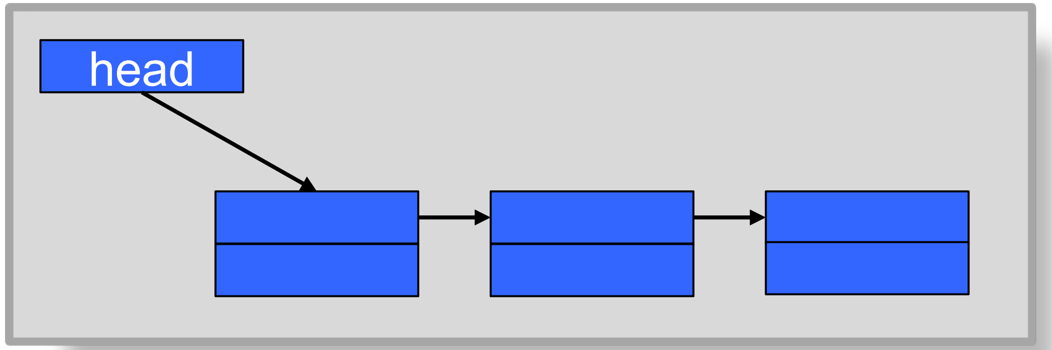
- With server-side locking, methods can typically not provide strong postconditions over the shared data because the permission is not held in the post-state
- In some cases, we can use ghost state to reason about server-side locking
- In general, reasoning about server-side locking requires Owickie-Gries-style rely-guarantee reasoning, which takes into account how all other threads may mutate the shared state (beyond the course)

```
method inc(cell: Ref)
{
  acquire cell
  cell.val := cell.val + 1
  release cell
}
```

```
cell := new(val)
cell.val := 0
share cell
t1 := fork inc(cell)
t2 := fork inc(cell)
join t1
join t2
acquire cell
assert cell.val == 2
release cell
```

Coarse-grained locking

- Coarse-grained locking uses one lock for the entire data structure



The lock of the list guards accesses to the list and all nodes

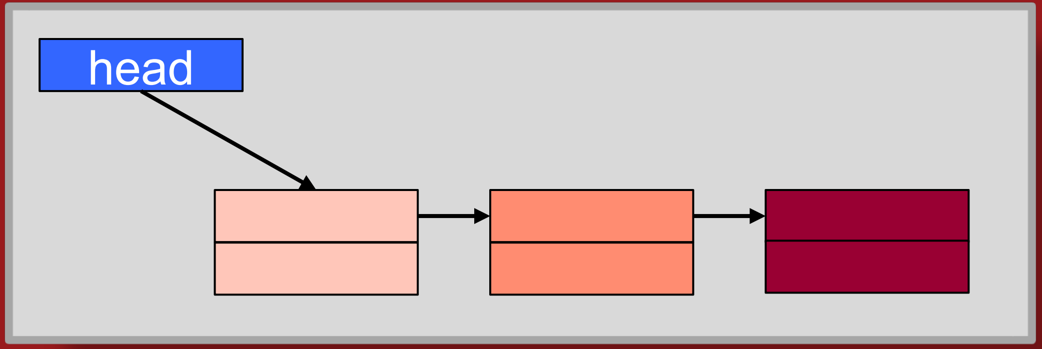
```
method incAll(this: Ref)
  requires acc(this.isLock, wildcard)
{
  acquire this
  if(this.head != null) {
      incAllNodes(this.head)
  }
  release this
}
```

Simple implementation, which uses sequential `incAllNodes` method, but limits concurrency

Exercise: fine-grained locking

➔ `12-fine-grained.vpr`

- Fine-grained locking uses multiple locks for the data structure to enable concurrent accesses



Each node has its own lock; multiple threads can traverse the list concurrently

- Implement `incAll` using fine-grained locking

Solution: fine-grained locking

- Implement `incAll` using fine-grained locking

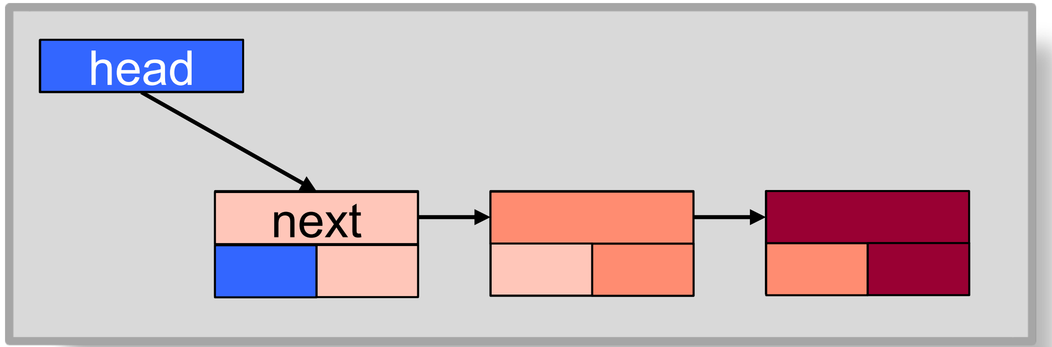
```
define InvList(this) (
  acc(this.head) &&
  (this.head != null ==>
      acc(this.head.isLock, wildcard))
)

define InvNode(this) (
  acc(this.elem) && acc(this.next) &&
  (this.next != null ==>
      acc(this.next.isLock, wildcard))
)
```

```
method incAll(this: Ref)
  requires acc(this.isLock, wildcard)
{
  acquire this
  var curr: Ref := this.head
  while(curr != null)
    invariant curr != null ==>
          acc(curr.isLock, wildcard)
  {
    acquireNode(curr)
    curr.elem := curr.elem + 1
    var n: Ref := curr.next
    releaseNode(curr)
    curr := n
  }
  release this
}
```

Fine-grained locking for complex invariants ➔ `13-hand-over-hand.vpr`

- Locking nodes in isolation is not possible if the lock invariants relates the states of multiple nodes (for instance, to express that the list is sorted)

- Such examples require hand-over-hand locking



Updating one `elem` field requires locking two nodes

```
define InvNode(this) (
  acc(this.elem, 1/2) && acc(this.next) &&
  (this.next != null ==> acc(this.next.elem, 1/2) &&
                         acc(this.next.isLock, wildcard) &&
                         this.elem <= this.next.elem)
)
```

Summary: concurrency

- We have seen that permissions enable verification for
  - Structured parallelism and threads
  - Data race freedom
  - Share mutable state and locks

- Many additional challenges exist

Properties	Implementations	Synchronization primitives
deadlock freedom, starvation freedom, fairness, linearizability, etc.	lock-free algorithms, weak-memory algorithms, etc.	messages, barriers, etc.

- Many of these are active research areas