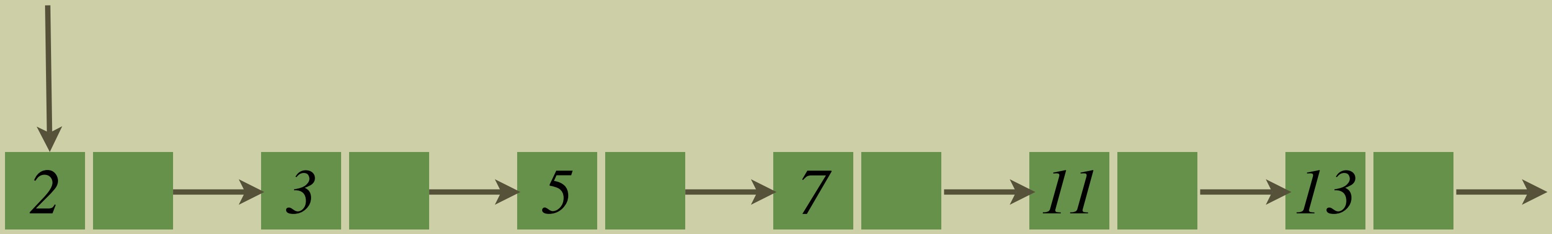
An introduction to RGSep

Viktor Vafeiadis

Example: lock-coupling list

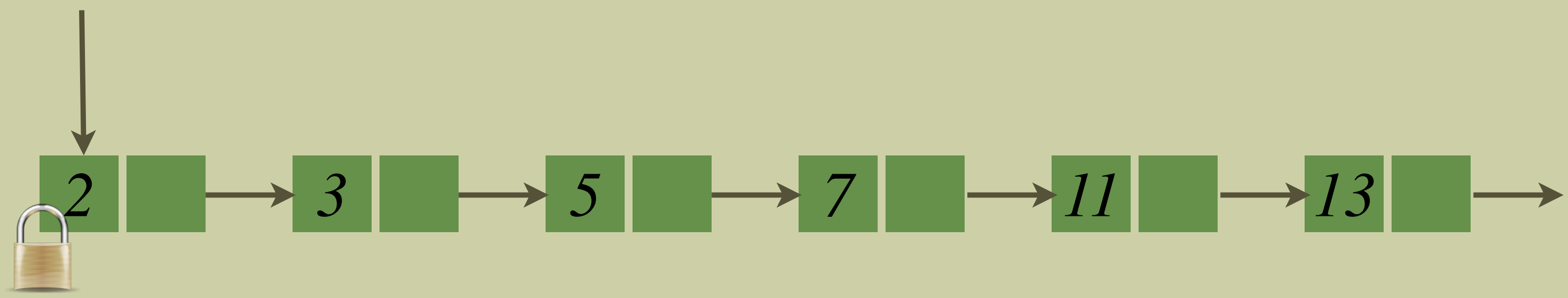


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

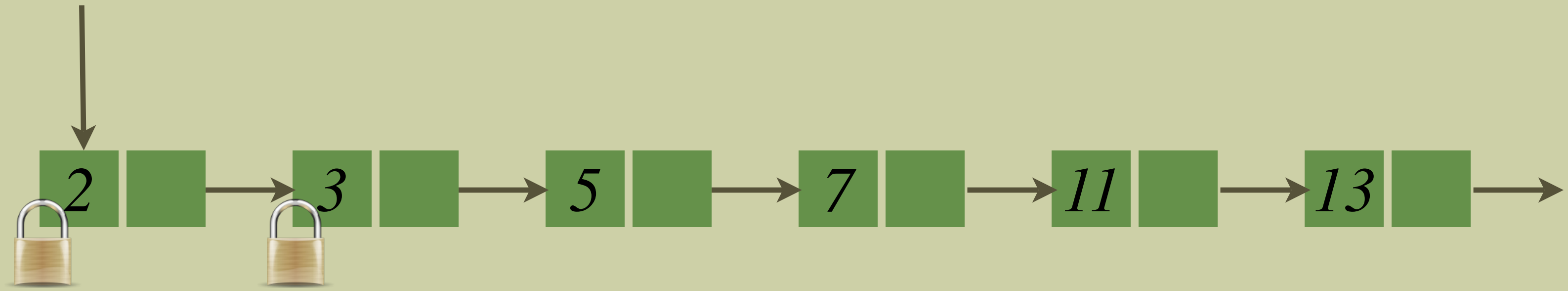


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

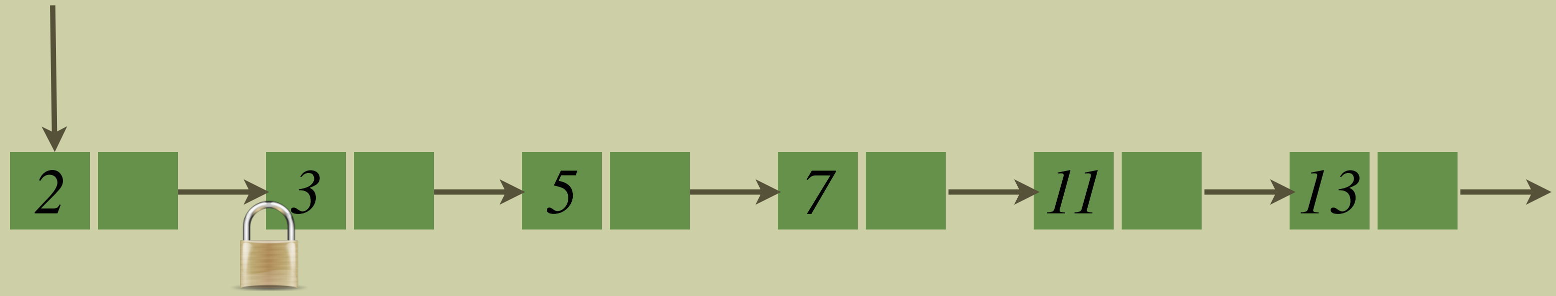


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

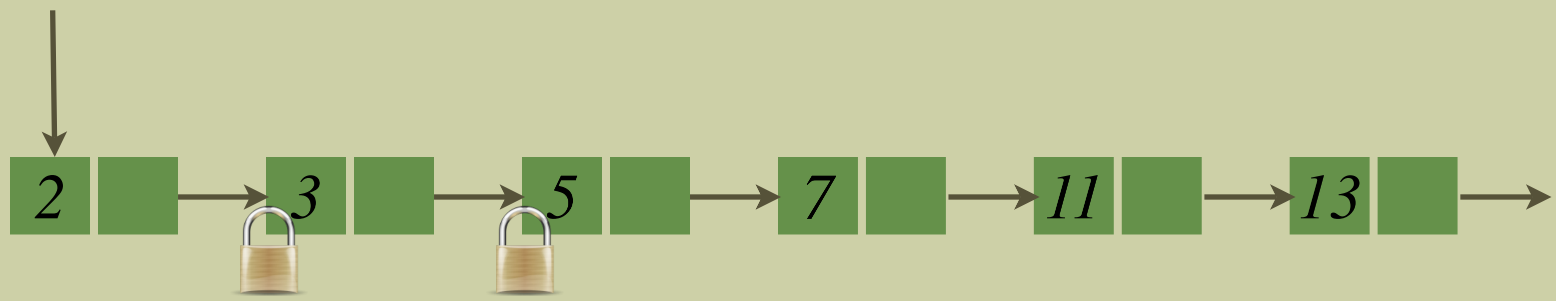


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

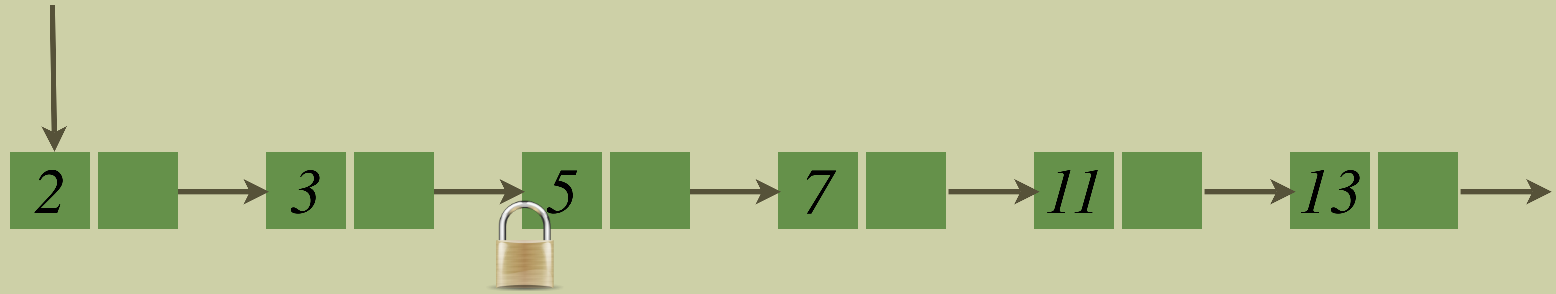


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

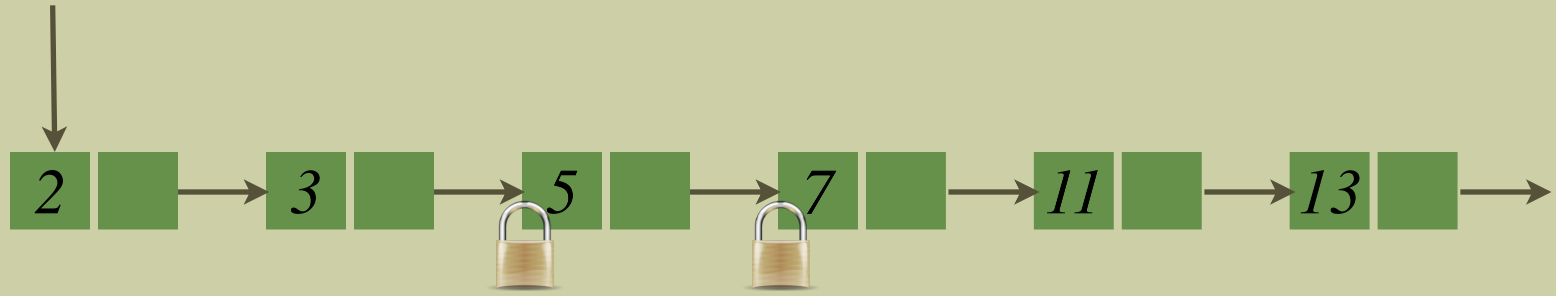


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list

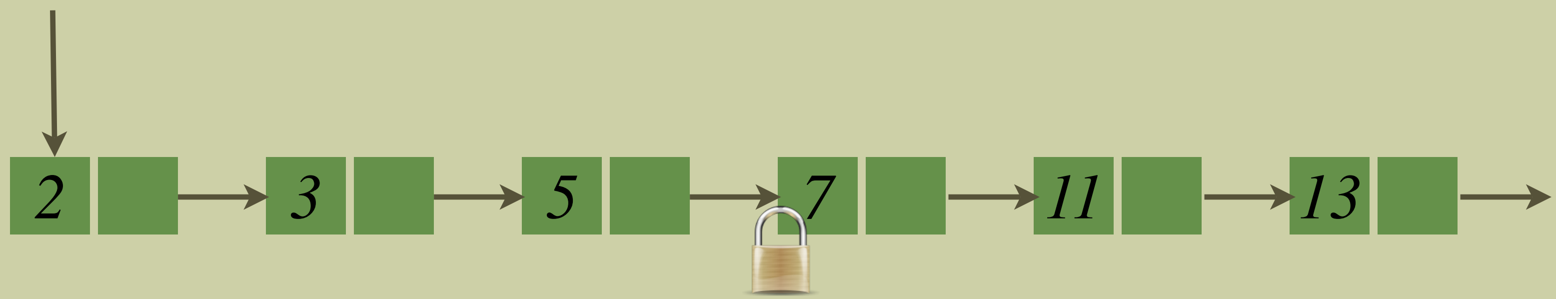


There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

Example: lock-coupling list



There is one lock per node; threads acquire locks in a hand over hand fashion.

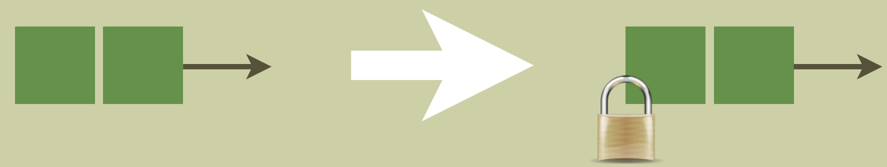
If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.

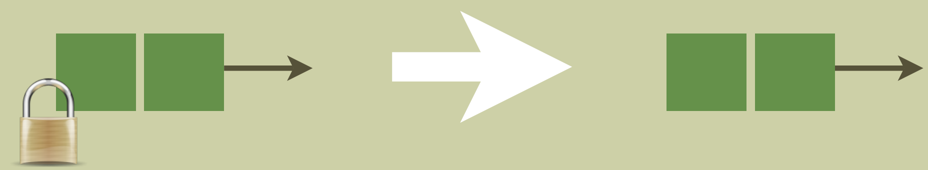
Operations (actions)

Lock-coupling list

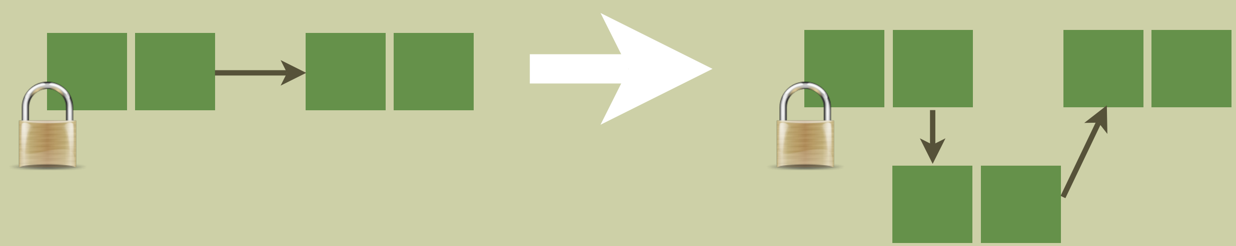
Lock



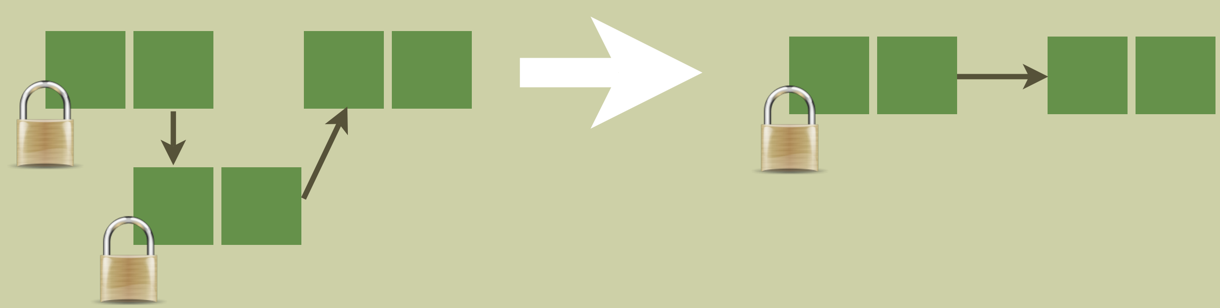
Unlock



Add node



Delete node



Part I. Basic concepts

- Local & shared state
- Actions
- Program & environment
- Program specifications
- Stability

Local & shared state

The total state is logically divided into two components:

- ***Shared:*** accessible by all threads via synchronisation
- ***Local:*** accessible only by one thread, its owner

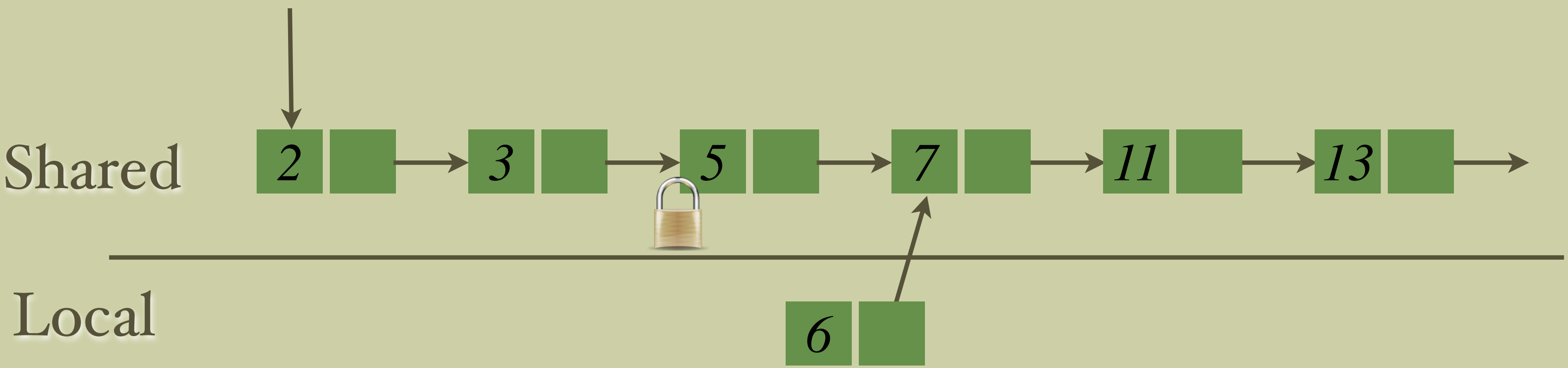
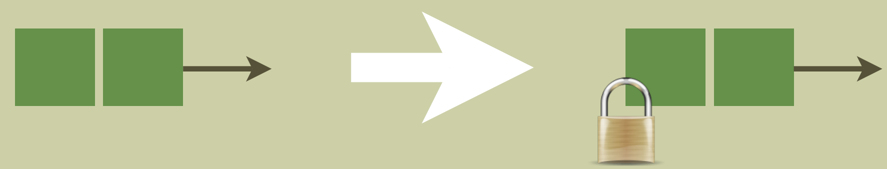


Figure. State of the lock-coupling list just before inserting a new node. The node to be added is local because other threads cannot yet access it.

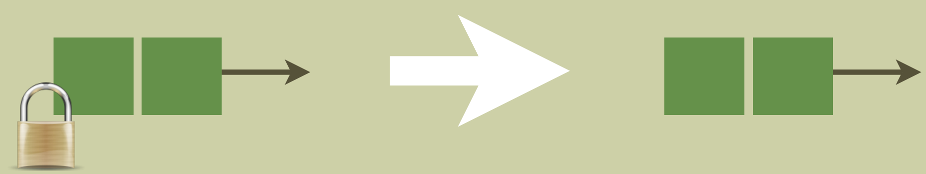
Actions (1/3)

Actions describe minimal atomic changes to the shared state.

Lock



Unlock

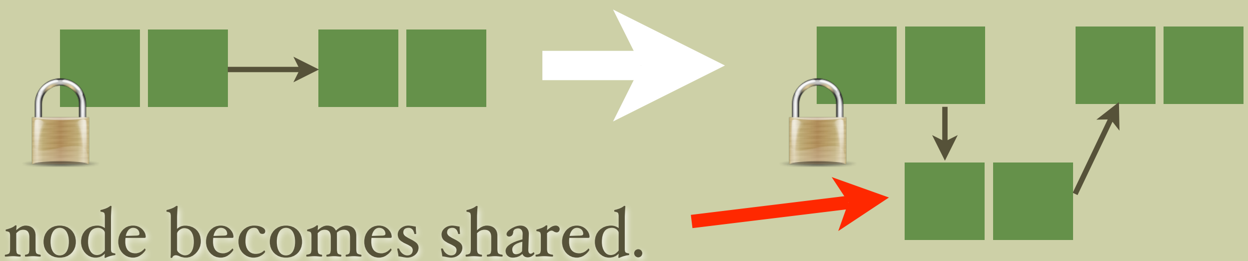


An action allows any part of the ***shared state*** that satisfies the LHS to be changed to a part satisfying the RHS, but the rest of the shared state must not be changed.

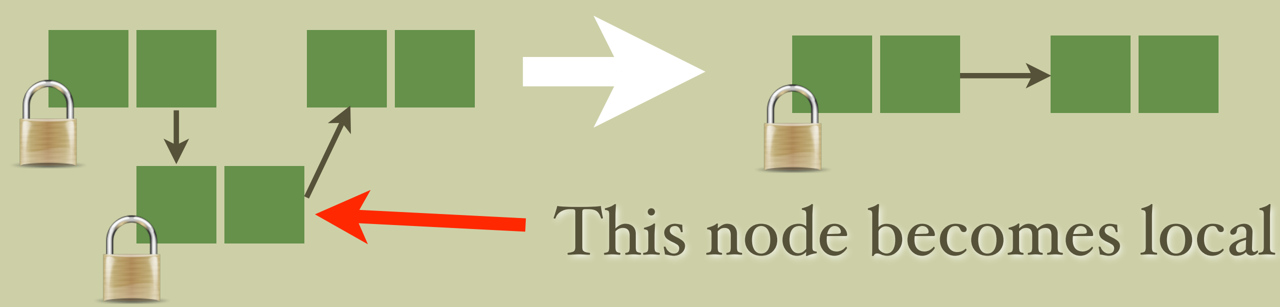
Actions (2/3)

Actions can also adjust the boundary between local state and stared state. This is also known as ***tranfer of ownership***.

Add node

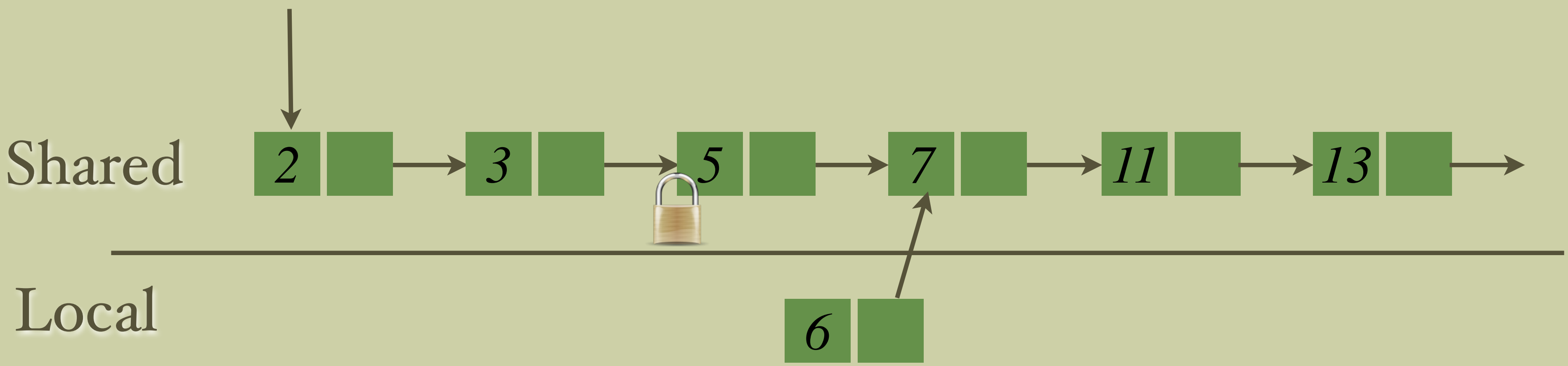


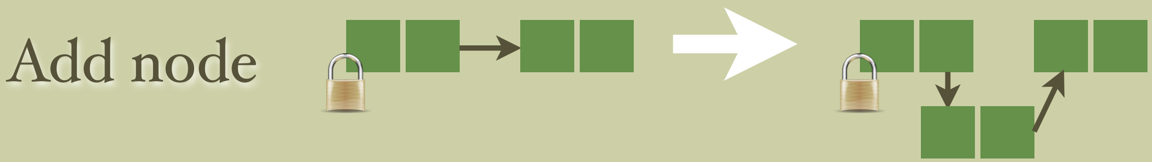
Delete node



Actions (3/3)

Example: Lock coupling list

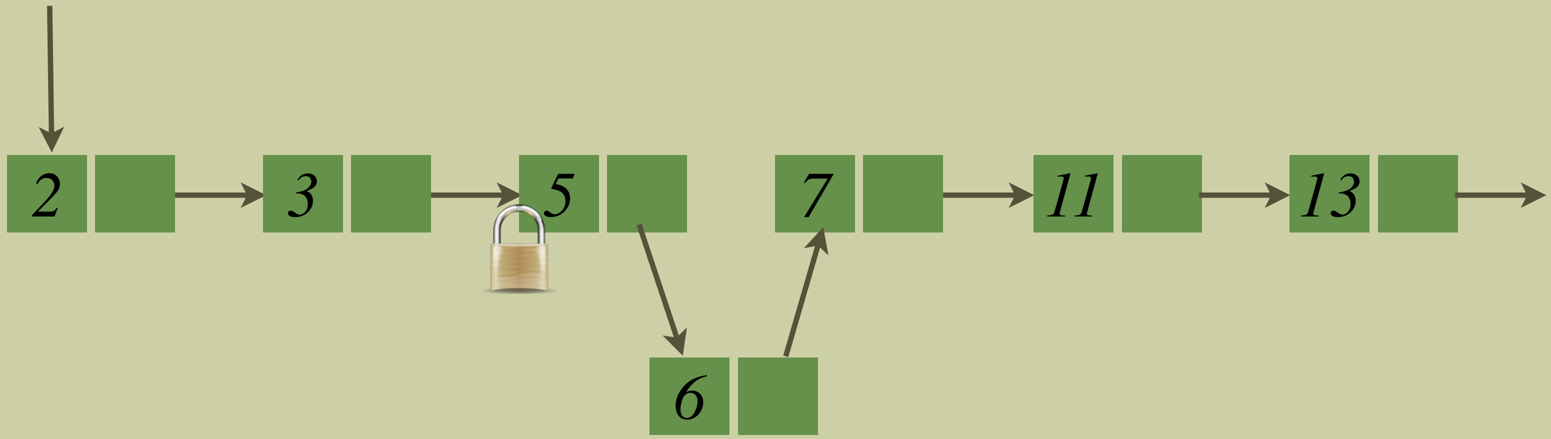




Actions (3/3)

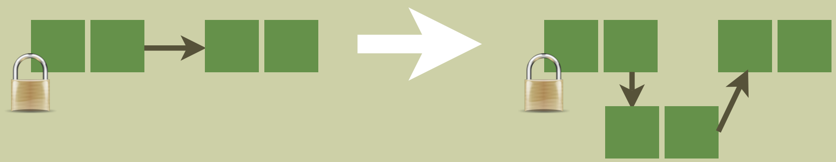
Example: Lock coupling list

Shared



Local

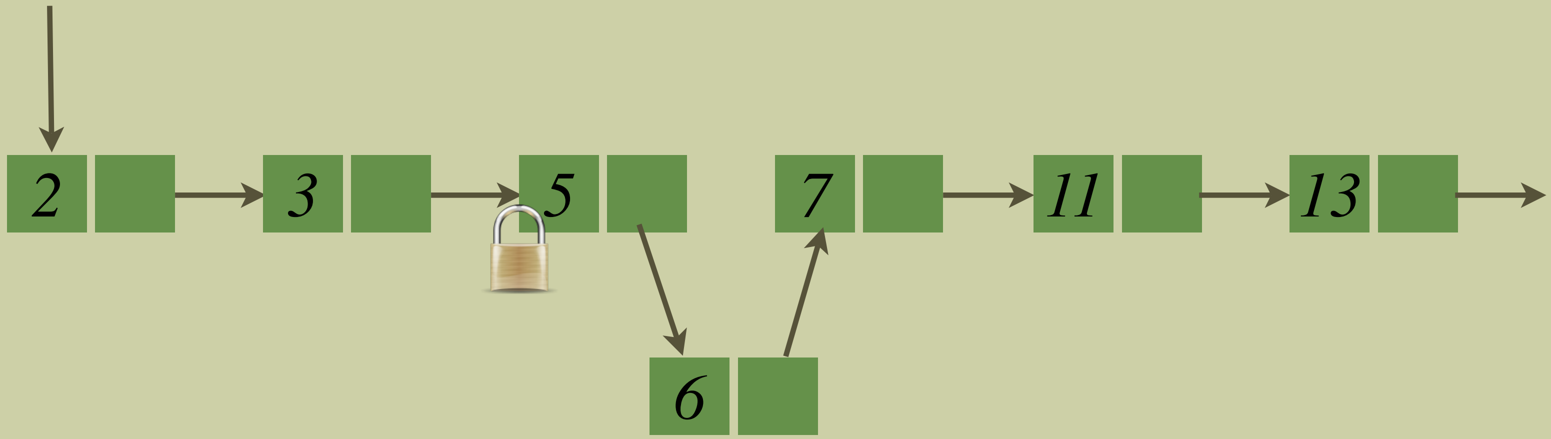
Add node



Actions (3/3)

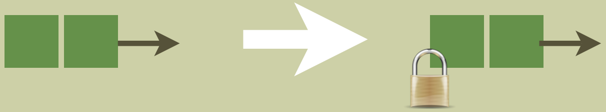
Example: Lock coupling list

Shared



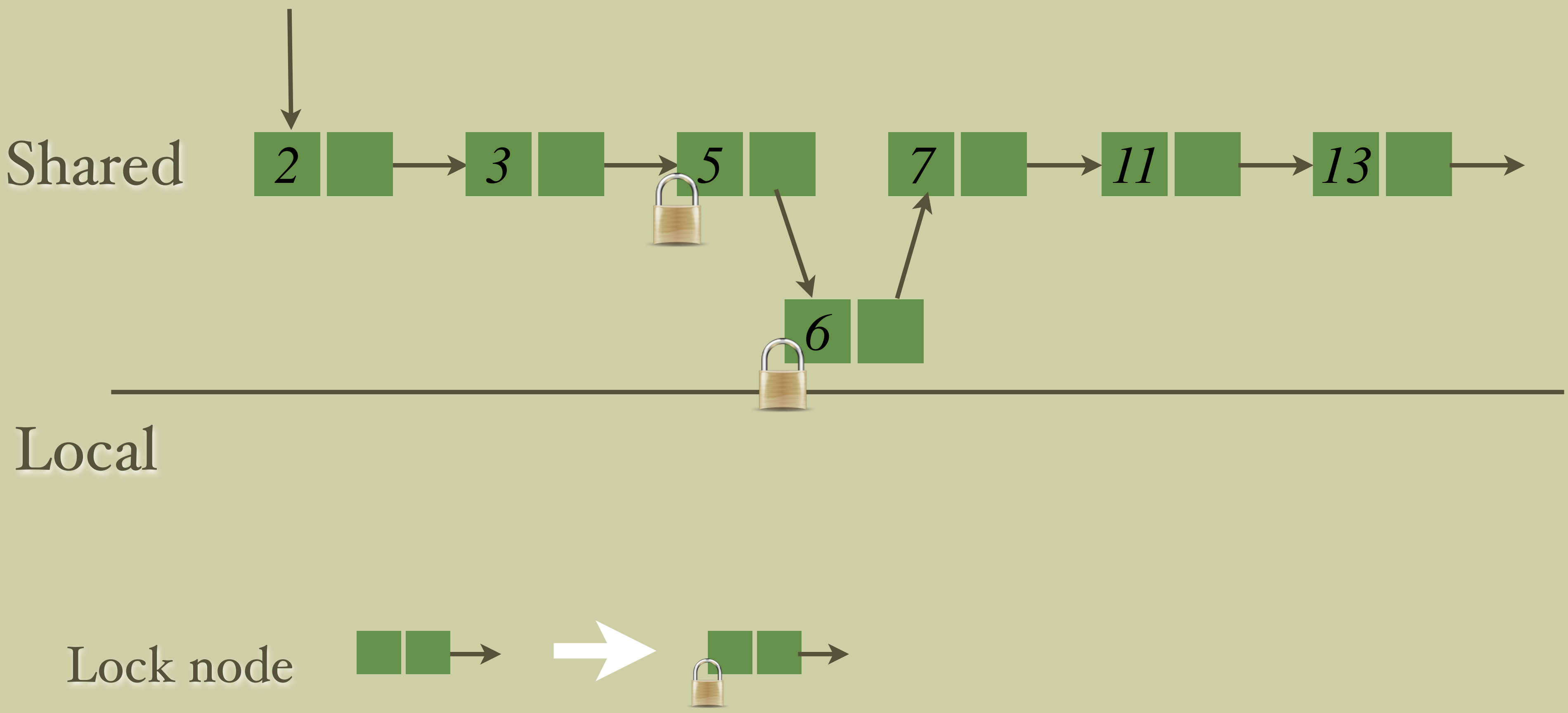
Local

Lock node



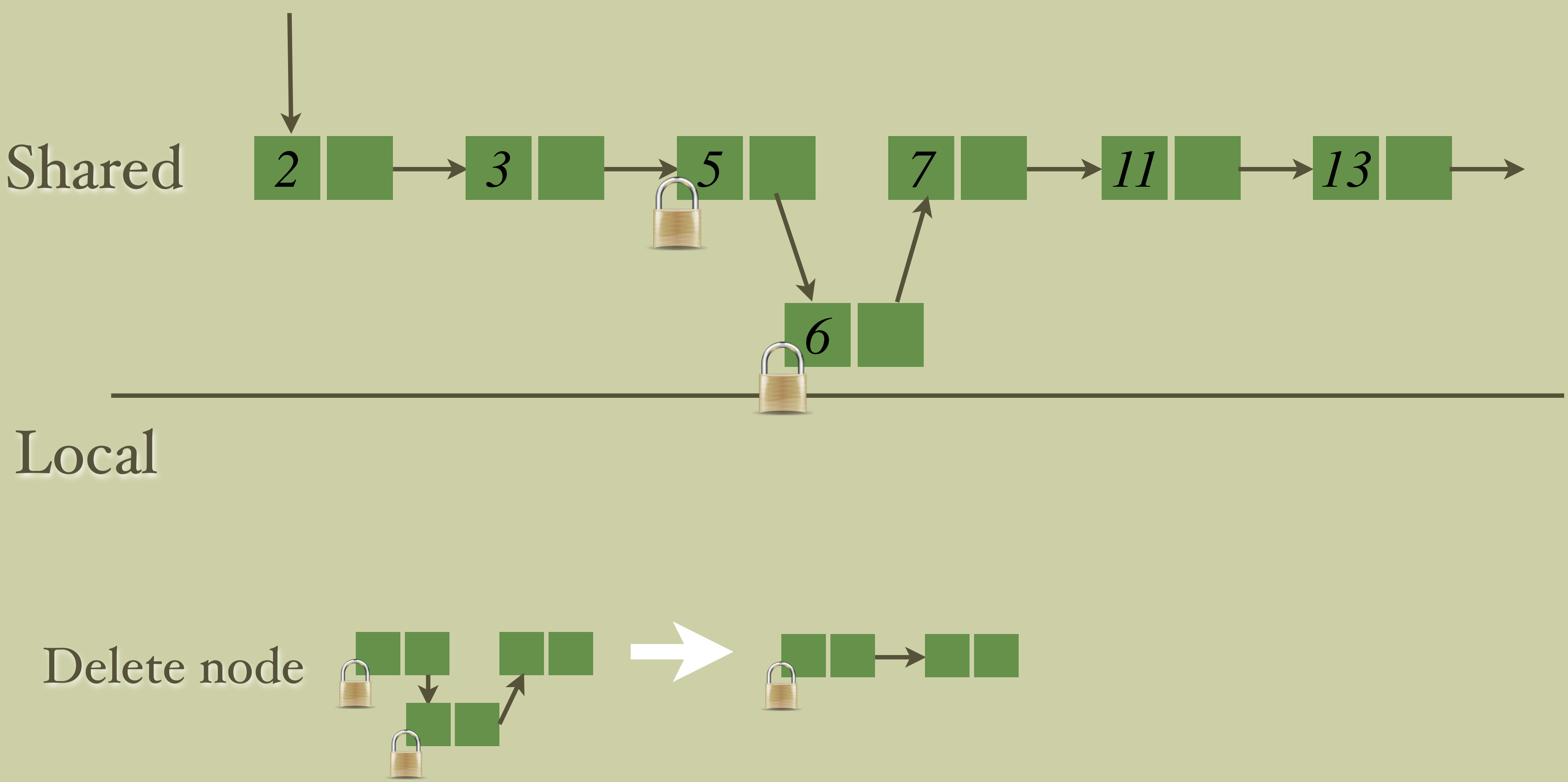
Actions (3/3)

Example: Lock coupling list



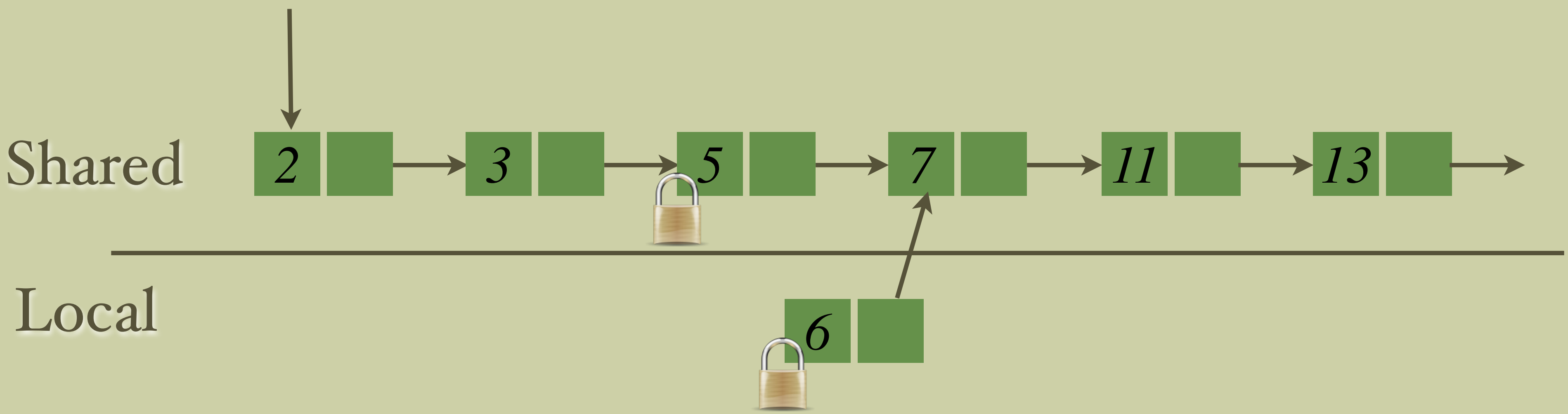
Actions (3/3)

Example: Lock coupling list

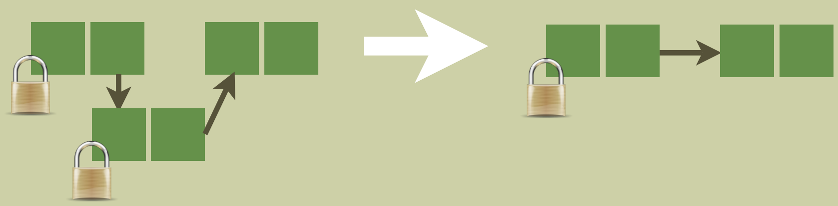


Actions (3/3)

Example: Lock coupling list

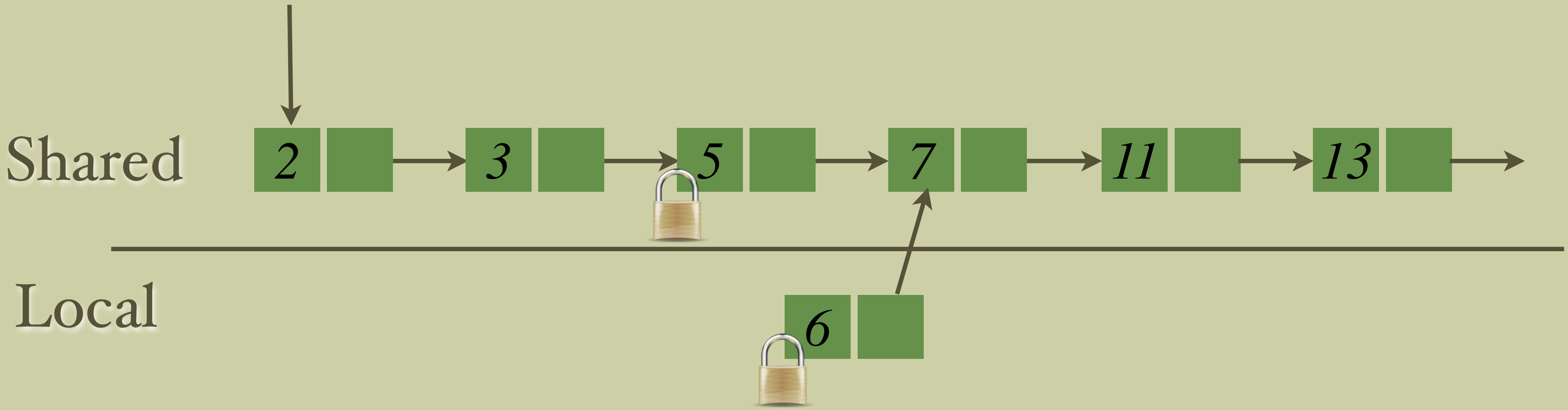


Delete node



Actions (3/3)

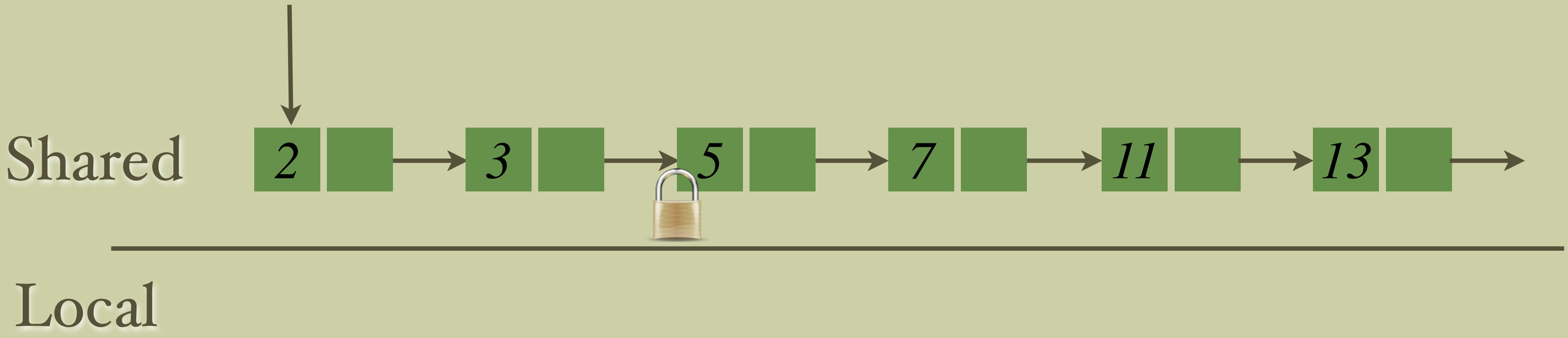
Example: Lock coupling list



Now, the node is local; we can safely dispose it.

Actions (3/3)

Example: Lock coupling list



Now, the node is local; we can safely dispose it.

Program & environment

Program: the current thread being verified.

Environment: all other threads of the system that execute in parallel with the thread being verified.

Interference. The program interferes with the environment by modifying the shared state. Conversely, the environment interferes with the program by modifying the shared state.

Program specifications

The specification of a program consists of two assertions (precondition & postcondition), and two sets of actions:

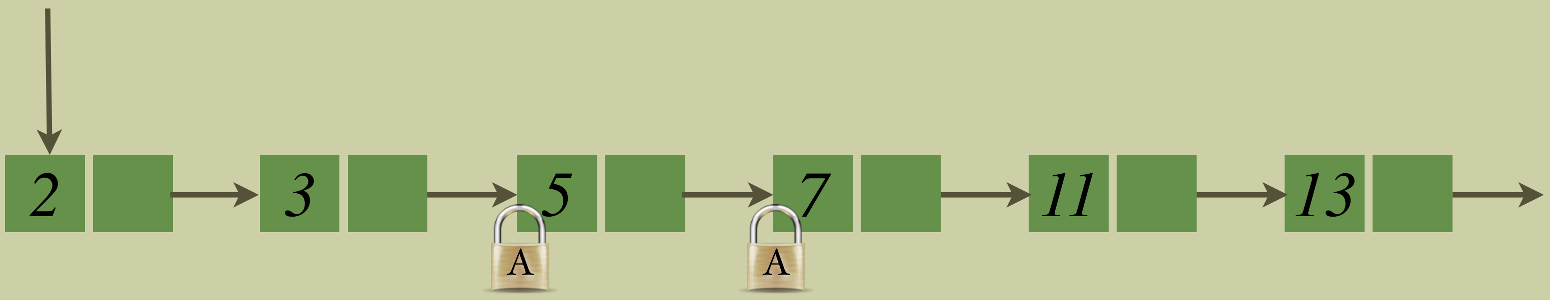
- ***Rely:*** Describes the interference that the program can tolerate from the environment; i.e. specifies how the environment can change the shared state.

- ***Guarantee:*** Describes the interference that the program imposes on its environment; i.e. specifies how the program can change the shared state.

Stability (1/2)

Definition. An assertion is stable if and only if it is preserved under interference by other threads.

Example 1. The following assertion is not stable.

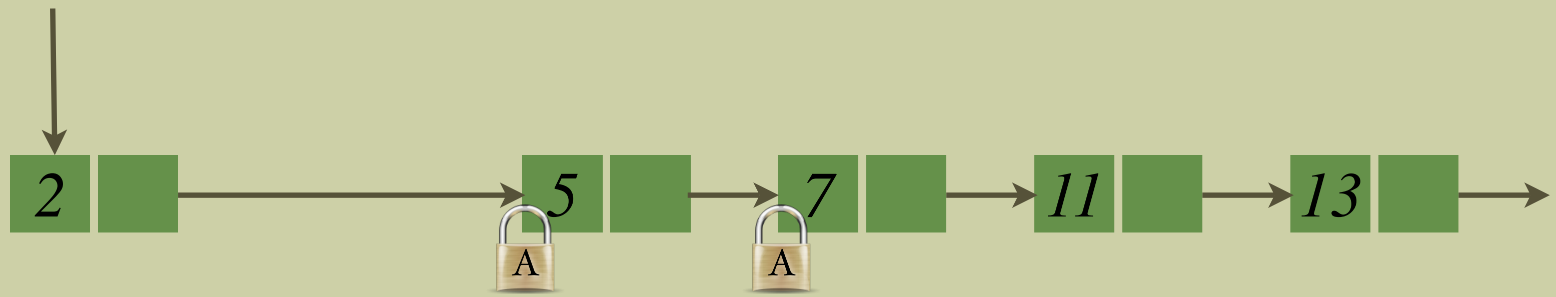


For instance, another thread could remove node 3 or add a node after node 11.

Stability (1/2)

Definition. An assertion is stable if and only if it is preserved under interference by other threads.

Example 1. The following assertion is not stable.

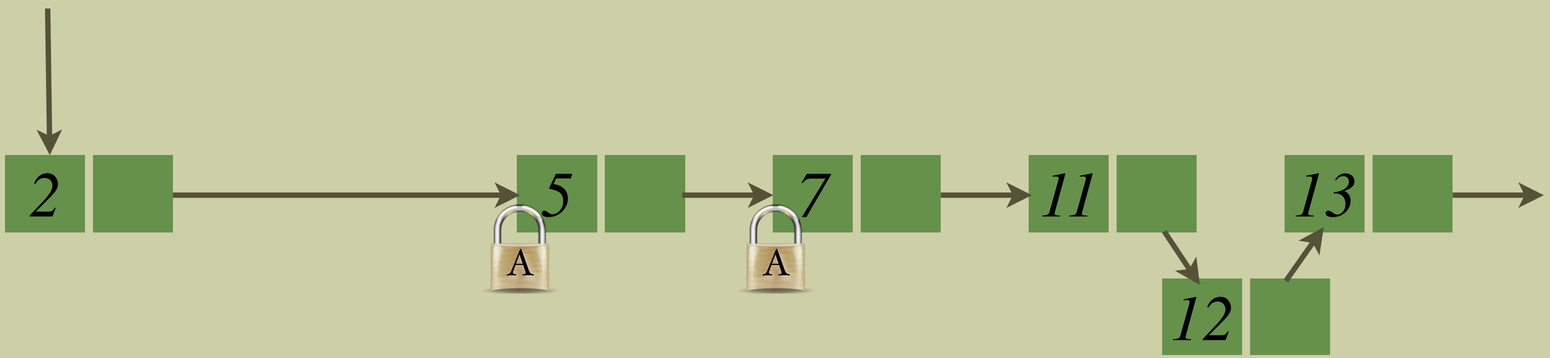


For instance, another thread could remove node 3 or add a node after node 11.

Stability (1/2)

Definition. An assertion is stable if and only if it is preserved under interference by other threads.

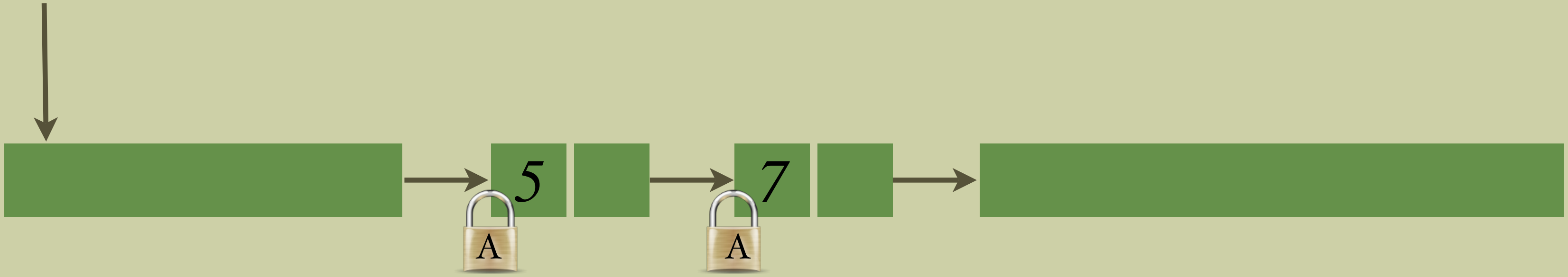
Example 1. The following assertion is not stable.



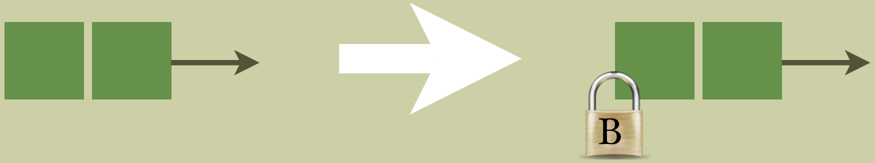
For instance, another thread could remove node 3 or add a node after node 11.

Stability (2/2)

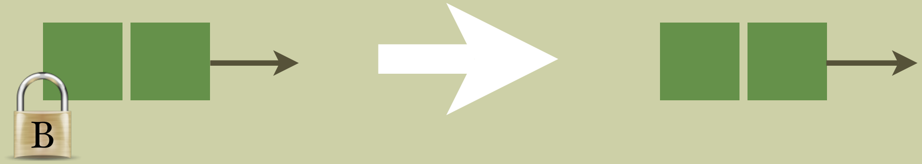
Example 2. The following assertion, however, is stable.



Lock

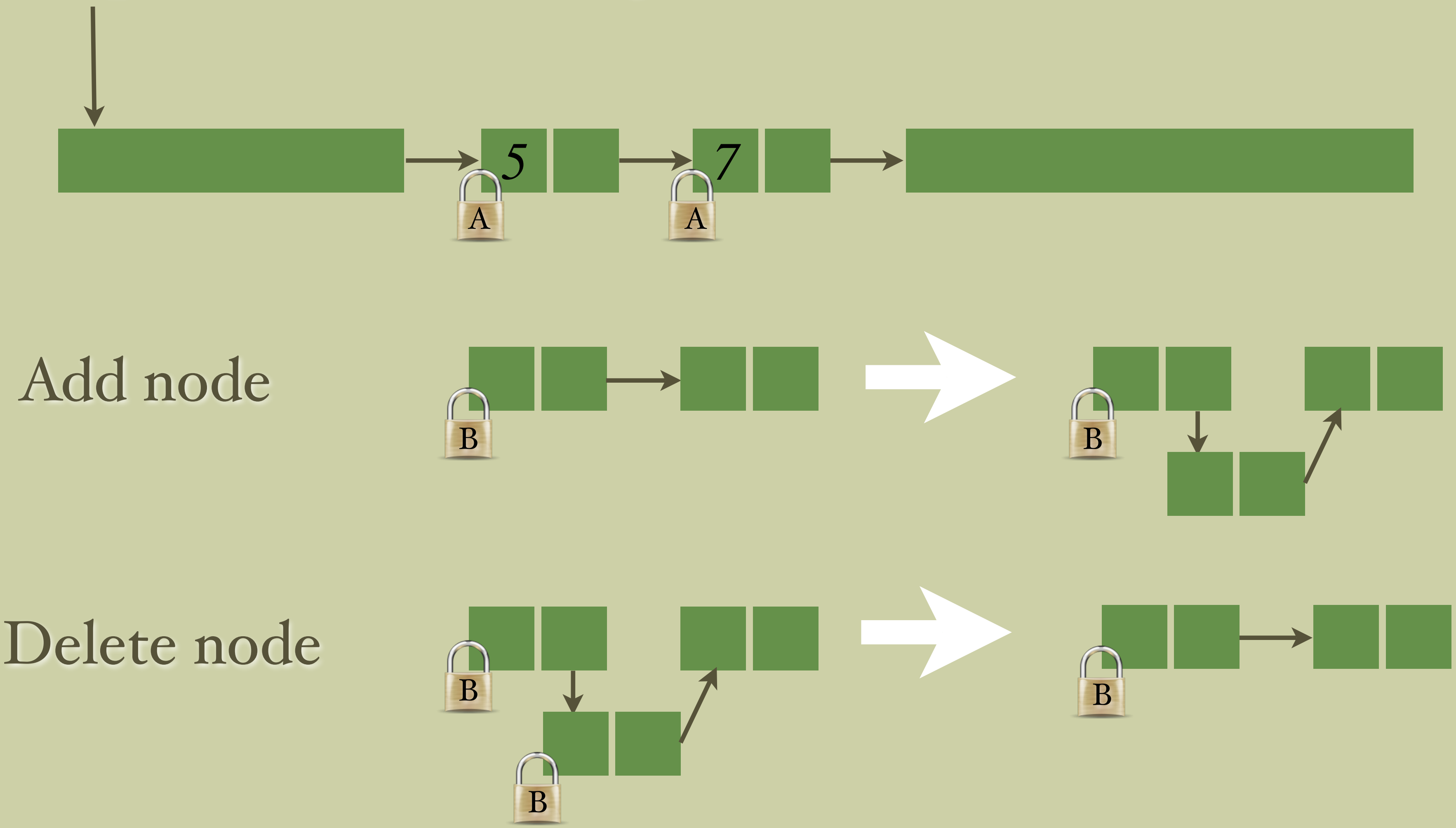


Unlock



Stability (2/2)

Example 2. The following assertion, however, is stable.



Part II. Program logic

- Syntax & semantics of assertions
- Syntax & semantics of actions
- Syntax & semantics of judgements
- Some proof rules
- Checking stability

Assertion syntax

Separation Logic

$$
\begin{aligned}
P, Q ::= {} & \mathbf{false} \mid \mathbf{emp} \mid e = e' \mid e \mapsto e' \\
& \mid \exists x.\, P \mid P \Rightarrow Q \mid P * Q \mid P \mathbin{-\!\!\circledast} Q
\end{aligned}
$$

$$
\begin{aligned}
h \vDash_{\mathsf{SL}} P \mathbin{-\!\!\circledast} Q \quad & \Longleftrightarrow \quad h \vDash_{\mathsf{SL}} \neg(P \mathbin{-\!\!*} \neg Q) \\
& \Longleftrightarrow \quad \exists h'.\, (h' \vDash_{\mathsf{SL}} P) \wedge (h \uplus h' \vDash_{\mathsf{SL}} Q)
\end{aligned}
$$

Extended logic

$$
p, q ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x.\, p \mid \forall x.\, p
$$

local

shared

Assertion semantics

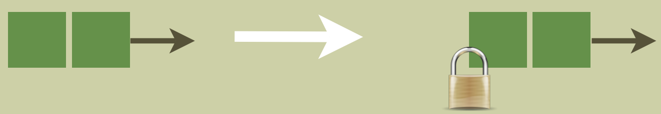
$$l, s \vDash P \iff l \vDash_{\mathsf{SL}} P$$

$$l, s \vDash \boxed{P} \iff l = \emptyset \wedge (s \vDash_{\mathsf{SL}} P)$$

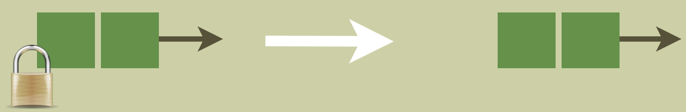
$$l, s \vDash p_1 * p_2 \iff \exists l_1, l_2.\ (l = l_1 \uplus l_2) \wedge (l_1, s \vDash p_1) \wedge (l_2, s \vDash p_2)$$

Split local state;
share global state.

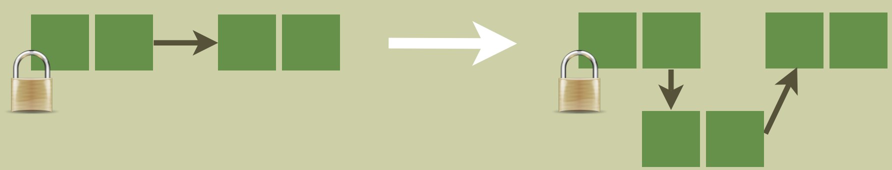
Actions



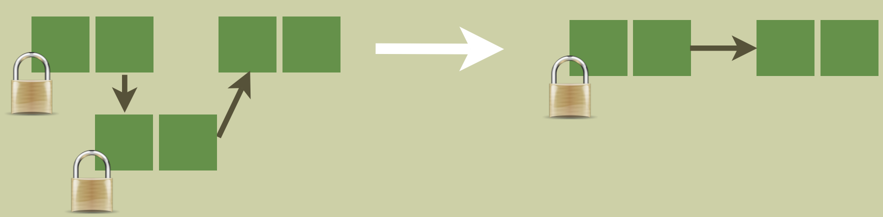
$$x \mapsto 0, v, t \quad \rightsquigarrow \quad x \mapsto tid, v, t$$



$$x \mapsto tid, v, t \quad \rightsquigarrow \quad x \mapsto 0, v, t$$



$$x \mapsto tid, v, t \quad \rightsquigarrow \quad \begin{array}{c} x \mapsto tid, v, y \\ * \, y \mapsto 0, v', t \end{array}$$

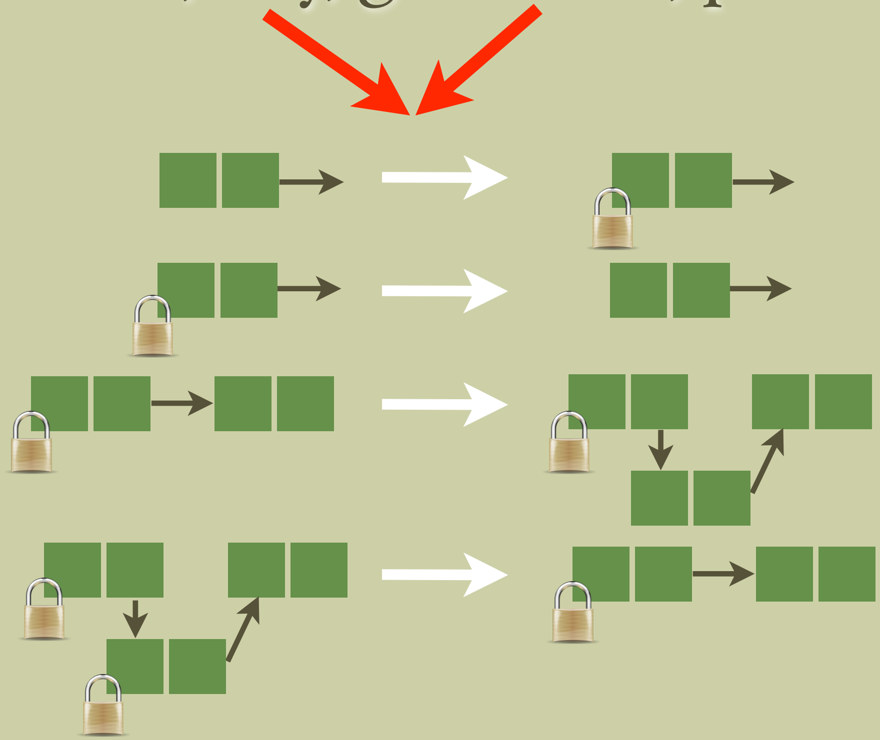


$$\begin{array}{c} x \mapsto tid, v, y \\ * \, y \mapsto tid, v', t \end{array} \quad \rightsquigarrow \quad x \mapsto tid, v, t$$

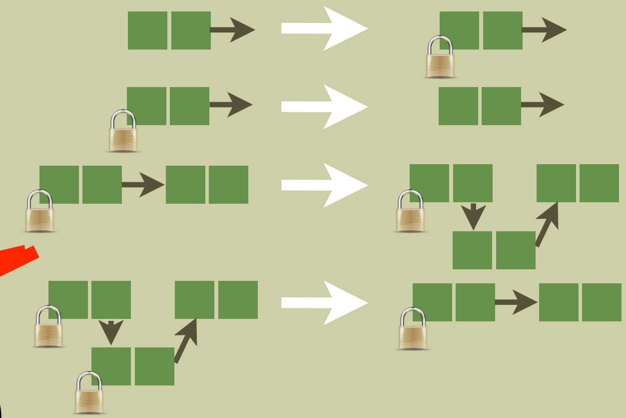
Judgements

$$\vdash C \ \mathbf{sat}\ (p, R, G, q)$$

(precondition, rely, guarantee, postcondition)



Parallel rule



$$\frac{\vdash C_1 \ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1) \qquad \vdash C_2 \ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2)}{\vdash (C_1 \| C_2) \ \mathbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)}$$

Splits local state;
Shares global state.

Atomic commands

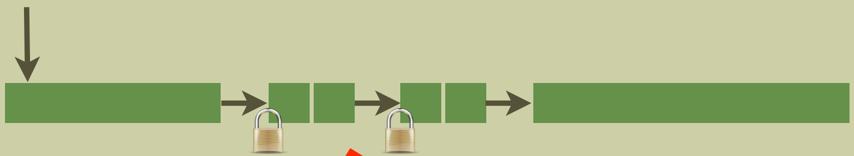
$$\frac{p, q \text{ stable under } R \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (p, \emptyset, G, q)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (p, R, G, q)}$$

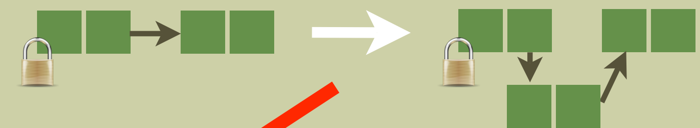
$$\frac{P_2, Q_2 \text{ precise} \qquad (P_2 \rightsquigarrow Q_2) \in G \qquad \vdash C\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ \left(P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F}\right)}$$

Local state

Shared state

Stability





$\boxed{S}$ stable under $P \rightsquigarrow Q$

iff

$$((P \mathrel{-\!\!\circledast} S) * Q) \;\Rightarrow\; S$$

The End

Further topics:

- Automation (SmallfootRG)
- Local guards & provided clauses
- Modular reasoning about memory allocators
- Proving linearisability of concurrent algorithms