A Basis for Verifying multi-threaded programs

Authors: K. Rustan Leino, P. Müller

Speaker: Martin Lanter

Challenges in multi-threading

- Fine-grained locking
- Thread-local and shared objects
- Distinguish between read and write access
- Prevent Deadlocks
  - Changable locking order
- Concurrent programs/data structures must be implemented correctly

Verifying multi-threaded programs

- Fine grained locking is hard to get right

- Verify against contracts
  - no deadlocks
  - no data races

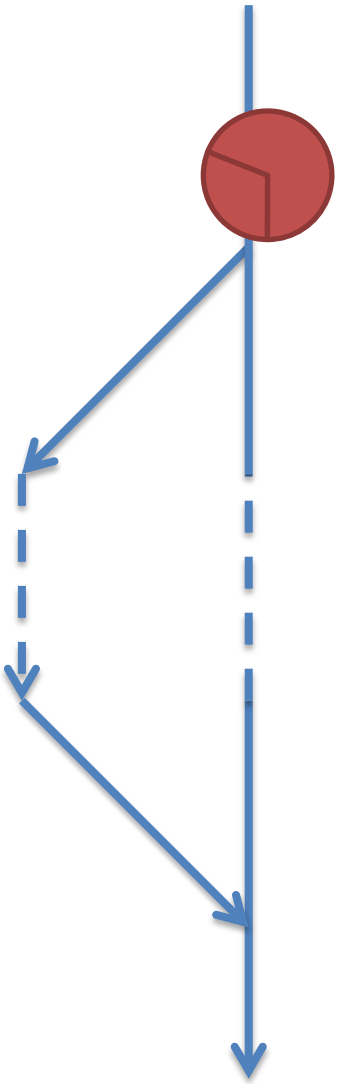
Permissions

- Between 0 and 100%
- Permission Holder
  - Threads
  - Monitors
  - System

- acc(f, n): n% permission for object f
- rd(f): infinitesimal permissions for object f

Permissions

- Obtain Permission by
  - Creating a new object (100%)
  - Acquire an object's monitor (Invariant)
  - Be forked from another thread (Precond.)
  - Join another thread (Postcond.)



Contracts with Permissions

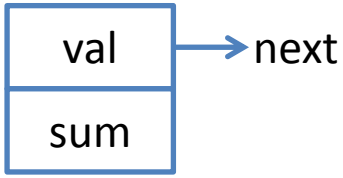
```
int apple;
int lemon;

void foo()
    requires acc(apple)
    ensures acc(lemon)
{
    apple = 5;
    bar();
    lemon = 7;
    ...
}
```

```
void bar()
    requires rd(apple)
    ensures rd(apple) ∧
            acc(lemon)
{ ... }
```

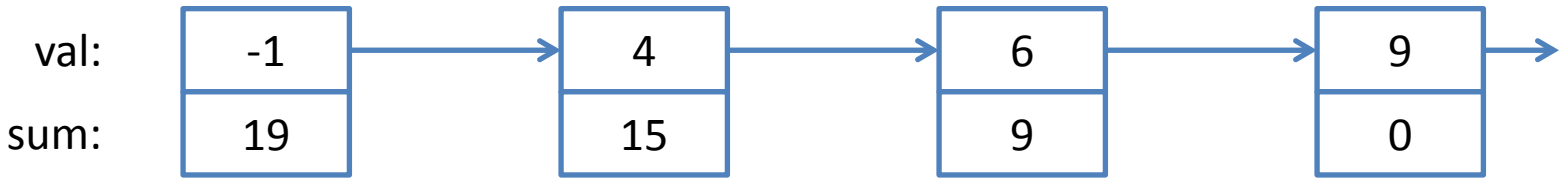
Sorted Linked List

Node:



μ: monitor's position in locking order

Sorted Linked List



Node's invariant:

a) *acc(next, 100) ∧ rd(val)*

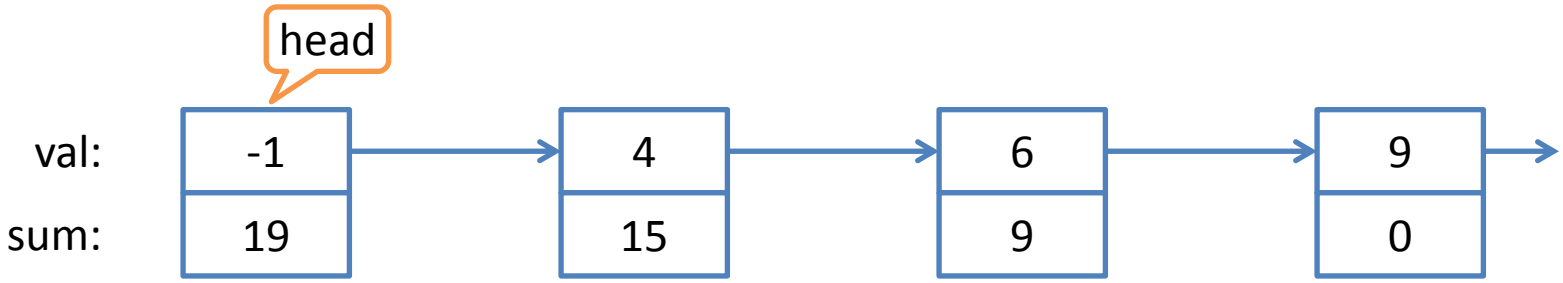
b) *next ≠ null → rd(next.val) ∧ val ≤ next.val*

c) *next ≠ null → acc(next.sum, 50) ∧ sum = next.val + next.sum*

d) *acc(sum, 50) ∧ (next = null → sum = 0)*

e) *acc(μ, 50) ∧ (next ≠ null → acc(next.μ, 50) ∧ μ ⊏ next.μ)*

Insert x=7



invariant $head \neq null \wedge acc(head) \wedge acc(head.sum, 50)$

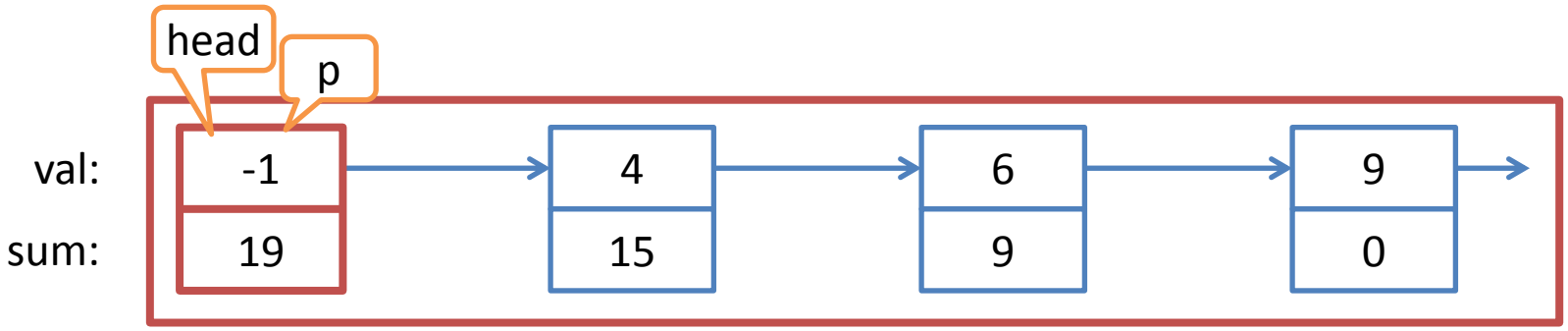
```
void Insert(x)
```
requires $rd(\mu) \wedge maxlock \subset \mu \wedge 0 \leq x;$

ensures $rd(\mu) \wedge maxlock \subset \mu \wedge$

$head.sum = old(head.sum) + x;$

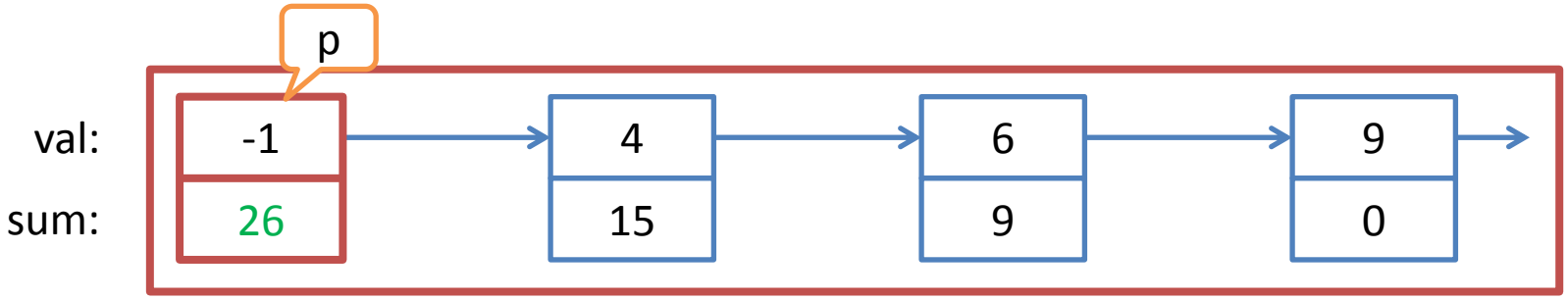
```
{
...
```

Insert x=7



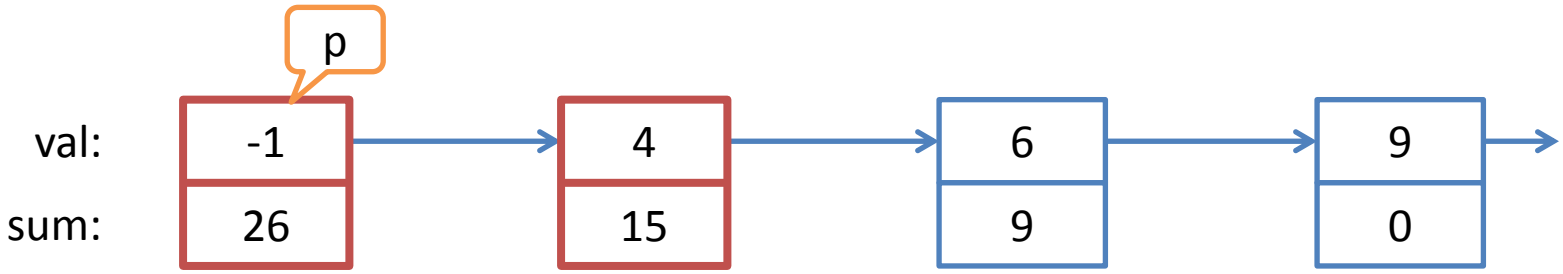

```
acquire this;
Node p = head;
acquire p;
p.sum = p.sum + x;
release this;
...
```

Insert x=7



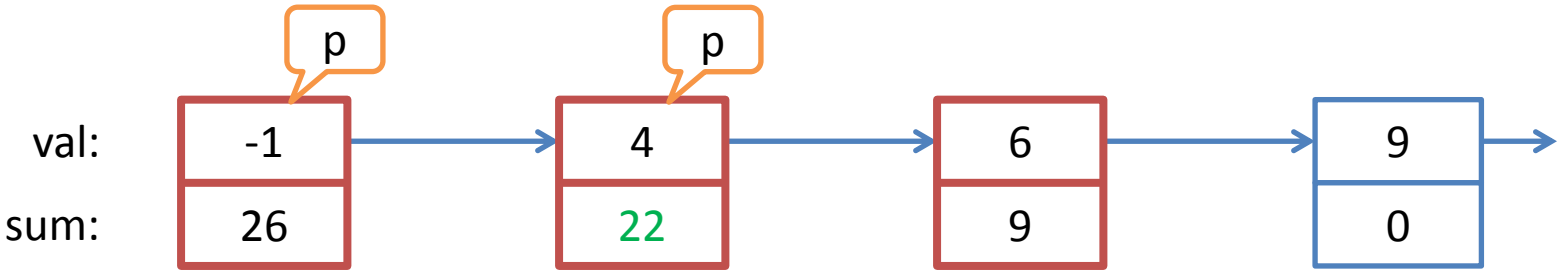

```
acquire this;
Node p = head;
acquire p;
p.sum = p.sum + x;
release this;
...
```

Insert x=7



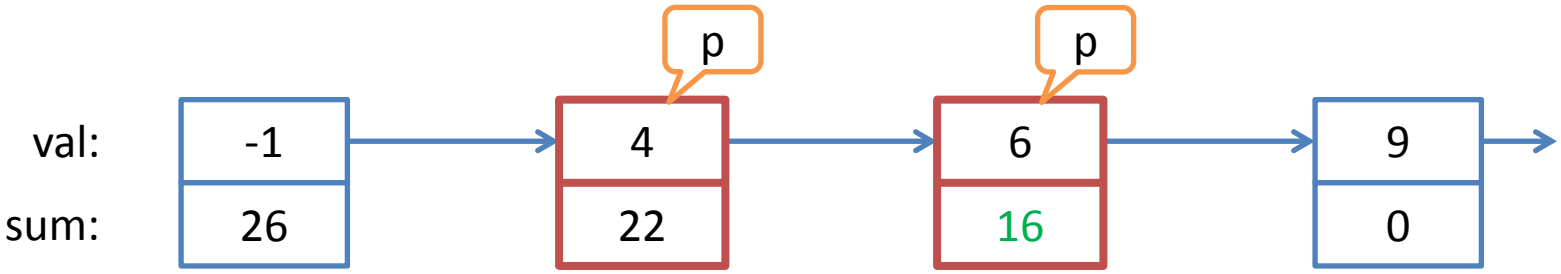

```
while (p.next ≠ null ∧ p.next.val < x)
    // loop invariant
{
    Node nx = p.next;
    acquire nx;
    nx.sum = nx.sum + x;
    release p;
    p = nx;
}
...
```

Insert x=7



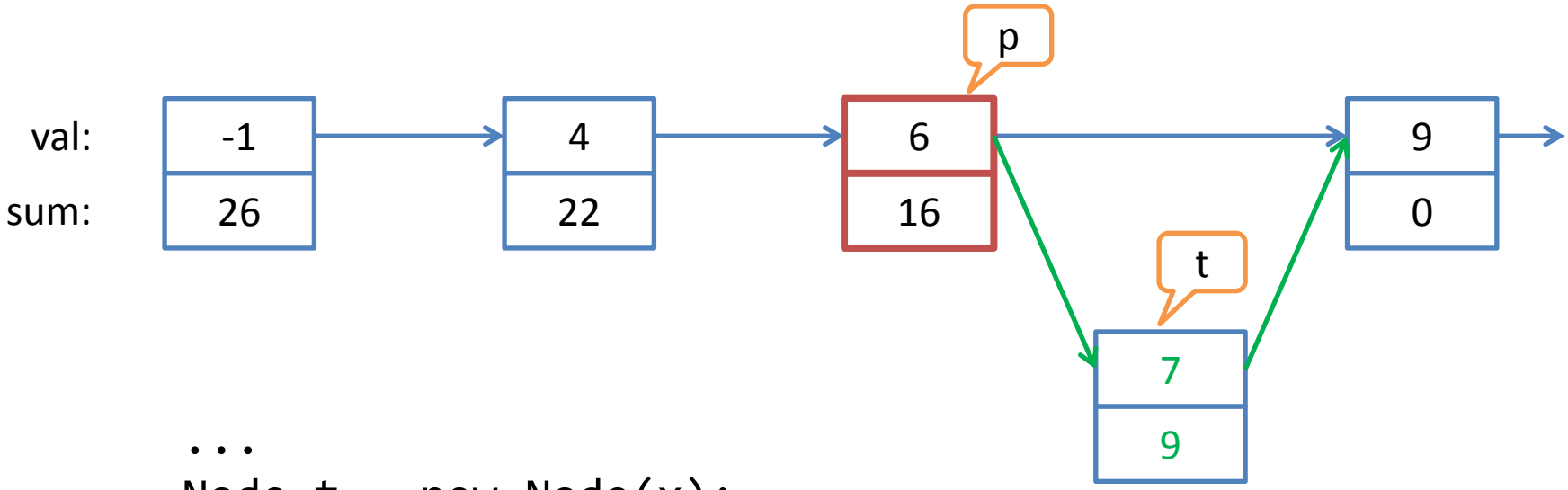

```
while (p.next ≠ null ∧ p.next.val < x)
    // loop invariant
{
    Node nx = p.next;
    acquire nx;
    nx.sum = nx.sum + x;
    release p;
    p = nx;
}
...
```

Insert x=7




```
while (p.next ≠ null ∧ p.next.val < x)
    // loop invariant
{
    Node nx = p.next;
    acquire nx;
    nx.sum = nx.sum + x;
    release p;
    p = nx;
}
...
```

Insert x=7




```
...
Node t = new Node(x);
t.sum = p.next.val + p.next.sum;
t.next = p.next;
share t between p and p.next;
p.next = t;
release p;
} // Insert(x)
```

Verification

- *Permission: (p,n)*
- *P[f]: Permission for location f*
- *CanRead(f) ≡* ***let*** *(p,n) = P[f]* ***in*** *p > 0 ∨ n > 0*
- *CanWrite(f) ≡* ***let*** *(p,n) = P[f]* ***in*** *p = 100 ∧ n = 0*

Field Access

$$x := o.f; \quad \equiv$$
$$\quad \textbf{assert } CanRead(o.f);$$
$$\quad x := Heap[o, f];$$

$$o.f := x; \quad \equiv$$
$$\quad \textbf{assert } CanWrite(o.f);$$
$$\quad Heap[o, f] := x;$$

Acquiring and Releasing

$$
\begin{array}{l}
\textbf{acquire}\ o; \quad \equiv \\
\quad \textbf{assert}\ CanRead(o.\mu); \\
\quad \textbf{assert}\ (\forall\, p \bullet\ Heap[p, held] \Rightarrow Heap[p, \mu] \sqsubset Heap[o, \mu]\ ); \\
\quad Heap[o, held] := \textbf{true}; \\
\quad \mathsf{Inhale}[\![J(o)]\!]
\end{array}
$$

$$
\begin{array}{l}
\textbf{release}\ o; \quad \equiv \\
\quad \textbf{assert}\ o \neq null; \\
\quad \textbf{assert}\ Heap[o, held]; \\
\quad \mathsf{Exhale}[\![J(o)]\!] \\
\quad Heap[o, held] := \textbf{false};
\end{array}
$$

Exhale and Inhale

$$\begin{array}{l}
\mathsf{Exhale}[\![E]\!] \equiv \\
\quad \textbf{assert } \mathsf{Tr}[\![E]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Inhale}[\![E]\!] \equiv \\
\quad \textbf{assume } \mathsf{Tr}[\![E]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Exhale}[\![P \wedge Q]\!] \equiv \\
\quad \mathsf{Exhale}[\![Q]\!]; \\
\quad \mathsf{Exhale}[\![P]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Inhale}[\![P \wedge Q]\!] \equiv \\
\quad \mathsf{Inhale}[\![P]\!]; \\
\quad \mathsf{Inhale}[\![Q]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Exhale}[\![P \Rightarrow Q]\!] \equiv \\
\quad \textbf{if } (\mathsf{Tr}[\![P]\!]) \{ \mathsf{Exhale}[\![Q]\!]; \}
\end{array}$$

$$\begin{array}{l}
\mathsf{Inhale}[\![P \Rightarrow Q]\!] \equiv \\
\quad \textbf{if } (\mathsf{Tr}[\![P]\!]) \{ \mathsf{Inhale}[\![Q]\!]; \}
\end{array}$$

$$\begin{array}{l}
\mathsf{Exhale}[\![acc(E.f, r)]\!] \equiv \\
\quad \textbf{assert } \mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \mathsf{Tr}[\![r]\!]; \\
\quad \mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \mathsf{Tr}[\![r]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Inhale}[\![acc(E.f, r)]\!] \equiv \\
\quad \textbf{if } (\mathcal{P}[\mathsf{Tr}[\![E]\!], f] = (0, 0)) \\
\quad\quad \textbf{havoc } \mathit{Heap}[\mathsf{Tr}[\![E]\!], f]; \\
\quad \mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] + \mathsf{Tr}[\![r]\!];
\end{array}$$

$$\begin{array}{l}
\mathsf{Exhale}[\![rd(E.f)]\!] \equiv \\
\quad \textbf{assert } \mathcal{P}[\mathsf{Tr}[\![E]\!], f] \geq \varepsilon; \\
\quad \mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] - \varepsilon;
\end{array}$$

$$\begin{array}{l}
\mathsf{Inhale}[\![rd(E.f)]\!] \equiv \\
\quad \textbf{if } (\mathcal{P}[\mathsf{Tr}[\![E]\!], f] = (0, 0)) \\
\quad\quad \{ \textbf{havoc } \mathit{Heap}[\mathsf{Tr}[\![E]\!], f]; \} \\
\quad \mathcal{P}[\mathsf{Tr}[\![E]\!], f] := \mathcal{P}[\mathsf{Tr}[\![E]\!], f] + \varepsilon;
\end{array}$$

Conclusion

- Verification against contracts
- Verification methodology
  - No data races
  - No deadlocks
  - Fine-grained locking
  - Sharing and unsharing of objects
  - Expressive
- Future work: Formal proof of soundness

Personal Opinion

- Complicated
- Great for critical datastructures

Variables

- acc(f, n): n% permission for object f
- rd(f): infinitesimal permissions for object f
- μ: lock position in locking order
- u $\subset$ *v: u's monitor position is strictly less than v*
- share, unshare: Functions to share objects between threads
- Inhale(J): overtake permissions granted by invariant J
- Exhale(J): hand over permissions granted by invariant J
- held: true if a thread has acquired an objects monitor
- reorder: reorder monitor position between others
- $\bar{p}$: list of monitors
- maxlock: maximum position of acquired monitors
- $\perp$: bottom monitor position
- havoc: assigns an arbitrary value to be constrained by following assume

Additional Slides

$$x := o.f; \quad \equiv$$
$$\quad \textbf{assert } CanRead(o.f);$$
$$\quad x := Heap[o, f];$$

$$o.f := x; \quad \equiv$$
$$\quad \textbf{assert } CanWrite(o.f);$$
$$\quad Heap[o, f] := x;$$

$$\textbf{reorder } o \textbf{ between } \overline{p} \textbf{ and } \overline{s}; \quad \equiv$$
$$\quad \textbf{assert } CanWrite(o, \mu) \wedge Heap[o, held];$$
$$\quad \#\textbf{foreach } p_i \in \overline{p}, s_j \in \overline{s} \, \{$$
$$\quad\quad \textbf{assert } p_i = null \vee s_j = null \vee$$
$$\quad\quad\quad (CanRead(p_i.\mu) \wedge CanRead(s_j.\mu) \wedge Heap[p_i, \mu] \sqsubset Heap[s_j, \mu]);$$
$$\quad \}$$
$$\quad \textbf{havoc } w;$$
$$\quad \#\textbf{foreach } p_i \in \overline{p} \, \{ \textbf{ assume } p_i = null \vee Heap[p_i, \mu] \sqsubset w; \} ;$$
$$\quad \#\textbf{foreach } s_j \in \overline{s} \, \{ \textbf{ assume } s_j = null \vee w \sqsubset Heap[s_j, \mu]; \} ;$$
$$\quad Heap[o, \mu] := w;$$

Additional Slides

share o **between** $\overline{p}$ **and** $\overline{s}$; $\equiv$
  assert $CanWrite(o, \mu)$;
  assert $Heap[o, \mu] = \bot$;
  // see $(*)$ of **reorder**
  Exhale$[\![J(o)]\!]$

unshare o; $\equiv$
  assert $CanWrite(o, \mu)$;
  assert $Heap[o, held]$;
  $Heap[o, held] := \mathbf{false}$;
  $Heap[o, \mu] := \bot$;

join o; $\equiv$
  assert $CanWrite(o.active)$;
  assert $Heap[o, active]$;
  $Heap[o, active] := false$;
  Inhale$[\![RunPost(o)]\!]$

fork o; $\equiv$
  assert $CanWrite(o.active)$;
  assert $\neg Heap[o, active]$;
  Exhale$[\![RunPre(o)]\!]$
  $Heap[o, active] := true$;