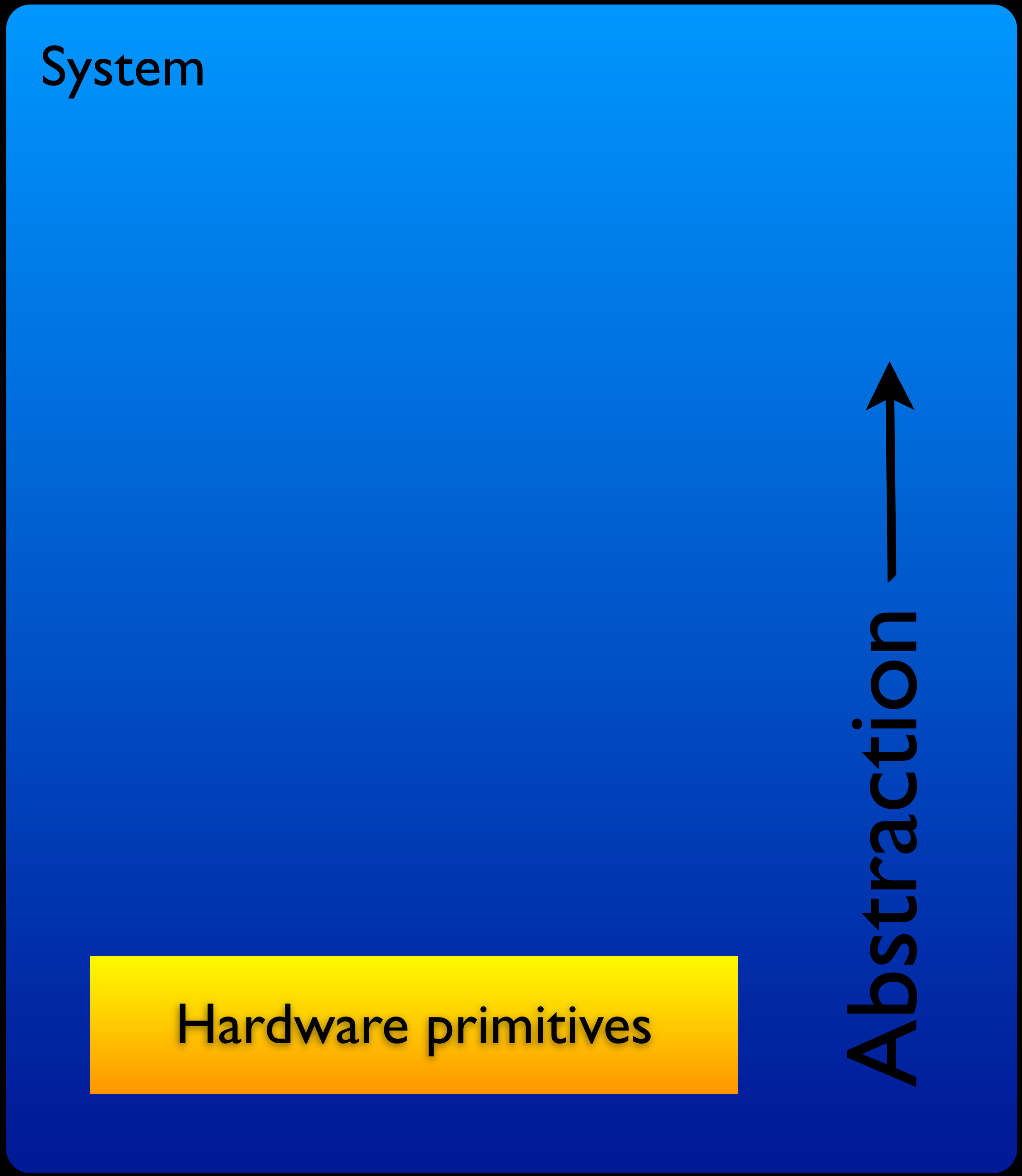
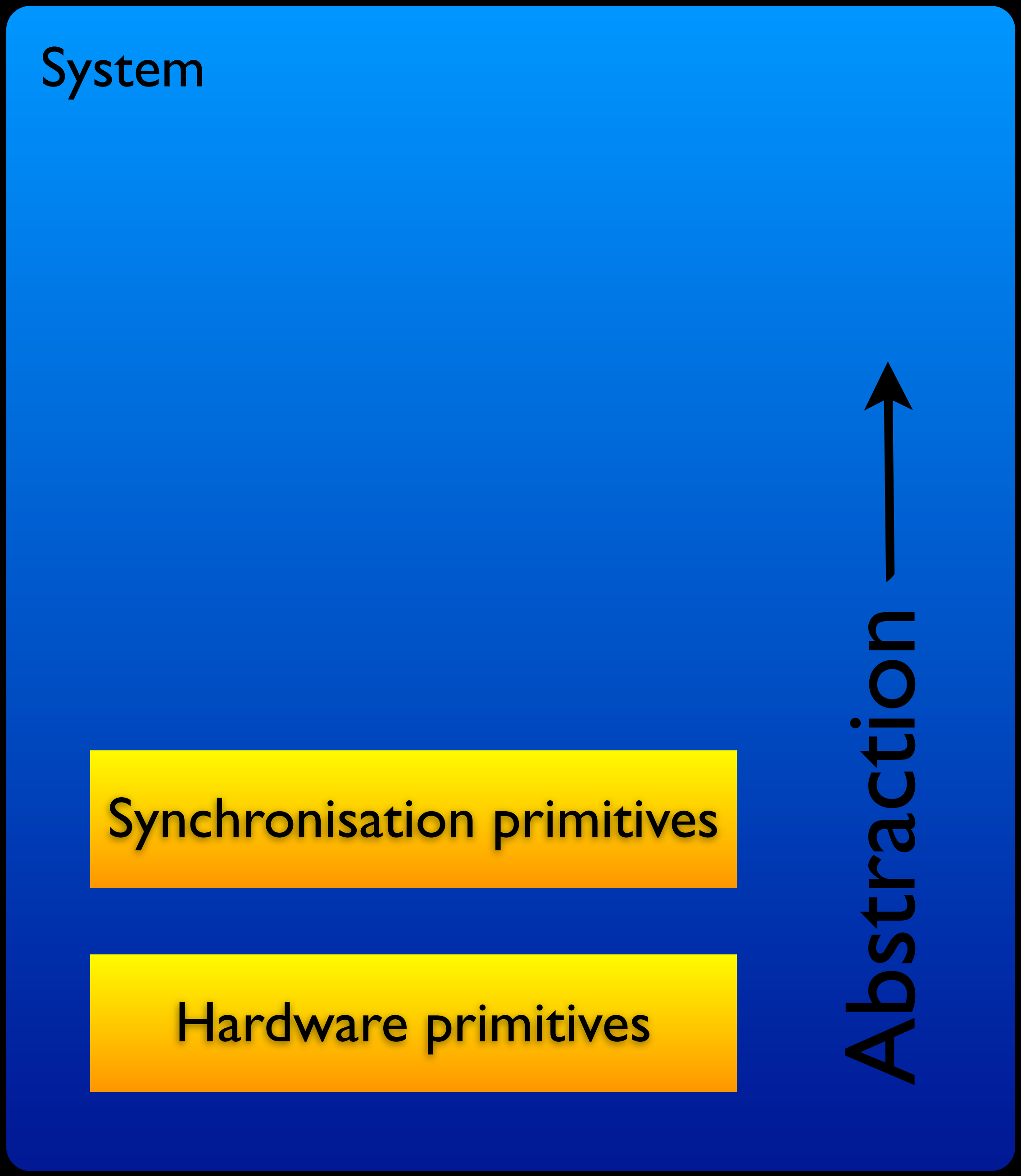
Providing a Fiction of Disjoint Concurrency

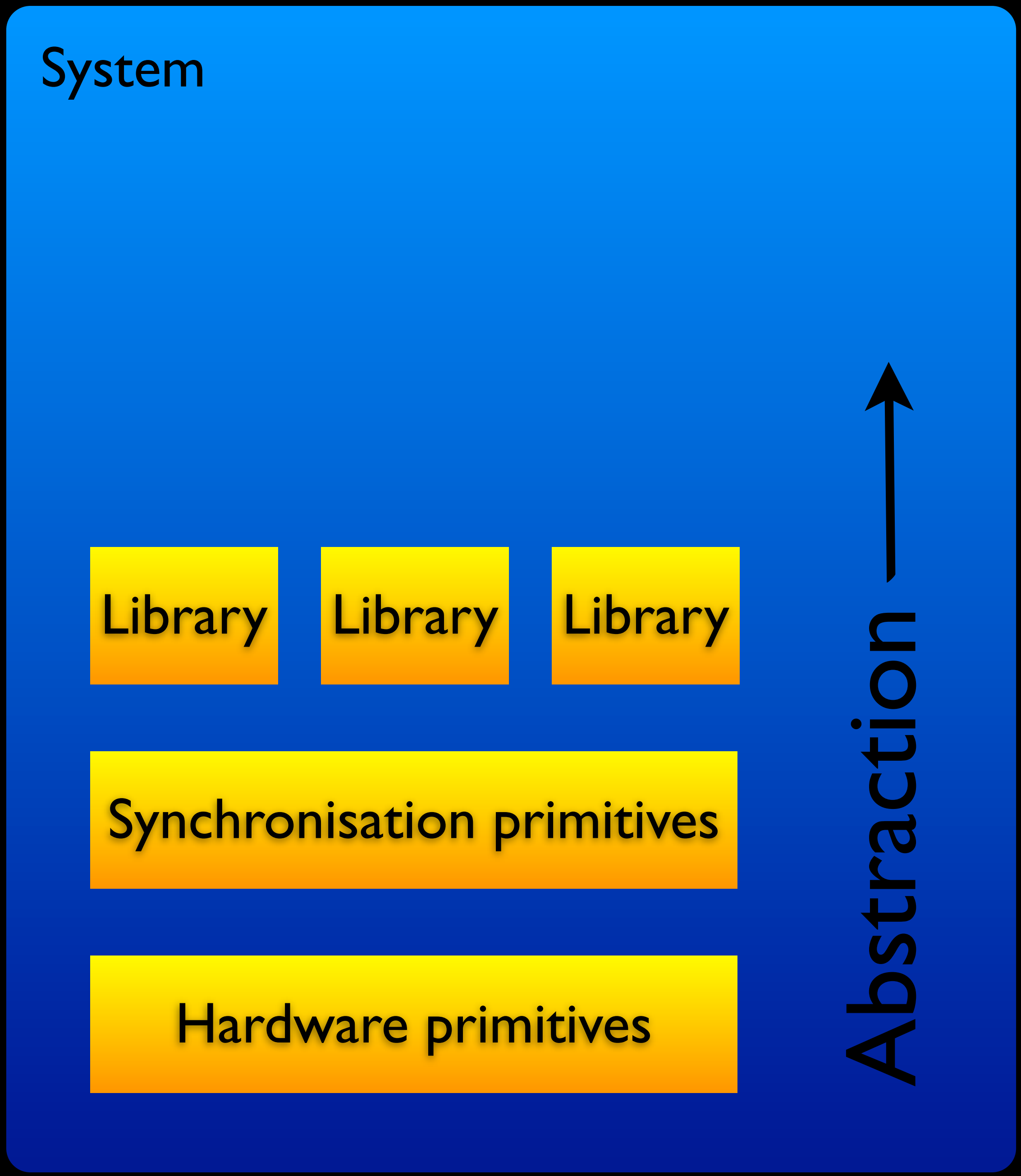
Thomas Dinsdale-Young, **Mike Dodds**,
Philippa Gardner, Matthew Parkinson

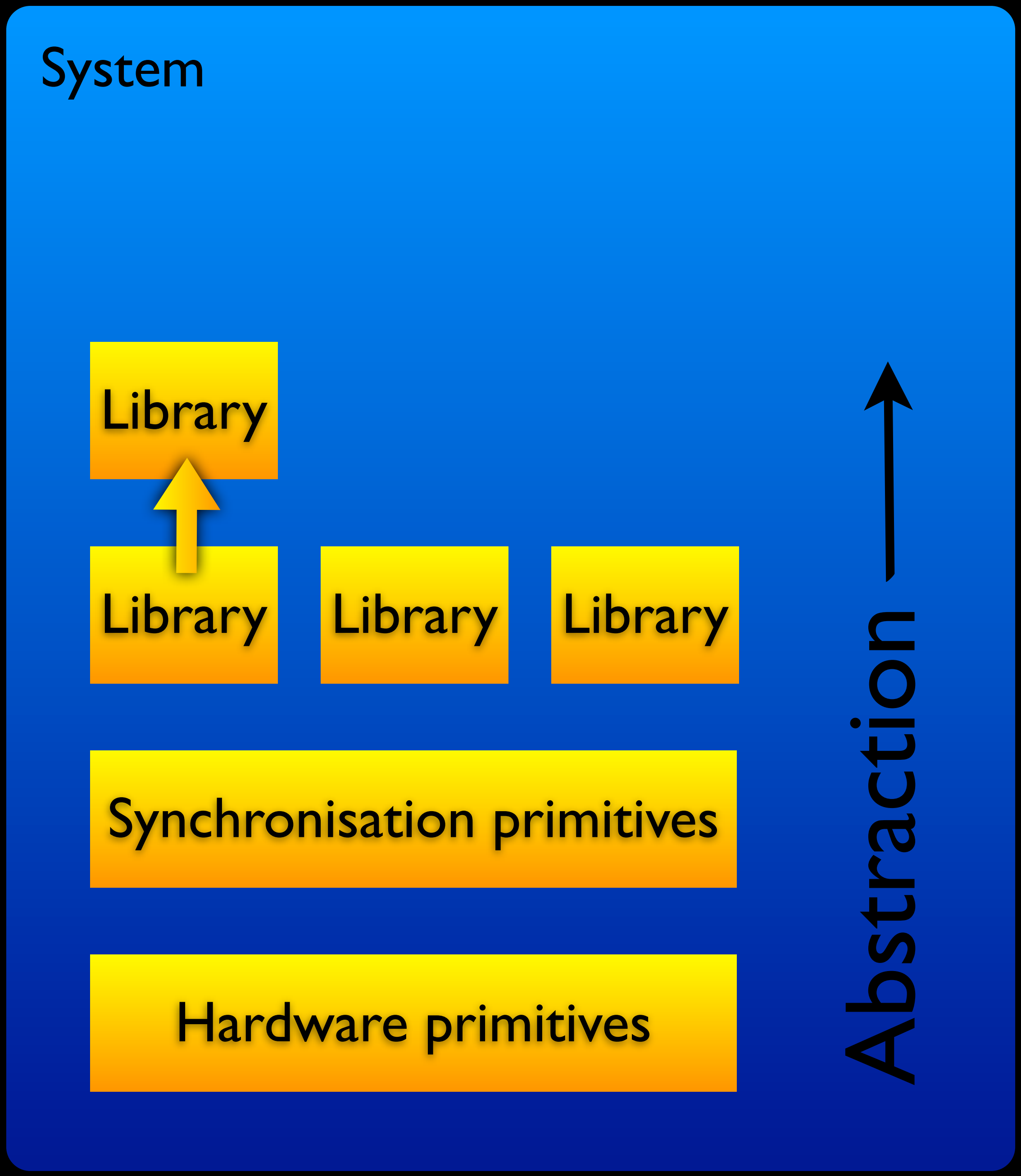
Summary

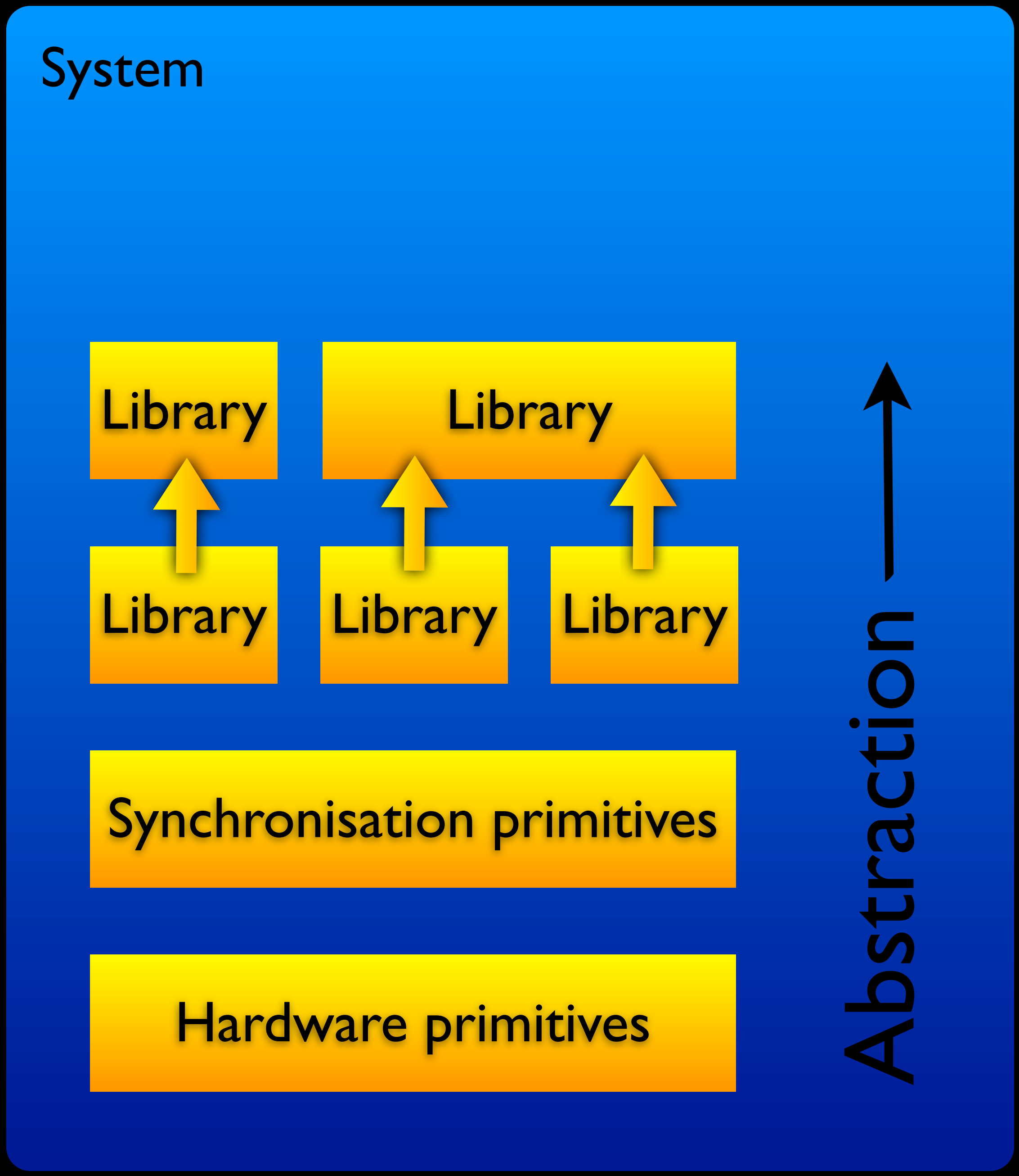
- Many concurrent programs are non-disjoint, but disjointness is very useful.
- We give disjoint specifications for non-disjoint algorithms, presenting a *fiction of disjointness*.
- Disjoint specifications for modules can be composed to give disjoint specifications for clients.
- In this way, we allow abstract reasoning and information hiding.

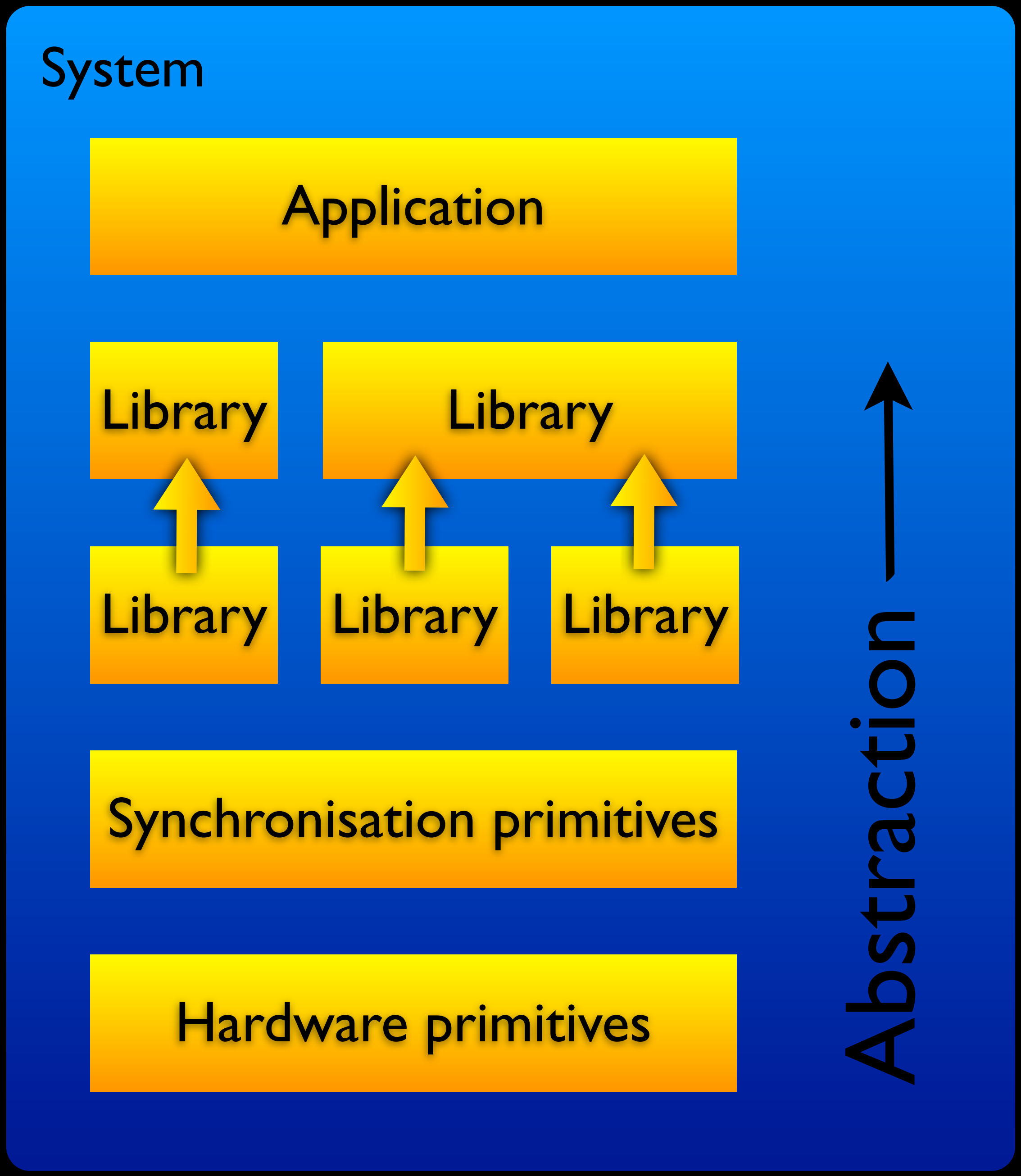

System
Hardware primitives
Abstraction

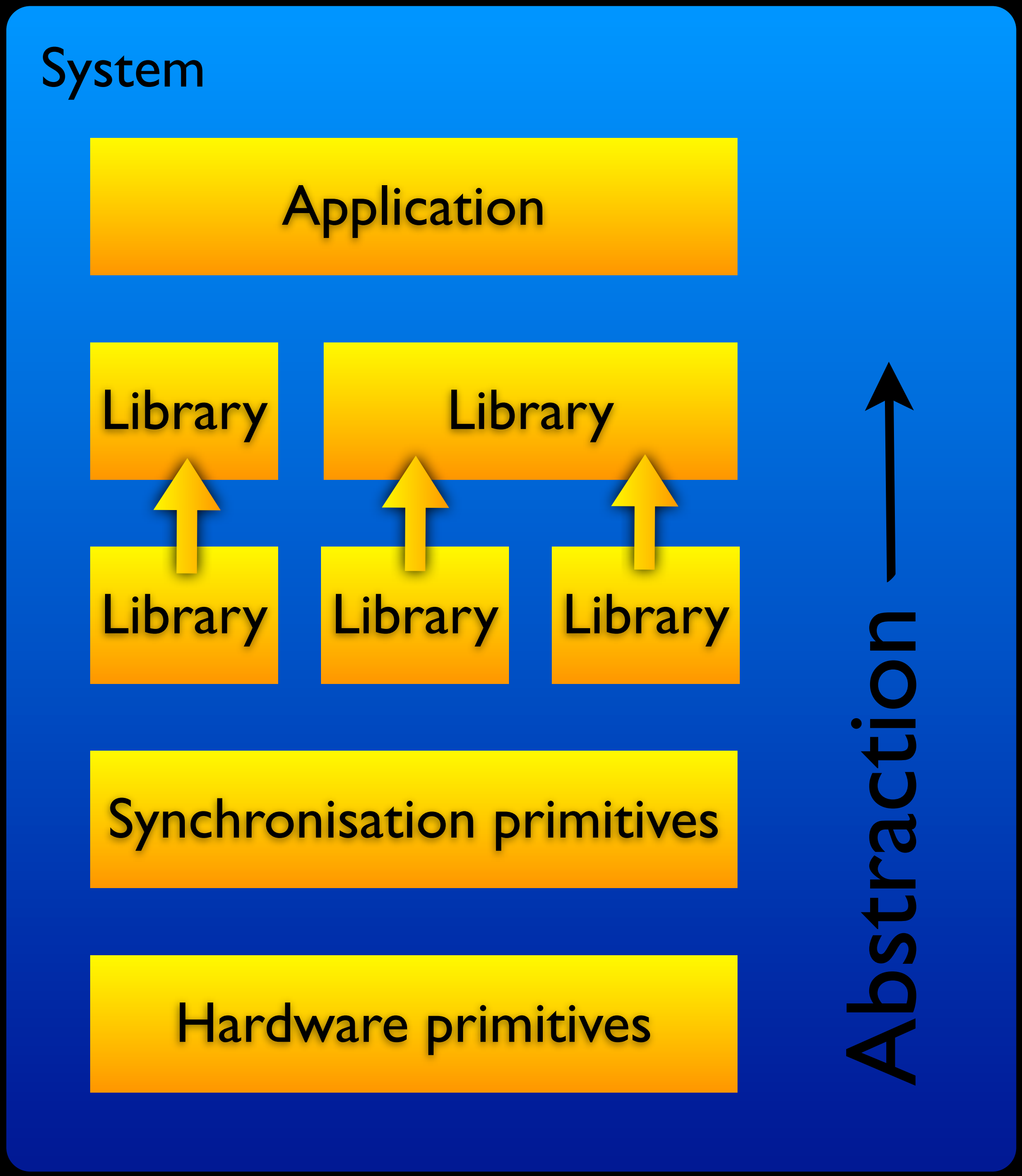

System
Synchronisation primitives
Hardware primitives
Abstraction


System
Library
Library
Library
Synchronisation primitives
Hardware primitives
Abstraction


System
Library
Library
Library
Library
Synchronisation primitives
Hardware primitives
Abstraction


System
Library
Library
Library
Library
Library
Synchronisation primitives
Hardware primitives
Abstraction


System
Application
Library
Library
Library
Library
Library
Synchronisation primitives
Hardware primitives
Abstraction



In reasoning about this system we'd like:

- Encapsulation: each module should be verified independent of other running modules.
- Abstract reasoning: each module should only reason in terms of of the preceding layer.
- Abstract specification: each module should present a specification that applies to any similar module.

Solution: disjointness.

Disjointness gives encapsulation and abstraction for free.

> Disjoint modules are naturally insulated from each other, giving encapsulation.
>
> Consequently they can be represented abstractly without worrying about overlapping properties.

Disjointness in separation logic

Separation logic requires that resources are disjoint.

Disjointness is expressed by a star operator.

P1 * (P2 * P3)

Disjointness in separation logic

Separation logic requires that resources are disjoint.

Disjointness is expressed by a star operator.

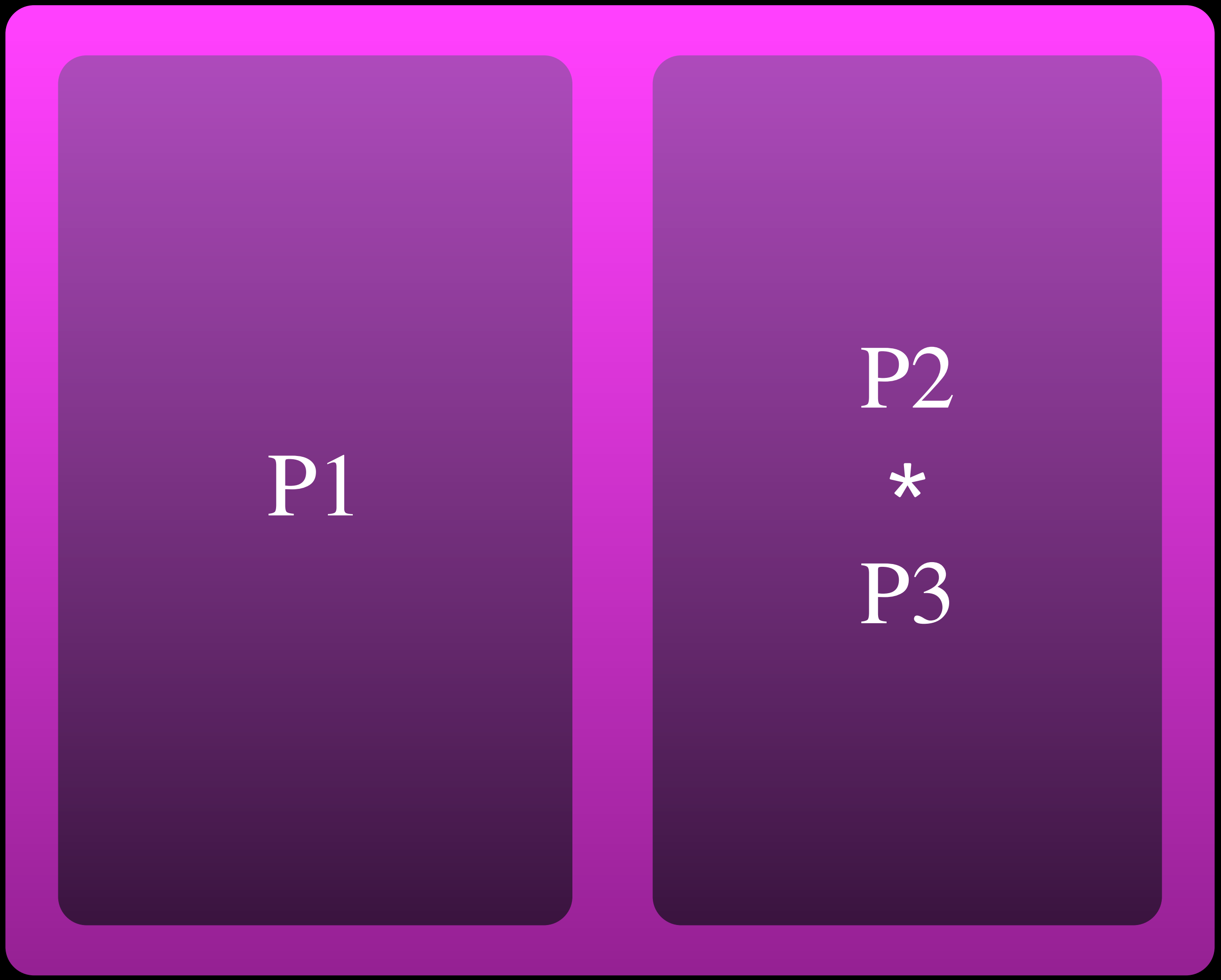
P1 * (P2 * P3)

The disjoint star

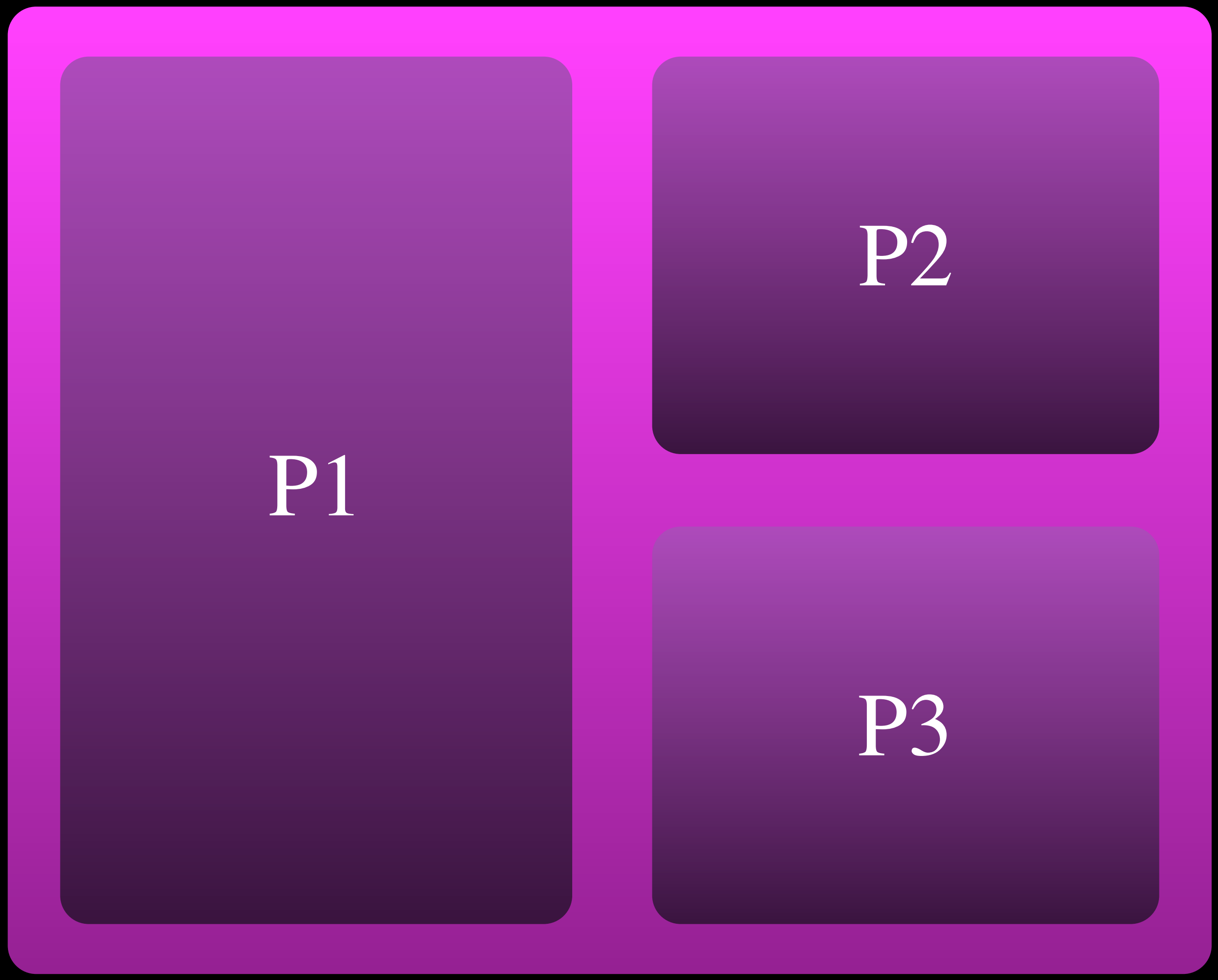
Disjointness in separation logic

P1 * (P2 * P3)

Disjointness in separation logic



Disjointness in separation logic



Disjointness and concurrency

Concurrent programs running disjointly can be reasoned about separately.

$$\frac{\vdash \{P_1\}\,C_1\,\{Q_1\} \qquad \vdash \{P_2\}\,C_2\,\{Q_2\}}{\vdash \{P_1 * P_2\}\,C_1 \| C_2\,\{Q_1 * Q_2\}}$$

Resources can be transferred through invariants.

Disjointness and concurrency

Concurrent programs running disjointly can be reasoned about separately.

$$\frac{\vdash \{P_1\}\, C_1\, \{Q_1\} \qquad \vdash \{P_2\}\, C_2\, \{Q_2\}}{\vdash \{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}}$$

Resources can be transferred through invariants.

However, many concurrent algorithms share state.

Even the humble lock breaks disjointness

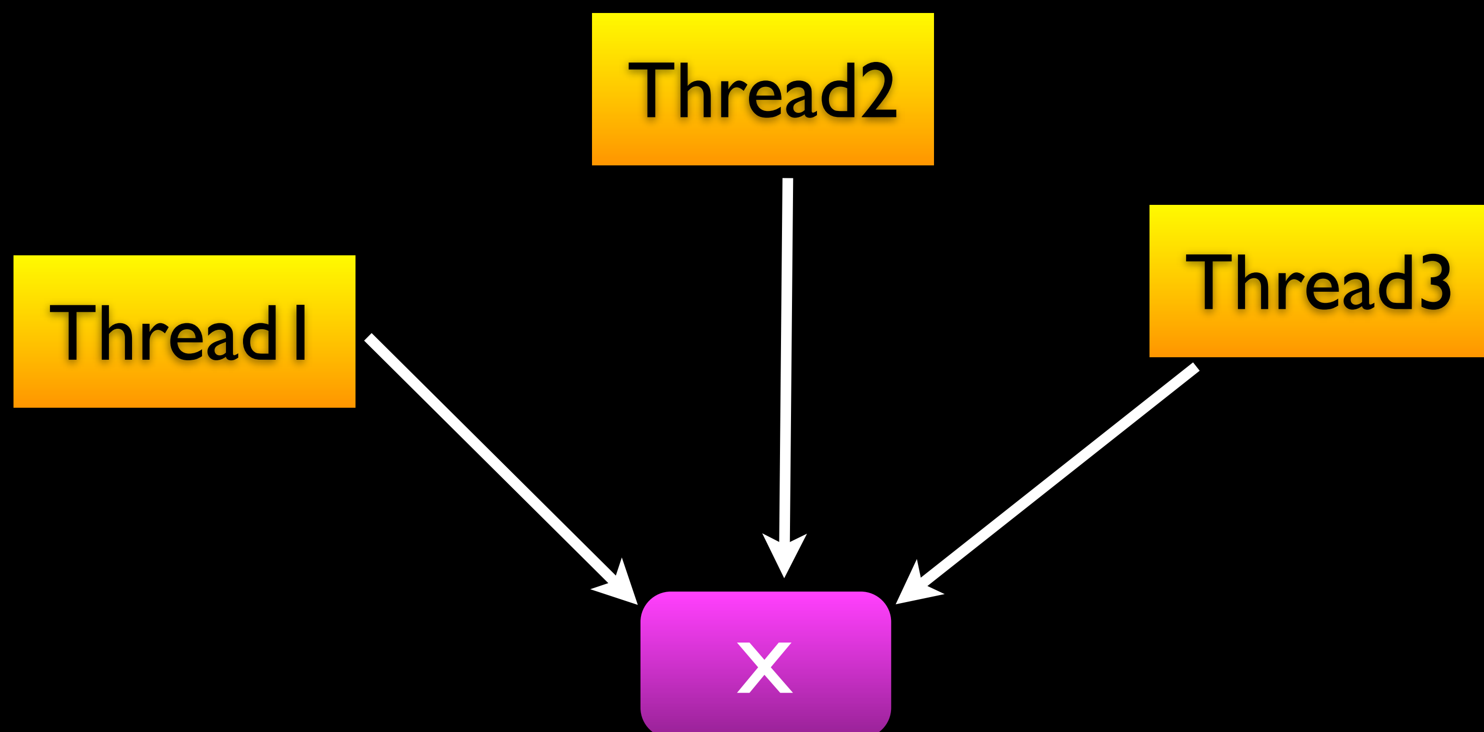
```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```

Even the humble lock breaks disjointness

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```



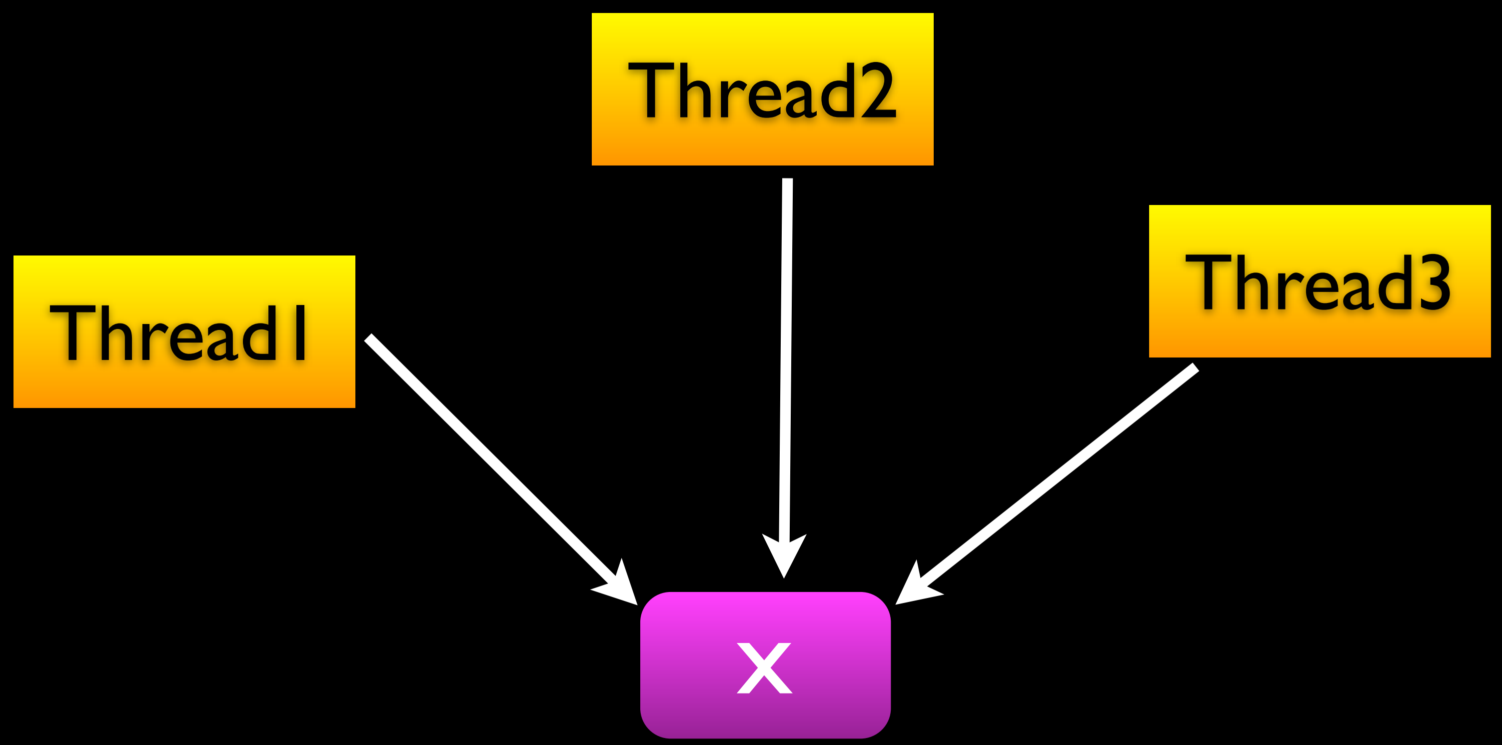
Previous solutions

- Share resources as read-only areas. Still rules out a large number of algorithms.
- Share through critical regions. Works for many examples, but becomes very complex for large examples.
- Rely-guarantee: model interference explicitly. No information hiding or abstraction

However, a lock presents a *high-level* disjointness

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

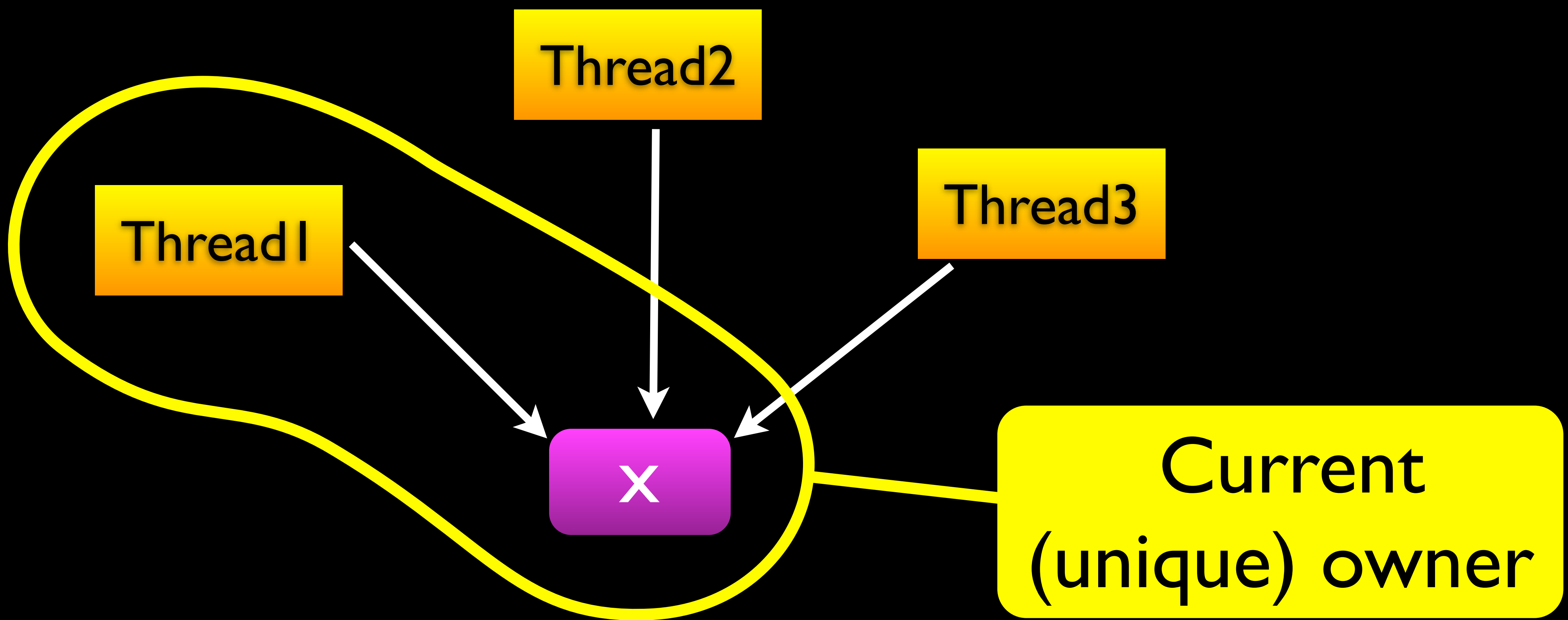
```
unlock(x) {
  x = 0;
}
```



However, a lock presents a *high-level* disjointness

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

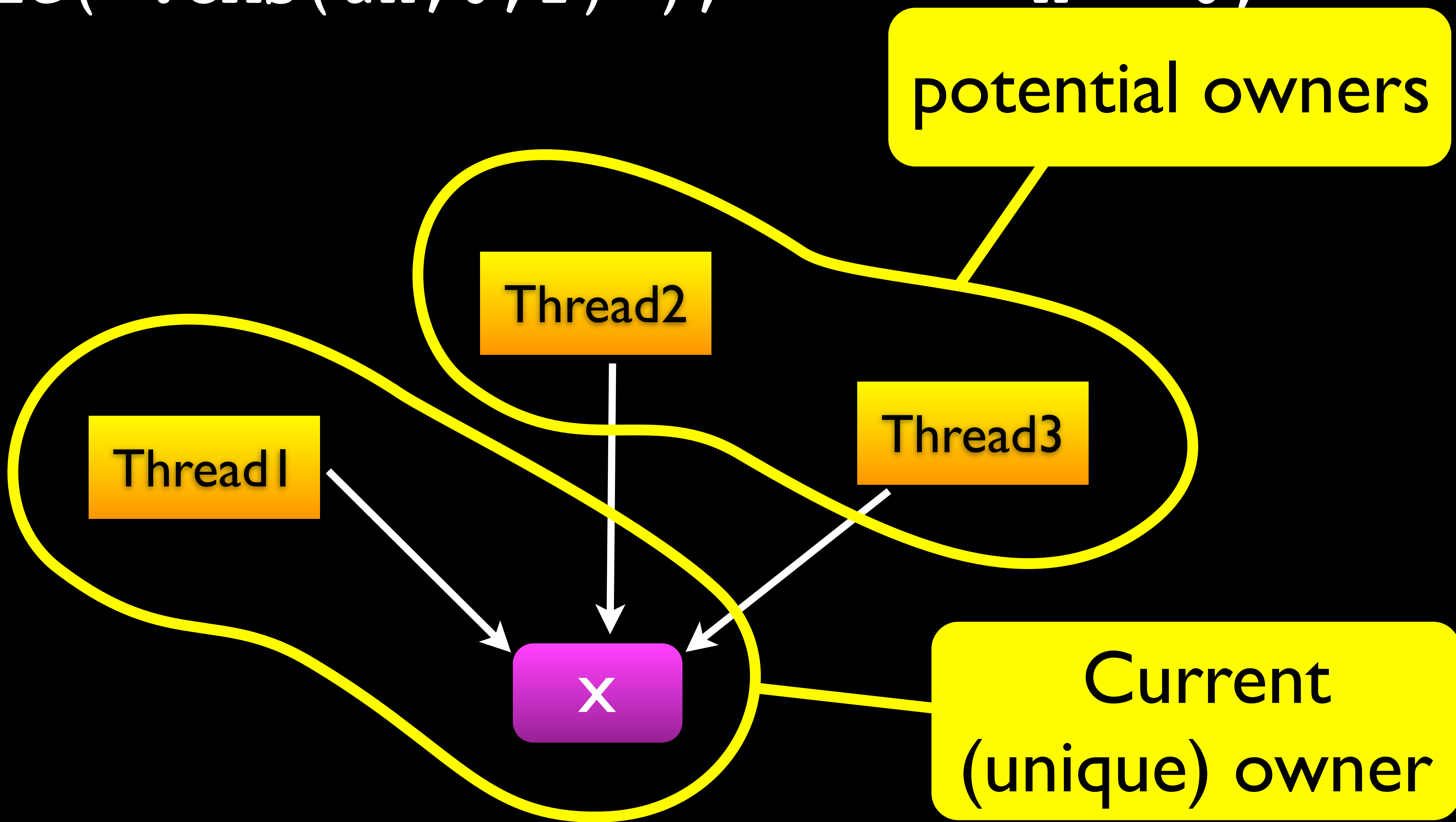
```
unlock(x) {
  x = 0;
}
```



However, a lock presents a *high-level* disjointness

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
```



Lock Specification

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```

We want the following abstract specifications to hold:

{ isLock(x) } `lock(x)` { isLock(x) * Locked(x) }

{ Locked(x) } `unlock(x)` { emp }

Lock Specification

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```

We want the following abstract specifications to hold:

{ isLock(x) } `lock(x)` { isLock(x) * Locked(x) }

{ Locked(x) } `unlock(x)` { emp }

High-level disjoint star

Lock Specification

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```

We want the following abstract specifications to hold:

{ isLock(x) } `lock(x)` { isLock(x) * Locked(x) }

{ Locked(x) } `unlock(x)` { emp }

Lock Specification

```
lock(x) {
  while( !CAS(&x,0,1) );
}
```

```
unlock(x) {
  x = 0;
}
```

We want the following abstract specifications to hold:

{ isLock(x) } `lock(x)` { isLock(x) * Locked(x) }

{ Locked(x) } `unlock(x)` { emp }

Specification is thread-centric, following the rely-guarantee approach.

The module should expose *axioms*:

$$\text{Locked}(x) * \text{Locked}(x) \iff \text{false}$$

$$\text{isLock}(x) \iff \text{isLock}(x) * \text{isLock}(x)$$

The predicates hide information, in that they can be reused without knowing how they are defined.

An abstract interface can be proved for an arbitrary module using our system.

Such abstract predicates give a *fiction of disjointness*, in that predicates can be composed as if they were disjoint.

Presenting High-level Disjointness

How do we verify that the lock implementation satisfies the high-level specification?

1. Instantiate Locked(x), isLock(x) etc. by concrete definitions;

2. prove that the definitions satisfy the required axioms;

3. prove that predicates are self-stable; and

4. prove that `lock(x)`, `unlock(x)` satisfy the required specifications under these definitions.

Divide state into shared and thread-local portions.

$$\boxed{P} * Q$$

Divide state into shared and thread-local portions.

Shared state assertion

$$\boxed{P} * Q$$

Divide state into shared and thread-local portions.

$$\boxed{P} * Q$$

Local state assertion

Divide state into shared and thread-local portions.

$$\boxed{P} * Q$$

The star behaves additively over shared state.

$$\boxed{P} * \boxed{Q} \iff \boxed{P \land Q}$$

Shared state is subject to *interference*, modelling changes from other threads.

Permission assertions control interference.

$$[\,\textsc{Action}(\ldots)\,]_i$$

Permissions value i records whether a state update is permitted to the current thread or other threads.

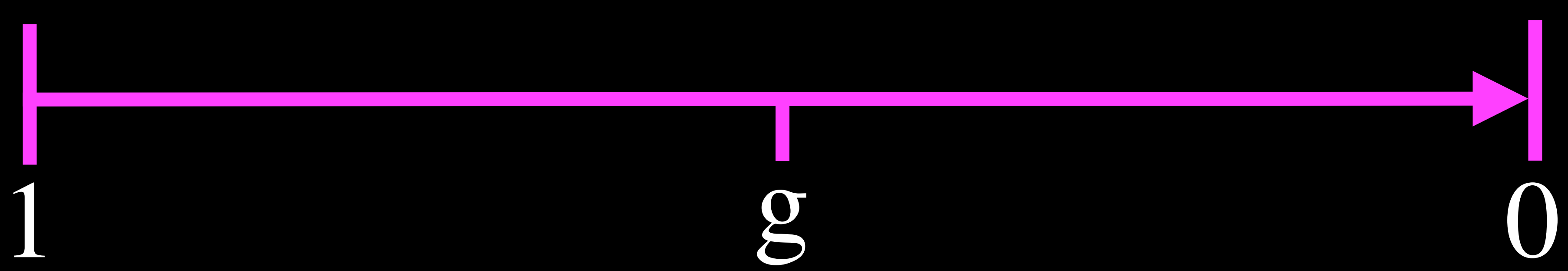
Operations in the program must be permitted by the permissions held in local state.

Our slogan: "Actions are resources"

Permission assertions control interference.

$$[\,\text{ACTION}(\ldots)\,]_i$$

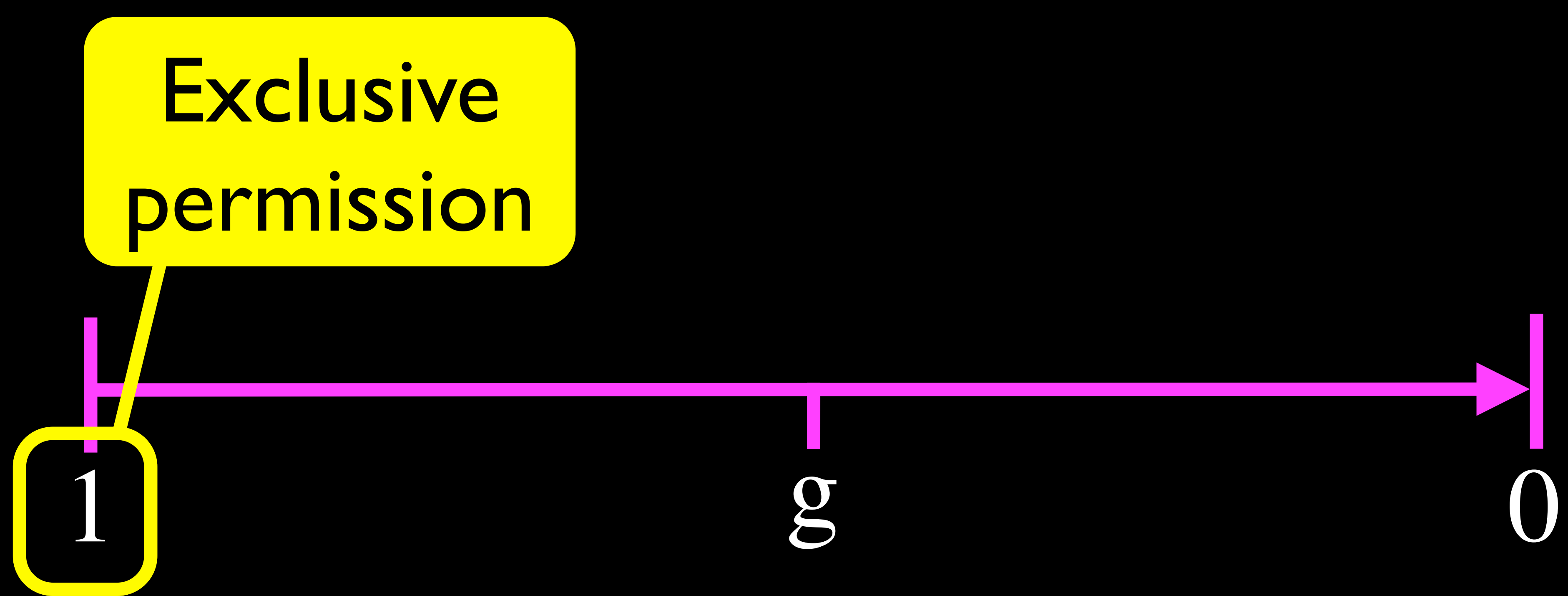
Permissions value i records whether a state update is permitted to the current thread or other threads.



Permission assertions control interference.

$$[\,\textsc{action}(\ldots)\,]_i$$

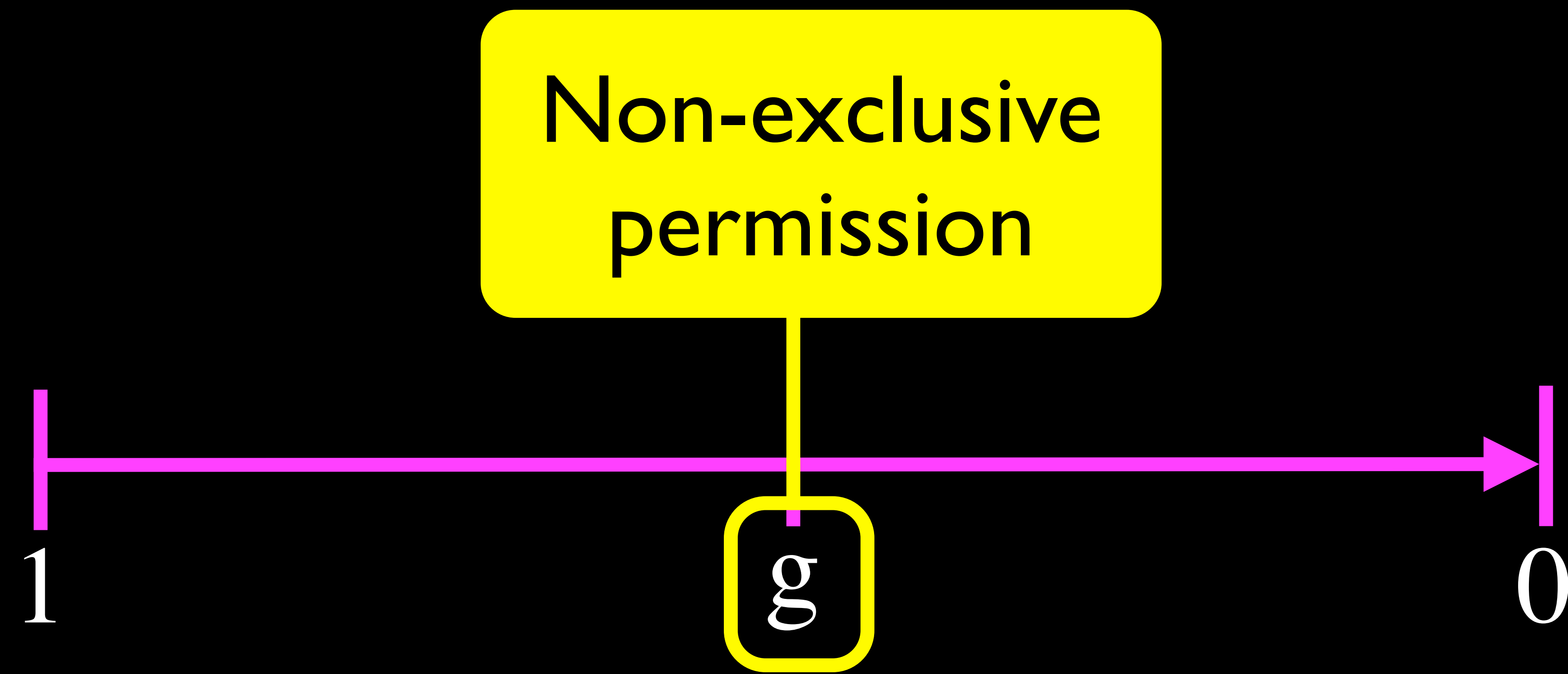
Permissions value i records whether a state update is permitted to the current thread or other threads.



Permission assertions control interference.

$$[\,\textsc{Action}(\ldots)\,]_i$$

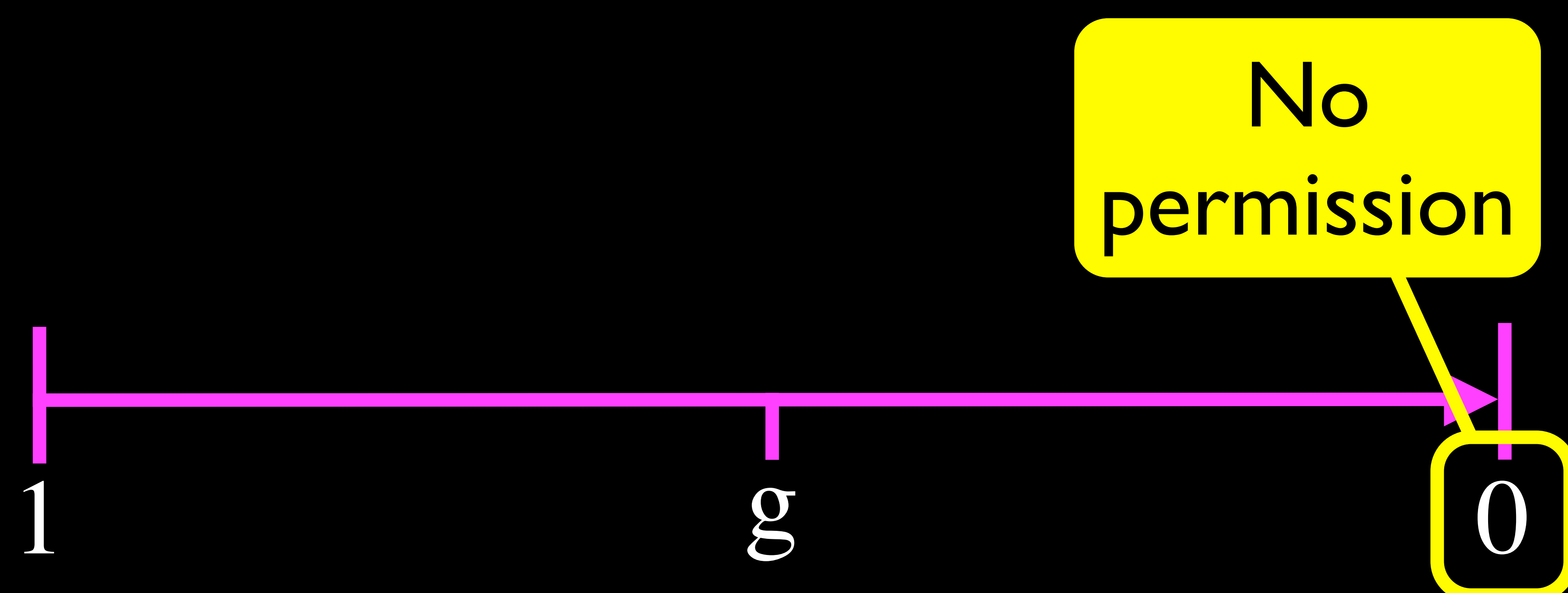
Permissions value i records whether a state update is permitted to the current thread or other threads.

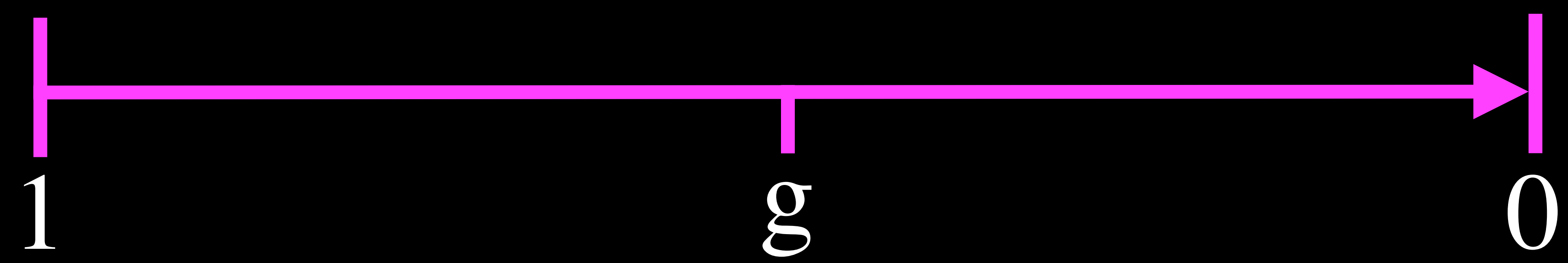


Permission assertions control interference.

$$[\,\textsc{Action}(\ldots)\,]_i$$

Permissions value i records whether a state update is permitted to the current thread or other threads.





Can weaken permissions down this axis:

$$[\,\textsc{Act}(\ldots)\,]_1 \implies [\,\textsc{Act}(\ldots)\,]_g$$

$$[\,\textsc{Act}(\ldots)\,]_g \implies [\,\textsc{Act}(\ldots)\,]_g * [\,\textsc{Act}(\ldots)\,]_g$$

$$[\,\textsc{Act}(\ldots)\,]_g \implies [\,\textsc{Act}(\ldots)\,]_0$$

Lock permissions

$$\textsc{unlock}(x): \quad x \mapsto 1 \;\leadsto\; x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}}$$

Lock permissions

$$\text{UNLOCK}(x): \quad x \mapsto 1 \;\rightsquigarrow\; x \mapsto 0 * [\text{UNLOCK}(x)]_{\mathbf{1}}$$

Permission on unlock is returned to shared state

Lock permissions

$$\textsc{unlock}(x): \quad x \mapsto 1 \;\rightsquigarrow\; x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}}$$

$$\textsc{lock}(x): \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}} \;\rightsquigarrow\; x \mapsto 1$$

Lock permissions

$$\text{UNLOCK}(x): \quad x \mapsto 1 \quad \rightsquigarrow \quad x \mapsto 0 * [\text{UNLOCK}(x)]_{\mathbf{1}}$$

$$\text{LOCK}(x): \quad x \mapsto 0 * [\text{UNLOCK}(x)]_{\mathbf{1}} \quad \rightsquigarrow \quad x \mapsto 1$$

Permission on unlock is moved to thread-local state

Lock permissions

$$\textsc{unlock}(x): \quad x \mapsto 1 \quad \leadsto \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}}$$

$$\textsc{lock}(x): \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}} \quad \leadsto \quad x \mapsto 1$$

Note permissions are part of state.

Interference can update permissions

Predicate definitions

$$\mathsf{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

Predicate definitions

Thread has permission to lock x

$$\mathsf{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

Predicate definitions

$$\mathsf{isLock}(x) \iff [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

x is either unlocked or locked in the shared state

Predicate definitions

$$\text{isLock}(x) \iff [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \text{true}}$$

remaining locks in the shared area

Predicate definitions

$$\mathsf{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Predicate definitions

$$\text{isLock}(x) \iff [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \text{true}}$$

Thread has permission to unlock x

$$\text{Locked}(x) \iff [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \text{true}}$$

Predicate definitions

$$\mathsf{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

x is locked in the shared state

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

We also define a constructor action:

$$\textsc{CreateLock}: \quad [\textsc{lock}(x)]_g \;\rightsquigarrow\; x \mapsto 0$$

Use this to define a lock constructor predicate:

$$\text{LockFactory}() \iff [\textsc{CreateLock}]_g * \boxed{\circledast_x. \begin{array}{c}(x \mapsto 0 * [\textsc{unlock}(x)]_1 \vee x \mapsto 1) \\ \vee\; ([\textsc{unlock}(x)]_1 * [\textsc{lock}(x)]_g)\end{array}}$$

All locations are either locked, unlocked, or uninitialized

Showing conformance to specification

1. definitions satisfy axioms

Showing conformance to specification

1. definitions satisfy axioms

$$\text{Locked}(x) * \text{Locked}(x) \iff \text{false}$$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{Locked(x)} * \text{Locked(x)} \iff \text{false}$$

Recall: $\text{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \text{true}}$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{Locked(x)} * \text{Locked(x)} \iff \text{false}$$

Recall: $\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$

As a consequence of permission semantics:

$$[\textsc{Unlock}(x)]_1 * [\textsc{Unlock}(x)]_1 \implies \mathsf{false}$$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{Locked(x)} * \text{Locked(x)} \iff \text{false}$$

Recall: $\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$

As a consequence of permission semantics:

$$[\textsc{Unlock}(x)]_1 * [\textsc{Unlock}(x)]_1 \implies \mathsf{false}$$

□

Showing conformance to specification

1. definitions satisfy axioms

$$\text{isLock(x)} \iff \text{isLock(x) * isLock(x)}$$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{isLock(x)} \iff \text{isLock(x) * isLock(x)}$$

Recall:

$$\text{isLock}(x) \iff [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \text{true}}$$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{isLock(x)} \iff \text{isLock(x)} * \text{isLock(x)}$$

Recall:

$$\text{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

By definition:

$$\boxed{\mathrm{P}} \iff \boxed{\mathrm{P}} * \boxed{\mathrm{P}}$$

$$[\textsc{Action}]_g \iff [\textsc{Action}]_g * [\textsc{Action}]_g$$

Showing conformance to specification

1. definitions satisfy axioms

$$\text{isLock(x)} \iff \text{isLock(x)} * \text{isLock(x)}$$

Recall:

$$\text{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \text{true}}$$

By definition:

$$\boxed{\mathrm{P}} \iff \boxed{\mathrm{P}} * \boxed{\mathrm{P}}$$

$$[\textsc{Action}]_g \iff [\textsc{Action}]_g * [\textsc{Action}]_g$$

□

Showing conformance to specification

1. definitions satisfy axioms

Other axioms are similar.

Showing conformance to specification

2. predicates are self-stable

For the fiction of disjointness, we want predicates that can be used without knowing their internal structure.

This means they must be *self-stable*: invariant under interference as a result of their contained permissions.

permissions record possible interference from other threads.

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

First possible action: $\textsc{lock}(x)$

$$\textsc{lock}(x): \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}} \quad \rightsquigarrow \quad x \mapsto 1$$

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

First possible action: $\textsc{lock}(x)$

$$\textsc{lock}(x): \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}} \rightsquigarrow x \mapsto 1$$

Stable under $\textsc{lock}(x)$ because the action can only fire if $x \mapsto 0$ in the shared state.

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Second possible action: $\textsc{unlock}(x)$

$$\textsc{unlock}(x): \quad x \mapsto 1 \quad \leadsto \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}}$$

Showing conformance to specification

2. predicates are self-stable

$$\mathsf{Locked}(x) \iff [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Second possible action: $\textsc{unlock}(x)$

$$\textsc{unlock}(x)\colon \quad x \mapsto 1 \quad \rightsquigarrow \quad x \mapsto 0 * [\textsc{unlock}(x)]_{\mathbf{1}}$$

Stable under $\textsc{unlock}(x)$ because Locked(x) includes full permission on it.

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\,\mathsf{isLock}(x)\,\}\quad \texttt{lock(x)}\ \{\,\mathsf{isLock}(x) * \mathsf{Locked}(x)\,\}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$\{\, \mathsf{isLock}(x) \,\}$

```
lock(x) {

    while( !CAS(&x,0,1) );

}
```

$\{\, \mathsf{isLock}(x) * \mathsf{Locked}(x) \,\}$

Showing conformance to specification

3. definitions satisfy abstract specifications

$\{ \mathsf{isLock}(x) \}$

```
lock(x) {
```

$\{ [\textsc{Lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}} \}$

```
    while( !CAS(&x,0,1) );


}
```

$\{ \mathsf{isLock}(x) * \mathsf{Locked}(x) \}$

Showing conformance to specification

3. definitions satisfy abstract specifications

$\{\ \mathsf{isLock}(x)\ \}$

```
lock(x) {
```

$\{\ [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}\ \}$

```
    while( !CAS(&x,0,1) );
```

$\{\ [\text{LOCK}(x)]_g * [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$

```
}
```

$\{\ \mathsf{isLock}(x) * \mathsf{Locked}(x)\ \}$

Showing conformance to specification

3. definitions satisfy abstract specifications

$\{\, \mathsf{isLock}(x) \,\}$

```
lock(x) {
```

$\{\ [\textsc{Lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{Unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}\ \}$

```
while( !CAS(&x,0,1) );
```

$\{\ [\textsc{Lock}(x)]_g * [\textsc{Unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$

```
}
```

$\{\, \mathsf{isLock}(x) * \mathsf{Locked}(x) \,\}$

Successful CAS corresponds to the lock action

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\ [\textsc{lock}(x)]_g * [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$$

$$\Longrightarrow \qquad \{\ \mathsf{isLock}(x) * \mathsf{Locked}(x)\ \}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\ [\textsc{lock}(x)]_g * [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$$

$$\Longrightarrow \qquad \{\ \mathsf{isLock}(x) * \underline{\mathsf{Locked}(x)}\ \}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\ [\text{LOCK}(x)]_g * [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$$

$$\implies \qquad \{\ \mathsf{isLock}(x) * \underline{\mathsf{Locked}(x)}\ \}$$

recall:

$$\mathsf{Locked}(x) \iff [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\ [\text{LOCK}(x)]_g * [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$$

$$\Longrightarrow \qquad \{\ \mathsf{isLock}(x) * \mathsf{Locked}(x)\ \}$$

recall:

$$\mathsf{Locked}(x) \iff [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{\ [\text{LOCK}(x)]_g * [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}}\ \}$$

$$\Longrightarrow \qquad \{\ \underline{\mathsf{isLock}(x)} * \mathsf{Locked}(x)\ \}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{ \ [\textsc{lock}(x)]_g * [\textsc{unlock}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}} \ \}$$

$$\Longrightarrow \qquad \{ \ \mathsf{isLock}(x) * \mathsf{Locked}(x) \ \}$$

recall:

$$\mathsf{isLock}(x) \iff [\textsc{lock}(x)]_g * \boxed{((x \mapsto 0 * [\textsc{unlock}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

$$\{ [\text{LOCK}(x)]_g * [\text{UNLOCK}(x)]_1 * \boxed{x \mapsto 1 * \mathsf{true}} \}$$

$$\Longrightarrow \quad \{ \mathsf{isLock}(x) * \mathsf{Locked}(x) \}$$

recall:

$$\mathsf{isLock}(x) \iff [\text{LOCK}(x)]_g * \boxed{((x \mapsto 0 * [\text{UNLOCK}(x)]_1) \vee x \mapsto 1) * \mathsf{true}}$$

Showing conformance to specification

3. definitions satisfy abstract specifications

The proof of `unlock(x)` is similar, and simpler.

Using Abstract Module Specifications

Showing conformance to specification

We have shown that the program implements the lock module interface.

Showing conformance to specification

We have shown that the program implements the lock module interface.

How do we use it?

Judgements

$$\Delta; I; \Gamma \vdash \{P\}C\{Q\}$$

Judgements

Predicate assumption, recording predicate definitions and axioms

$$\Delta; I; \Gamma \vdash \{P\}C\{Q\}$$

Judgements

Predicate assumption, recording predicate definitions and axioms

$$\Delta; I; \Gamma \vdash \{P\}C\{Q\}$$

Interference environment, recording the definitions of permissions.

Judgements

Predicate assumption, recording predicate definitions and axioms

Abstract specifications for functions used by the program

$$\Delta; I; \Gamma \vdash \{P\}C\{Q\}$$

Interference environment, recording the definitions of permissions.

Module rule

$$\frac{\Delta; I \vdash \{P_1\}C_1\{Q_1\} \quad \ldots \quad \Delta; I \vdash \{P_n\}C_n\{Q_n\} \qquad \Delta \Rightarrow \Delta' \quad \Delta'; \{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}}{\vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C \ \{Q\}}$$

Module rule

$$\frac{\Delta \Rightarrow \Delta' \quad \begin{array}{c}\Delta; I \vdash \{P_1\}C_1\{Q_1\} \quad \ldots \quad \Delta; I \vdash \{P_n\}C_n\{Q_n\} \\ \Delta'; \{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}\end{array}}{\vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C \; \{Q\}}$$

Weaken the predicate environment to hide the predicate definitions and leave only external axioms

Module rule

Module specifications must be written entirely in terms of abstract predicates, not state assertions.

$$\Delta; I \vdash \{P_1\}C_1\{Q_1\} \quad \ldots \quad \Delta; I \vdash \{P_n\}C_n\{Q_n\}$$

$$\frac{\Delta \Rightarrow \Delta' \qquad \Delta'; \{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}}{\vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C \ \{Q\}}$$

Module rule

$$\frac{\Delta \Rightarrow \Delta' \quad \Delta' ; \{P_1\} f_1 \{Q_1\}, \ldots, \{P_n\} f_n \{Q_n\} \vdash \{P\} C \{Q\} \qquad \Delta ; I \vdash \{P_1\} C_1 \{Q_1\} \quad \ldots \quad \Delta ; I \vdash \{P_n\} C_n \{Q_n\}}{\vdash \{P\} \text{ let } f_1 = C_1 \ldots f_n = C_n \text{ in } C \; \{Q\}}$$

The client specification is proved under a module specification defined entirely abstractly

Module rule

$$\frac{\Delta; I \vdash \{P_1\}C_1\{Q_1\} \quad \dots \quad \Delta; I \vdash \{P_n\}C_n\{Q_n\} \qquad \Delta \Rightarrow \Delta' \quad \Delta'; \{P_1\}f_1\{Q_1\}, \dots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}}{\vdash \{P\} \text{ let } f_1 = C_1 \dots f_n = C_n \text{ in } C \; \{Q\}}$$

We require that predicate definitions are self-stable as a side-condition

Modular rule

$$\frac{\Delta; I \vdash \{P_1\}C_1\{Q_1\} \quad \ldots \quad \Delta; I \vdash \{P_n\}C_n\{Q_n\} \qquad \Delta \Rightarrow \Delta' \quad \Delta'; \{P_1\}f_1\{Q_1\}, \ldots, \{P_n\}f_n\{Q_n\} \vdash \{P\}C\{Q\}}{\vdash \{P\}\ \mathrm{let}\ f_1 = C_1 \ldots f_n = C_n\ \mathrm{in}\ C\ \{Q\}}$$

By erasing predicate definitions and interference definitions, we make predicates purely abstract.

Modules can be used purely in terms of their abstract specs.

Parallel rule

$$\frac{\Delta, I, \Gamma \vdash \{P_1\}\, C_1\, \{Q_1\} \qquad \Delta, I, \Gamma \vdash \{P_2\}\, C_2\, \{Q_2\}}{\Delta, I, \Gamma \vdash \{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}}$$

Our major result: the parallel rule *just works*, even though we can erase predicate definitions to give abstract module specifications.

Parallel rule

$$\frac{\Delta, I, \Gamma \vdash \{P_1\}\, C_1\, \{Q_1\} \qquad \Delta, I, \Gamma \vdash \{P_2\}\, C_2\, \{Q_2\}}{\Delta, I, \Gamma \vdash \{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}}$$

High-level disjoint star

Our major resu rule *just works*, even though we can erase predicate definitions to give abstract module specifications.

Frame rule

$$\frac{\Delta, I, \Gamma \vdash \{P\}\, C\, \{Q\} \qquad \mathsf{stable}(F)}{\Delta, I, \Gamma \vdash \{P * F\}\, C\, \{Q * F\}}$$

The frame rule also just works. Consequently we can compose both in space and between threads while maintaining modularity.

Linearizability

Linearizability: every concurrent trace can be converted to an equivalent sequential trace.

Orthogonal to the fiction of disjointness: one gives fiction of disjointness in time, the other in space.

Many algorithms with disjoint specifications are also linearizable.

Conclusions

Present disjoint specifications to non-disjoint algorithms

Can layer proofs to prove complex compositions of systems.

Fiction of disjointness is a powerful notion for describing complex concurrent systems.