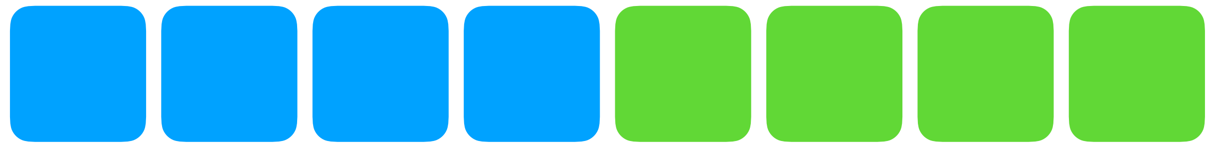
Verifying concurrent software using movers in CSPEC

Tej Chajed, Frans Kaashoek, Butler Lampson*, Nickolai Zeldovich
MIT CSAIL and *Microsoft

Concurrent software is difficult to get right

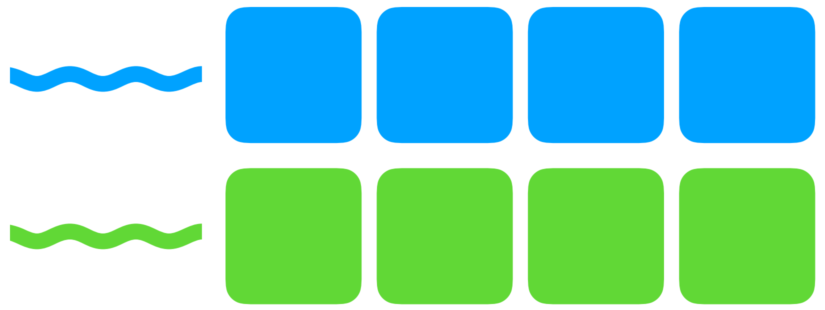
Programmer cannot reason
about code in sequence…



Concurrent software is difficult to get right

Programmer cannot reason
about code in sequence…

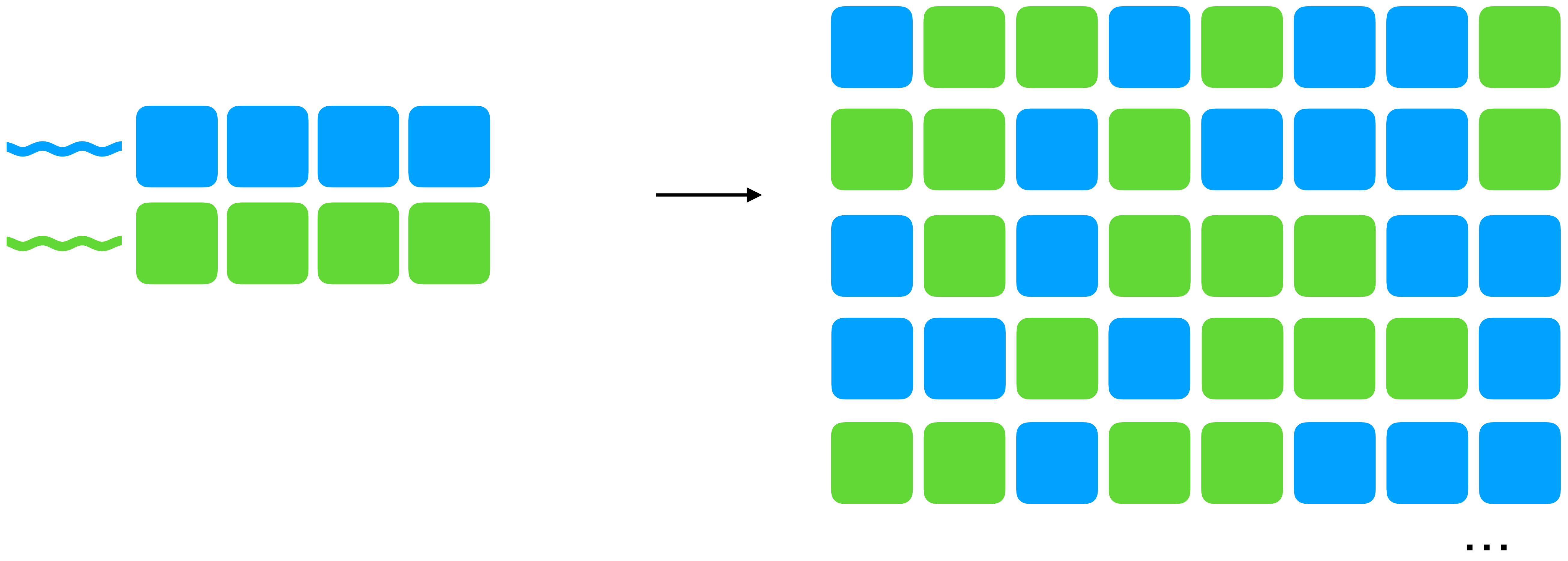
instead, must consider many executions:



Concurrent software is difficult to get right

Programmer cannot reason
about code in sequence…

instead, must consider many executions:



Goal: verify concurrent software

Challenge for formal verification

- Proofs must also cover every execution
- Many approaches to managing this complexity
    - movers [Lipton, 1975]
    - rely-guarantee [1983]
    - RGSep [CONCUR 2007]
    - FCSL [PLDI 2015]
    - Iris [POPL 2017, LICS 2018, others]
    - many others

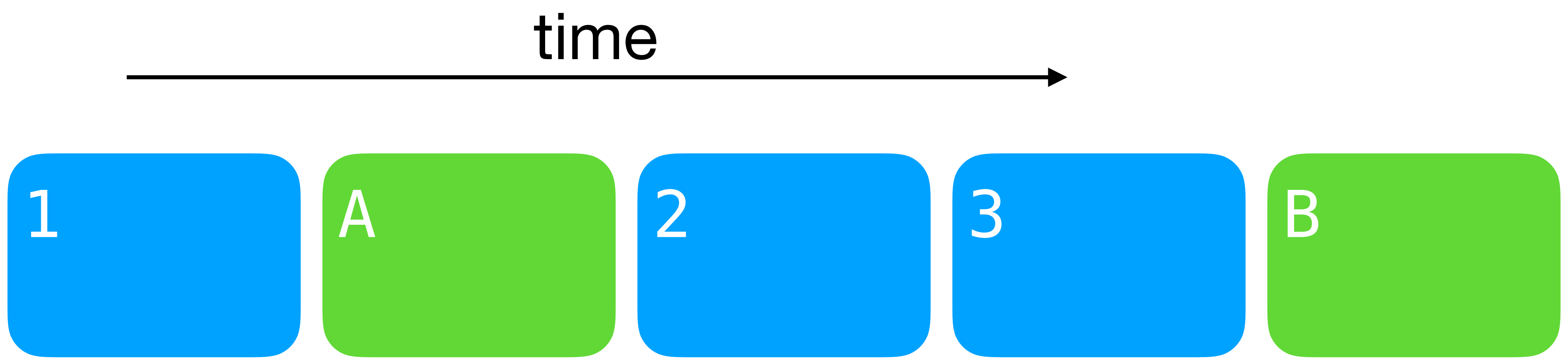
Challenge for formal verification

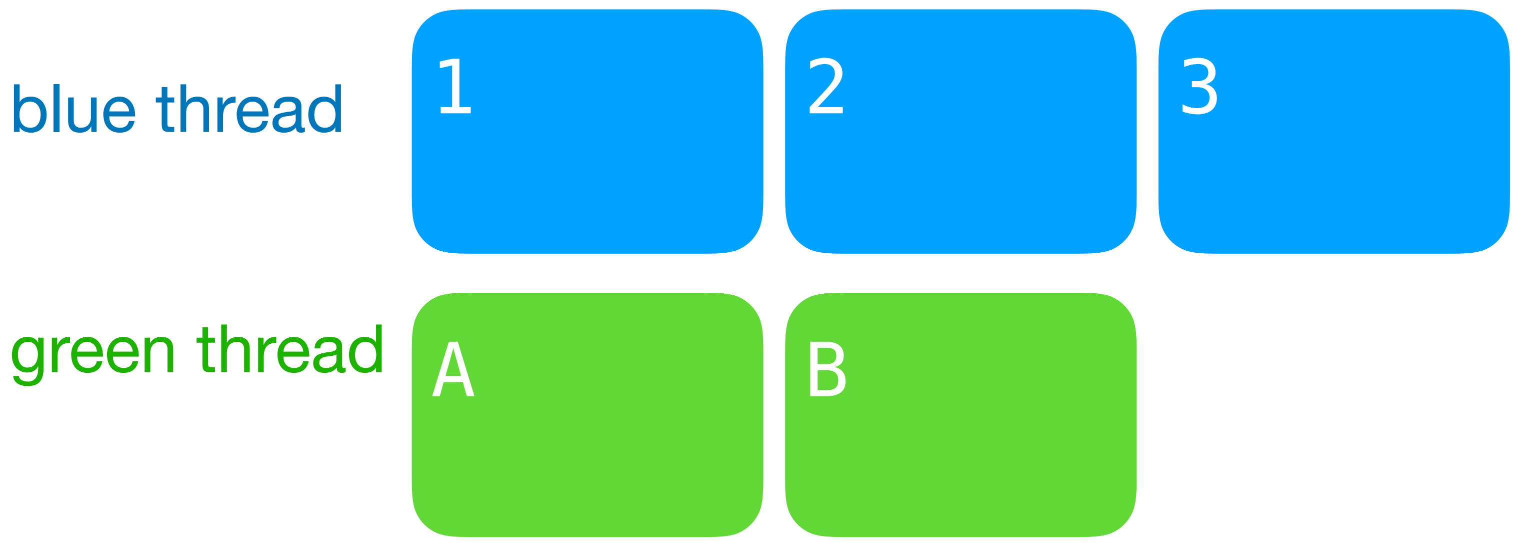
- Proofs must also cover every execution

- Many approaches to managing this complexity
  - movers [Lipton, 1975]
  - rely-guarantee [1983]
  - RGSep [CONCUR 2007]
  - FCSL [PLDI 2015]
  - Iris [POPL 2017, LICS 2018, others]
  - many others

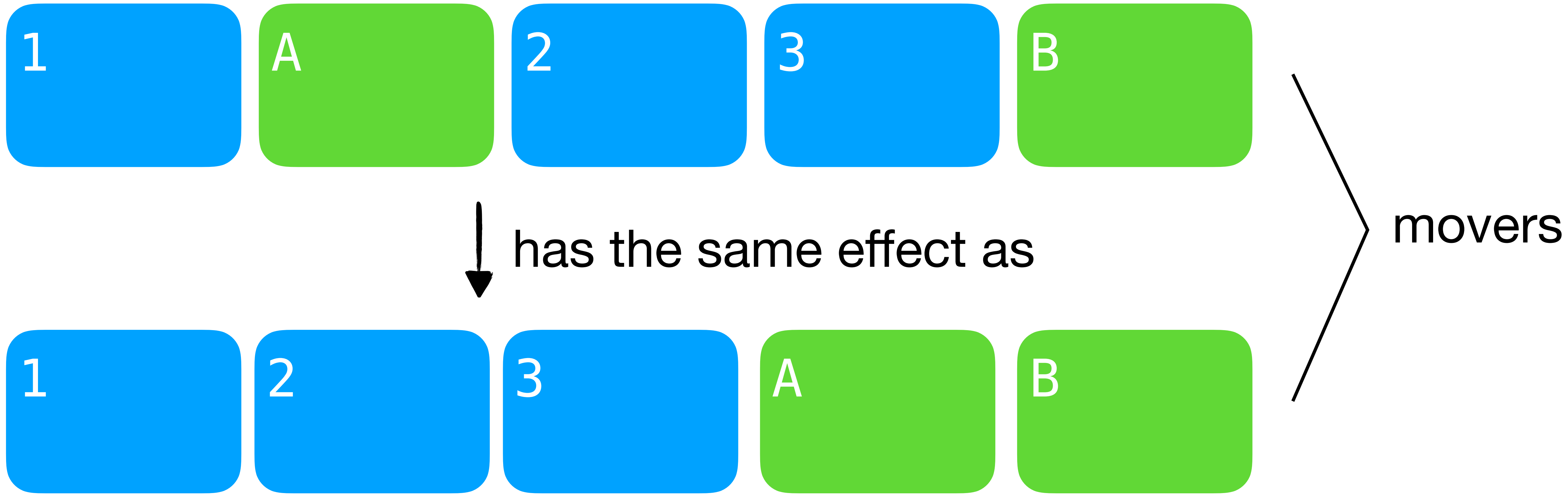
- This work: our experience using **movers**

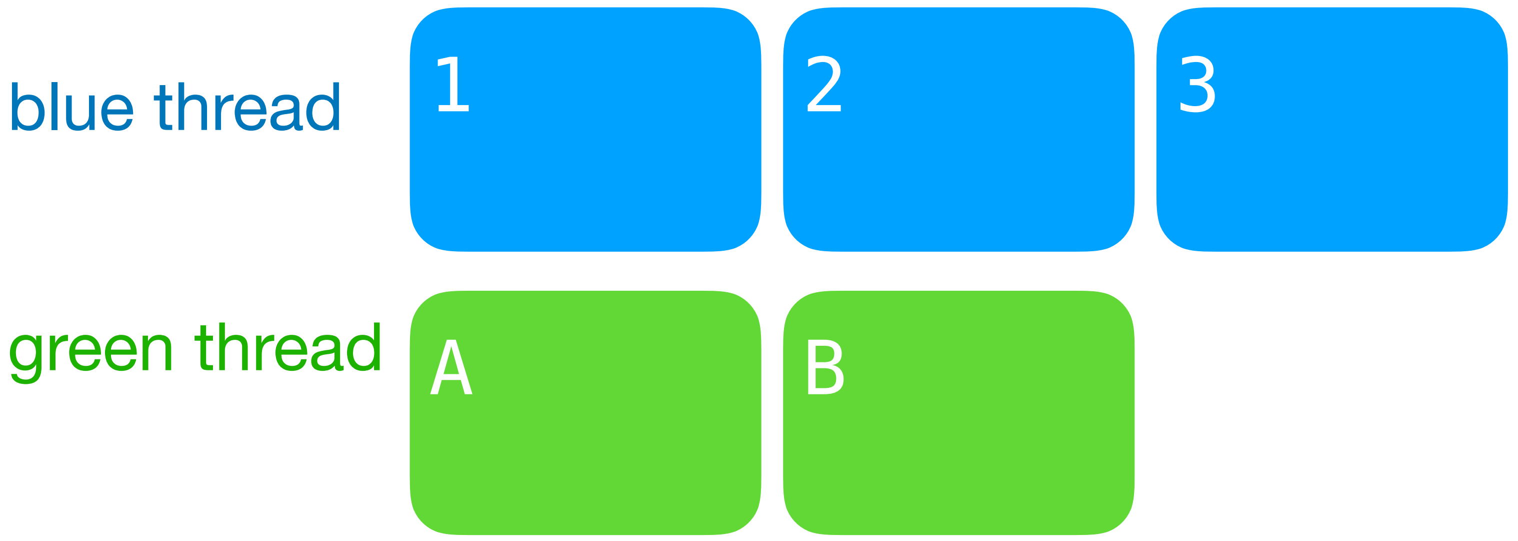
Movers: reduce concurrent executions to sequential ones



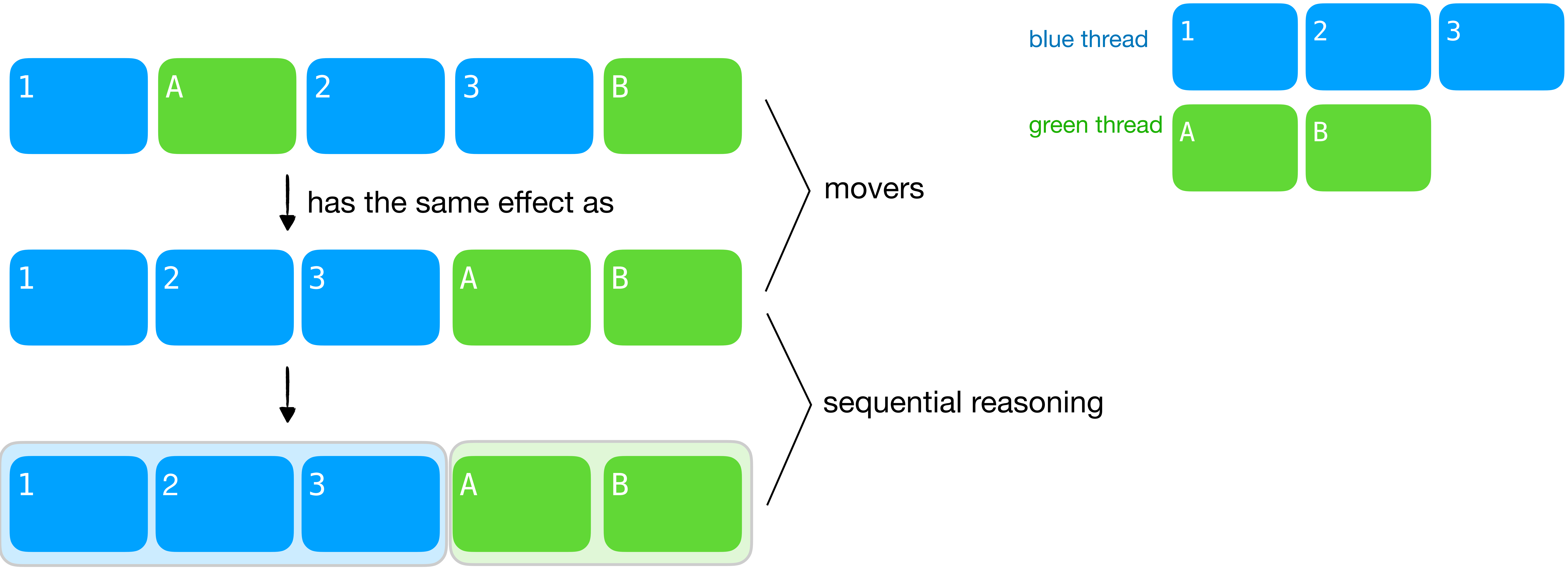


Movers: reduce concurrent executions to sequential ones





Movers: reduce concurrent executions to sequential ones



has the same effect as

movers

sequential reasoning

blue thread

green thread

Prior systems with mover reasoning

CIVL [CAV ’15, CAV ’18]	framework relies pen & paper proofs
IronFleet [SOSP ’15]	only move network send/receive

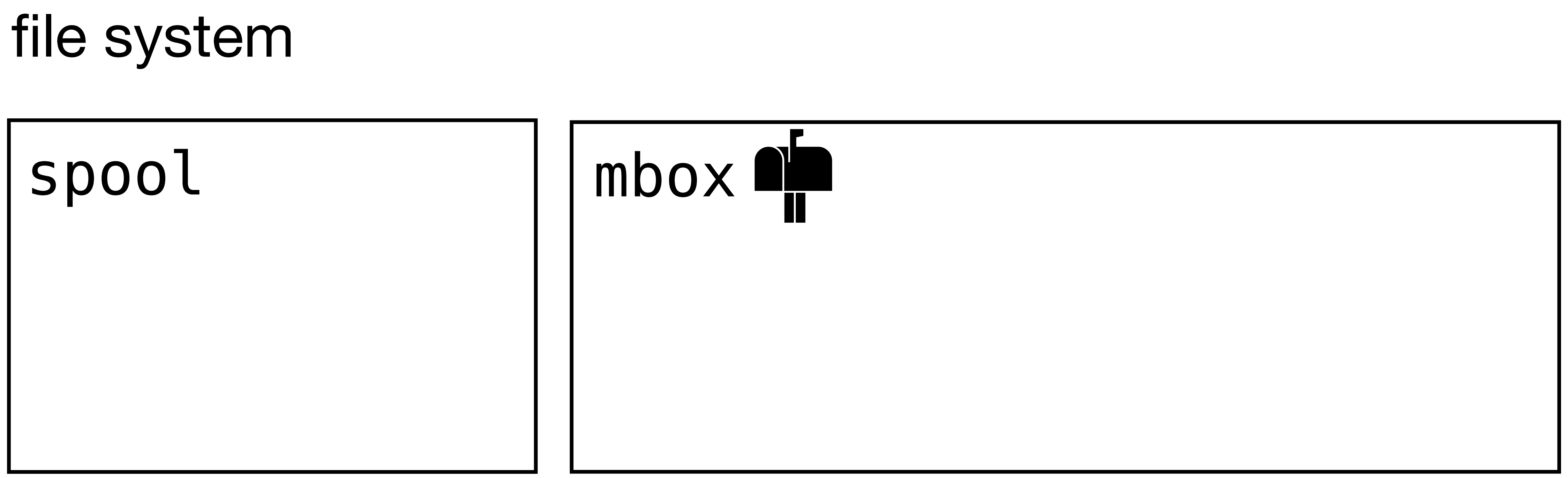
Contribution: CSPEC

- Framework for verifying concurrency in systems software
  - **general-purpose movers**
  - **patterns** to support mover reasoning
  - **machine checked** in Coq to support extensibility

Contribution: CSPEC

- Framework for verifying concurrency in systems software
  - **general-purpose movers**
  - **patterns** to support mover reasoning
  - **machine checked** in Coq to support extensibility
- Case studies using CSPEC
  - Lock-free file-system concurrency
  - Spinlock on top of x86-TSO (see paper)

Case study: mail server using file-system concurrency



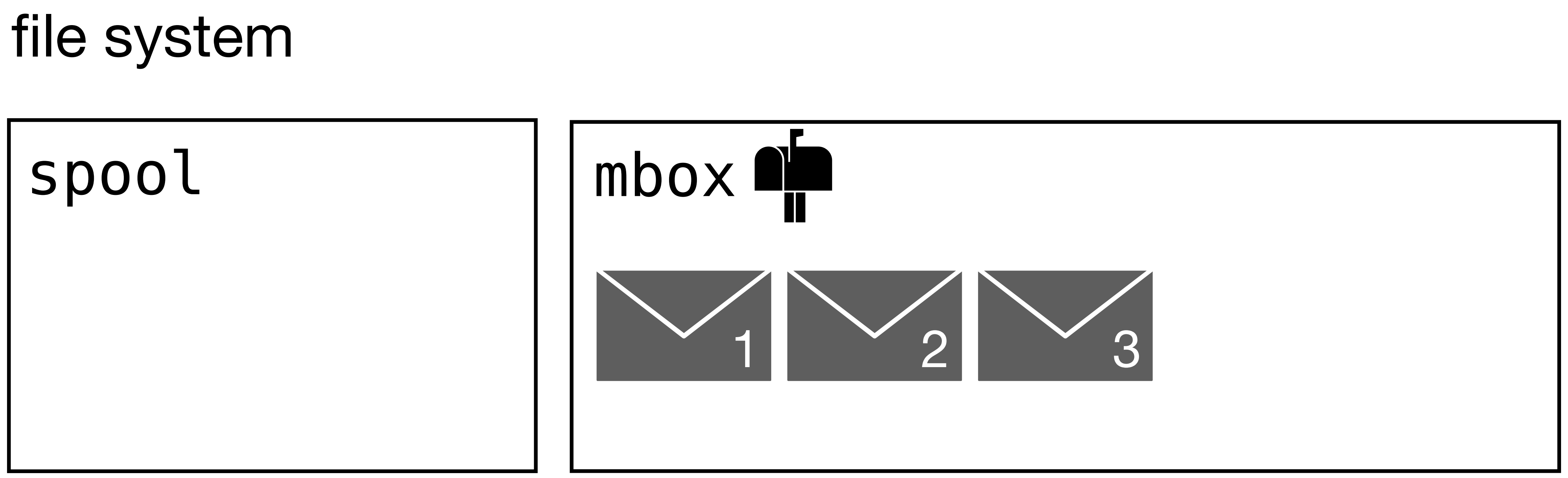
file system

spool

mbox 📫

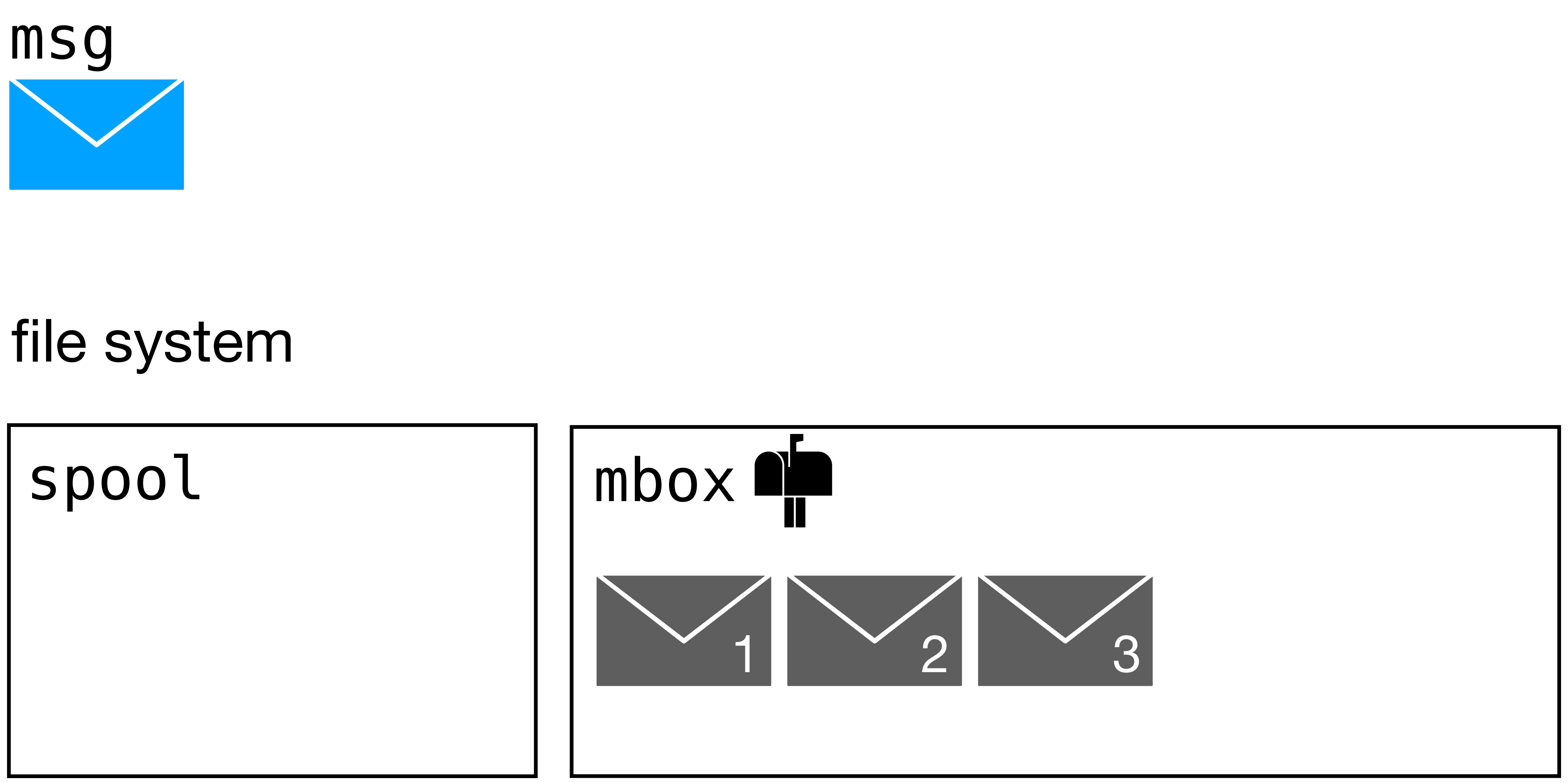
Mail servers exploit file-system concurrency

```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
  write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```



Mail servers exploit file-system concurrency

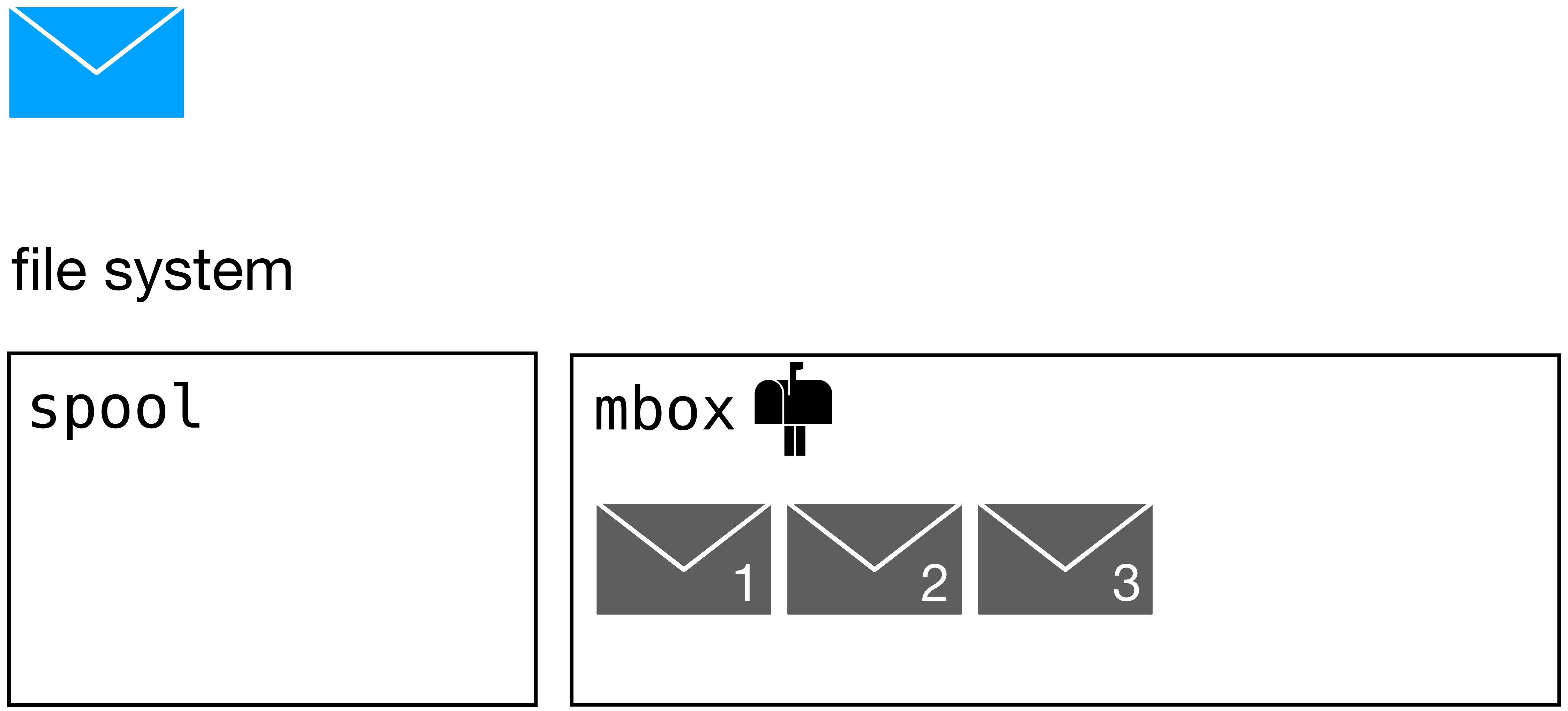
```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
  write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```



Spooling avoids reading partially-written messages

$TID =10

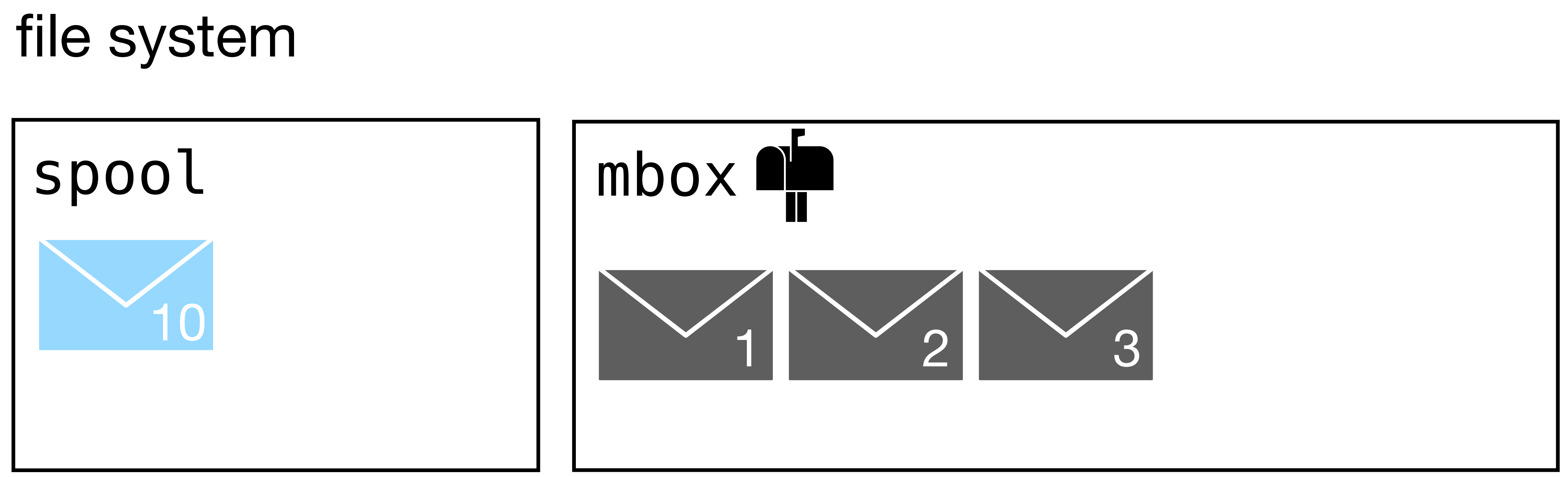
```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
▶ write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```



Spooling avoids reading partially-written messages

$TID =10

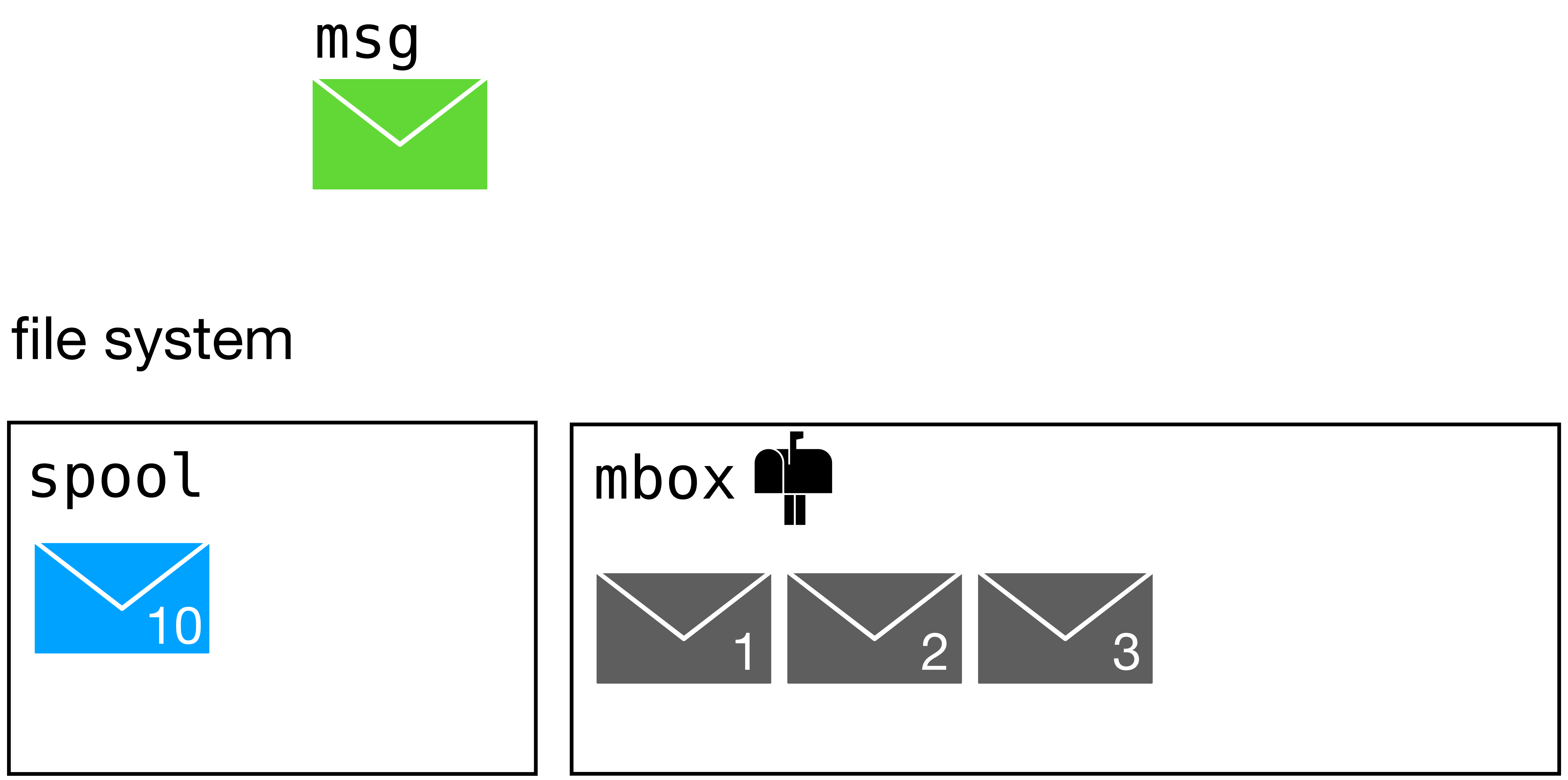
```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
  write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
```



Threads use unique IDs to avoid conflicts

$TID =10 $TID =11

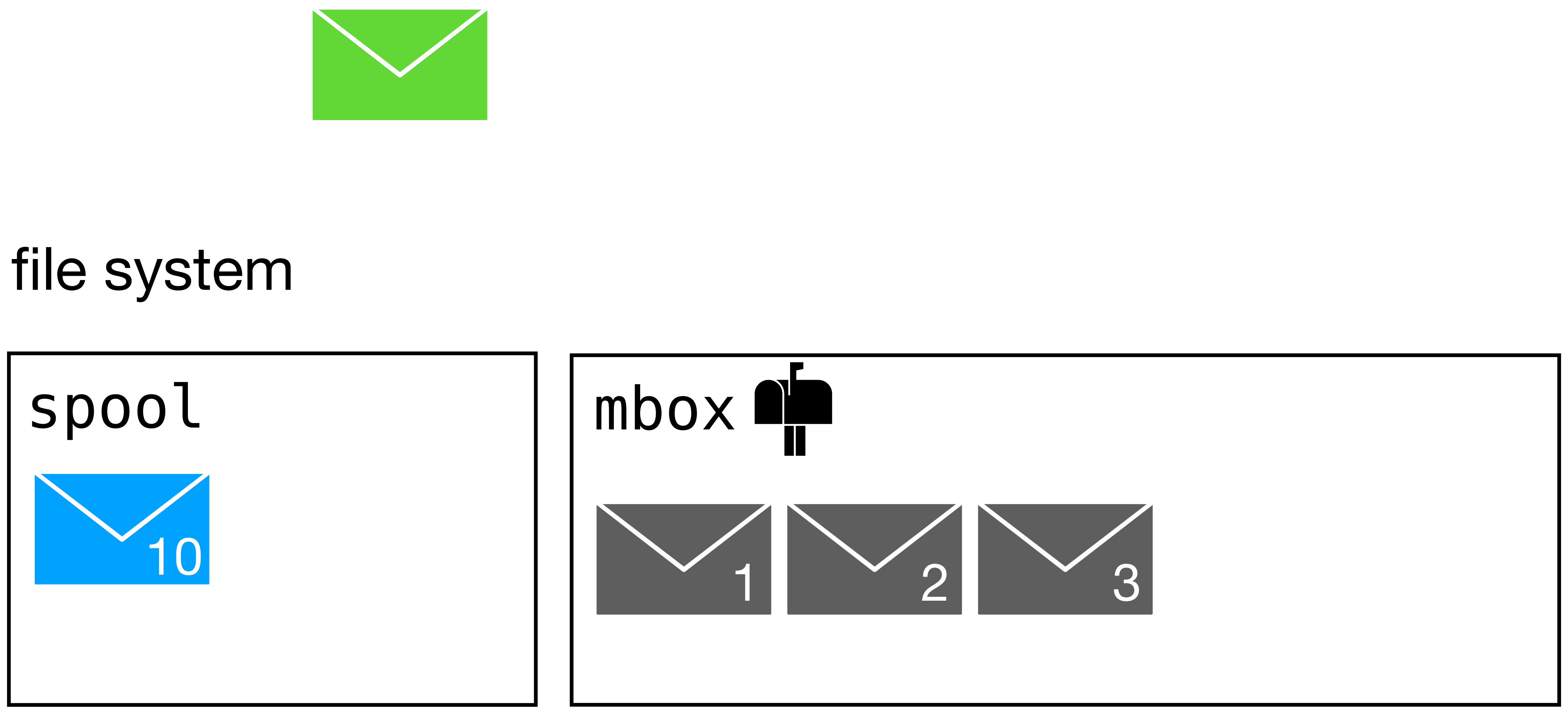
```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
  write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```



Threads use unique IDs to avoid conflicts

$TID =10 $TID =11

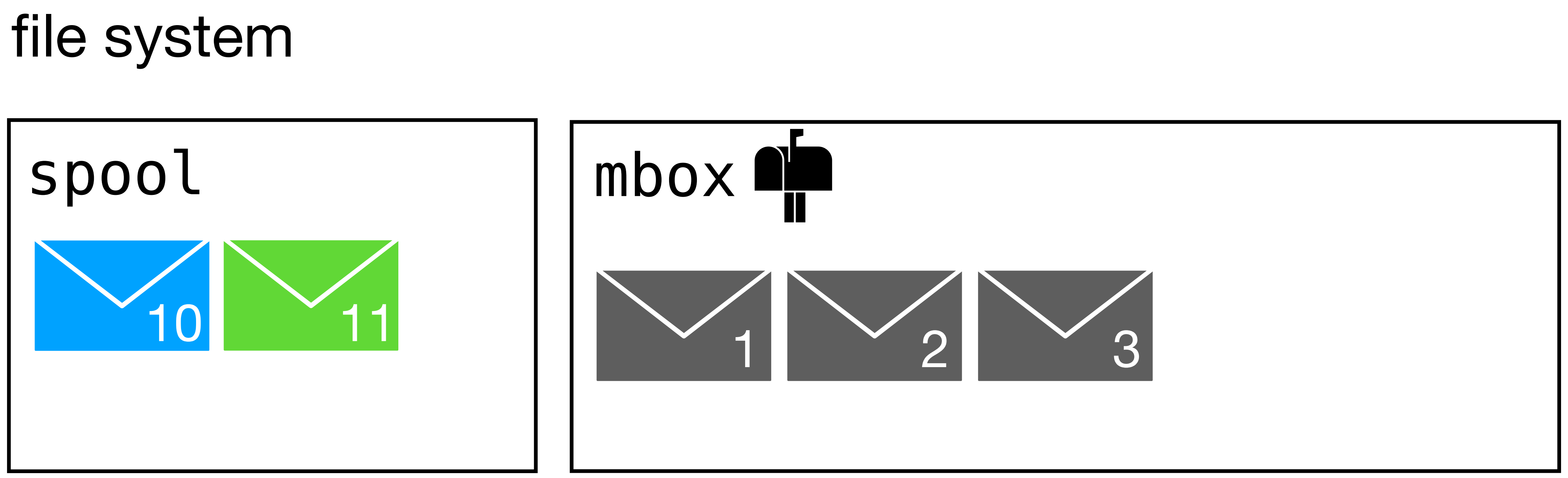
```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
▶▶write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
```



Threads use unique IDs to avoid conflicts

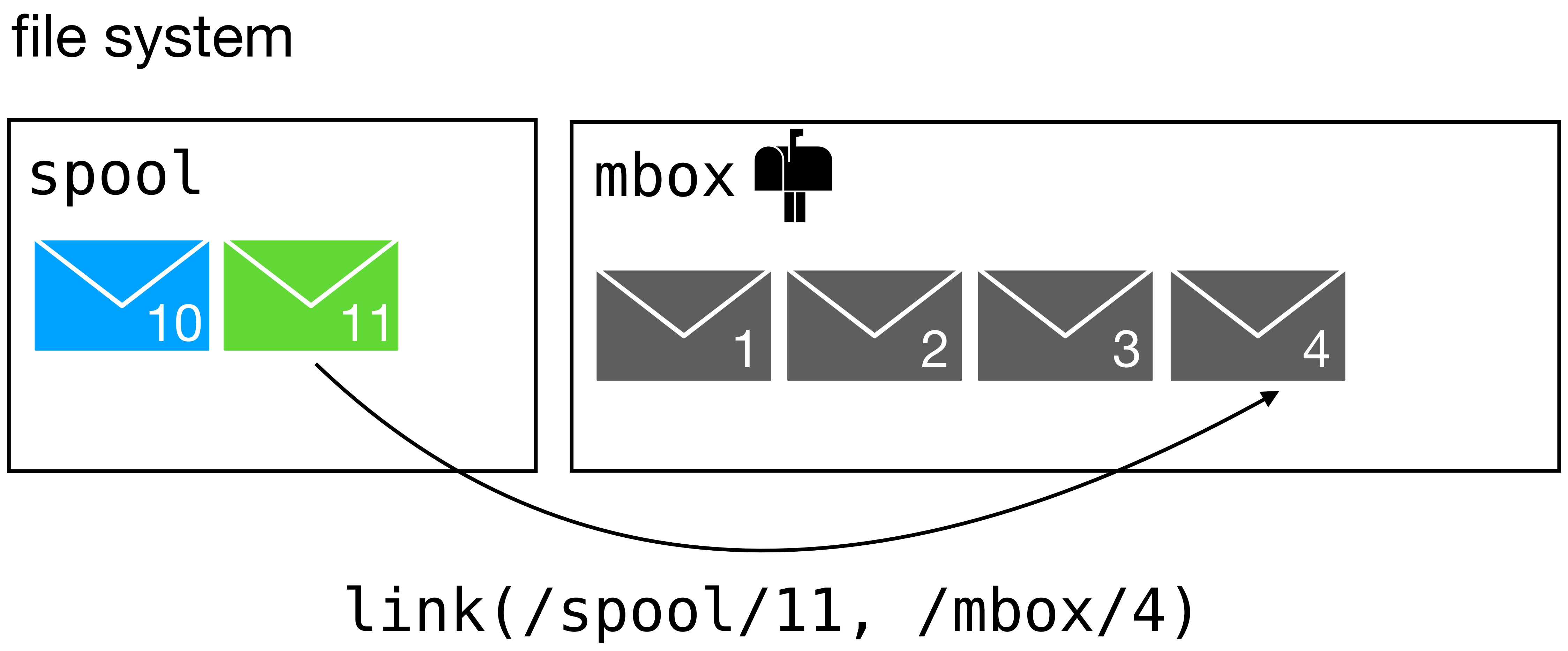
$TID =10 $TID =11

```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
▶▶write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```



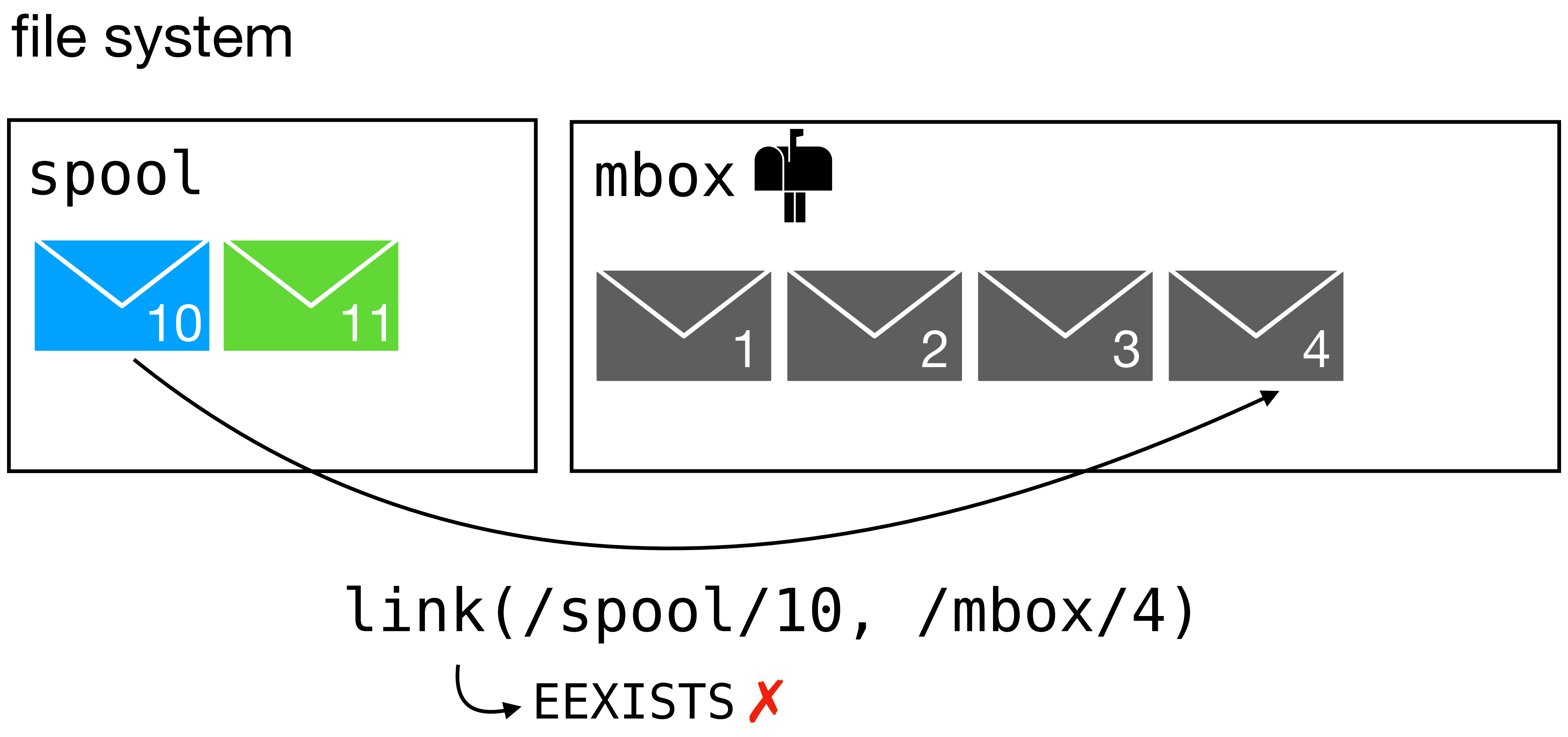
Timestamps help generate unique message names

```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
  write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
```



Timestamps help generate unique message names

```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
  write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
```



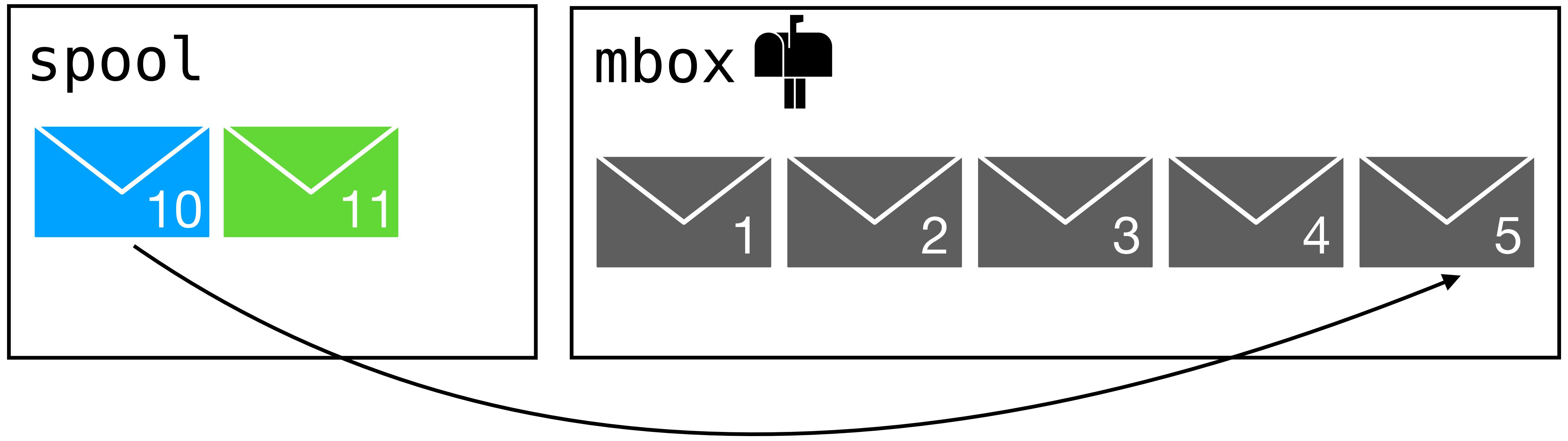
link(/spool/10, /mbox/4)

EEXISTS ✗

Timestamps help generate unique message names

```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
  write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
```

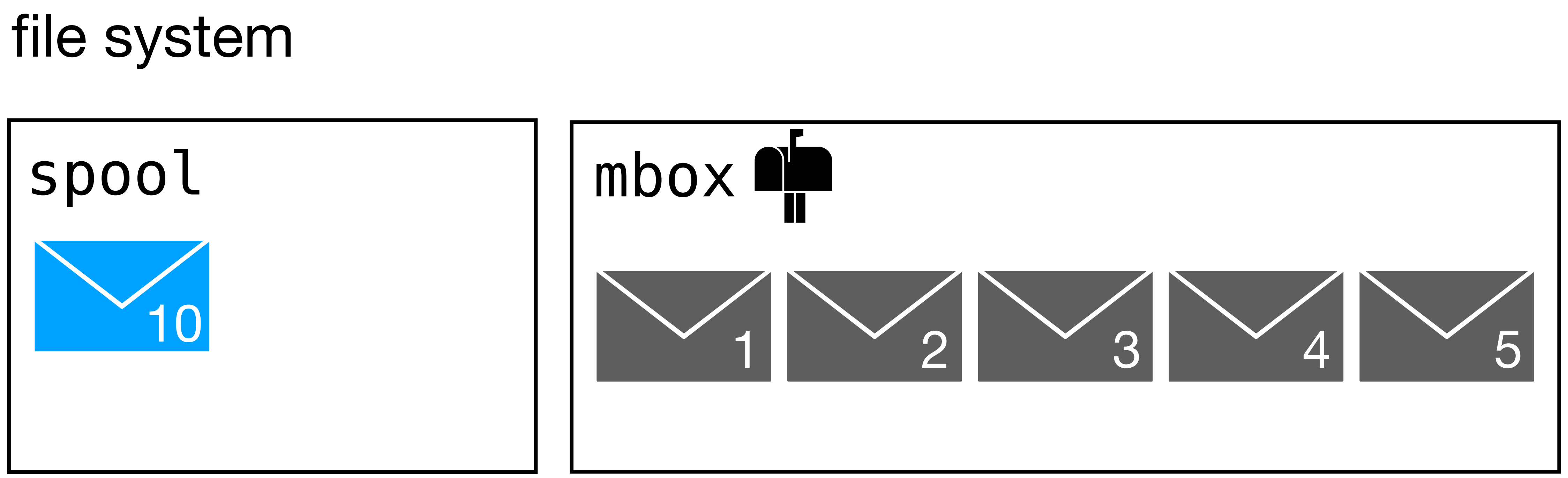
file system



link(/spool/10, /mbox/5)

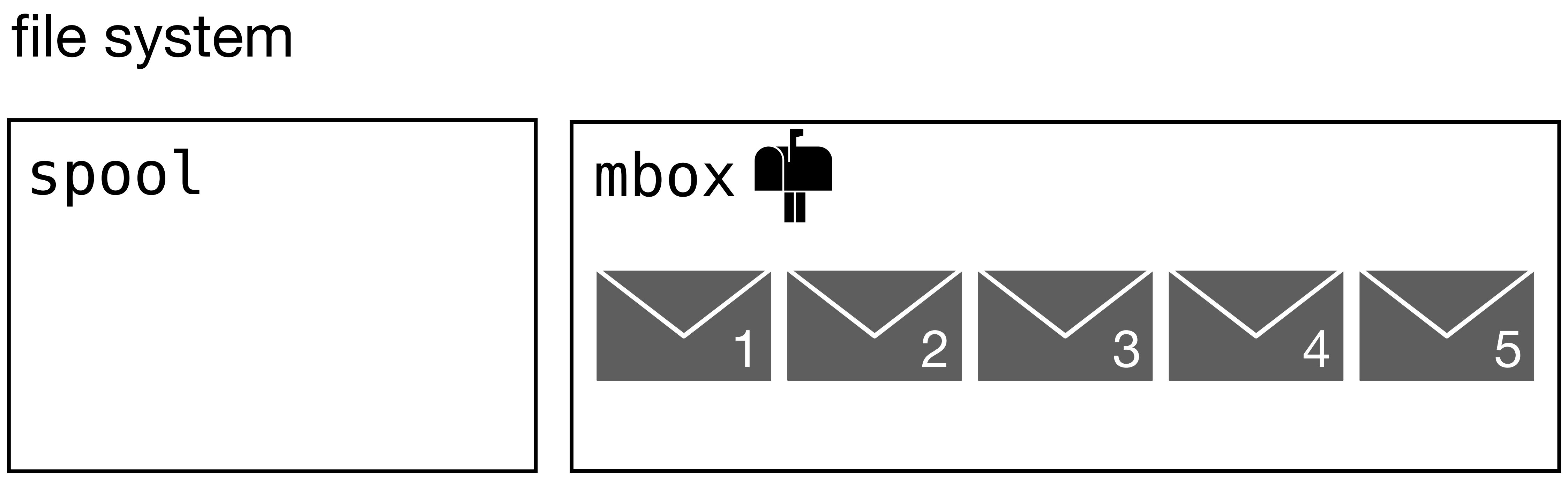
Delivery concurrency does not use locks

```
# accept
def deliver(msg):
  # spool
  create(“/spool/$TID”)
  write(“/spool/$TID”, msg)
  # store
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  # cleanup
  unlink(“/spool/$TID”)
```

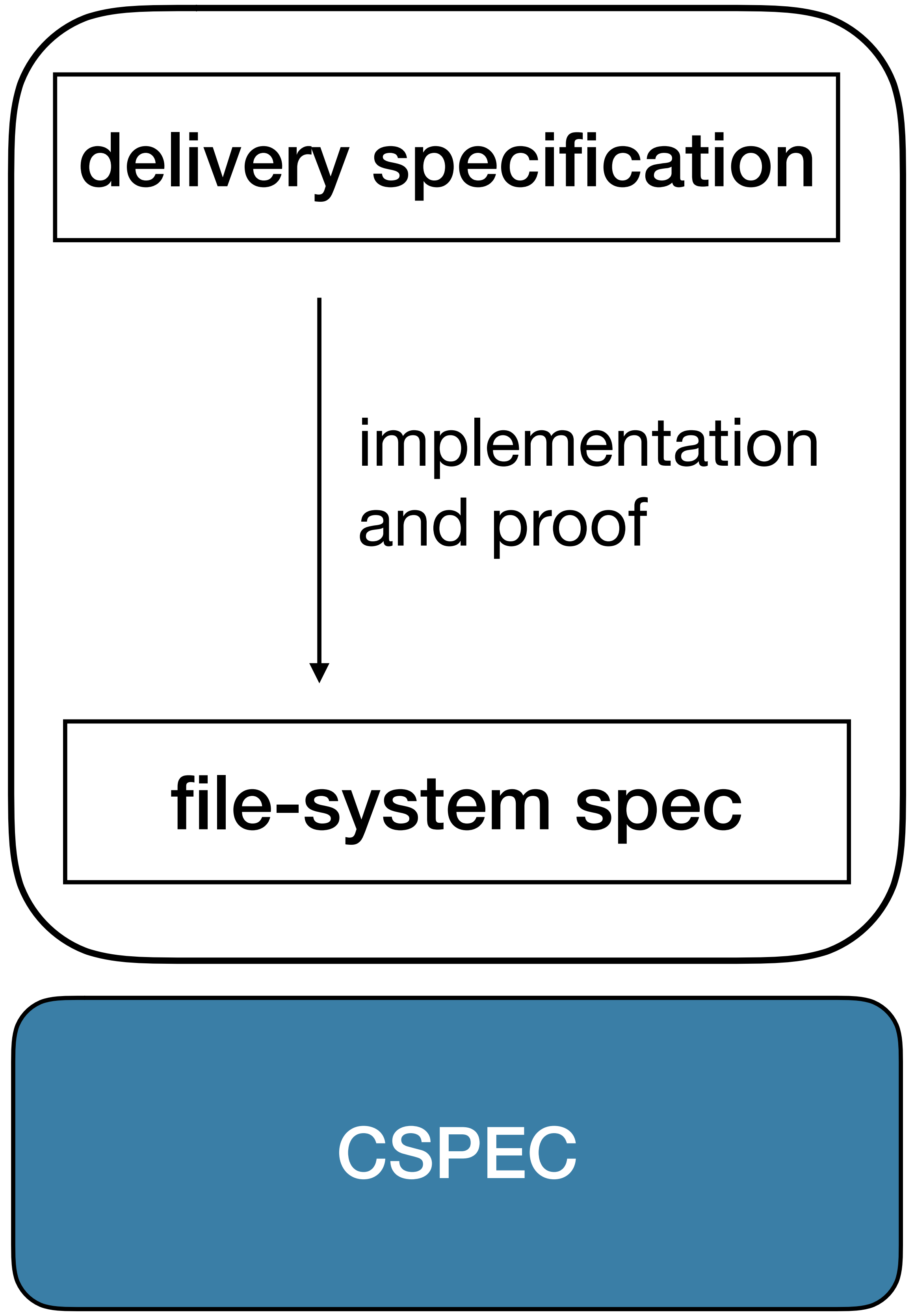


Delivery concurrency does not use locks

```
# accept
def deliver(msg):
  # spool
  create("/spool/$TID")
  write("/spool/$TID", msg)
  # store
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  # cleanup
  unlink("/spool/$TID")
▶▶
```



Proving delivery correct in CSPEC



delivery specification

implementation and proof

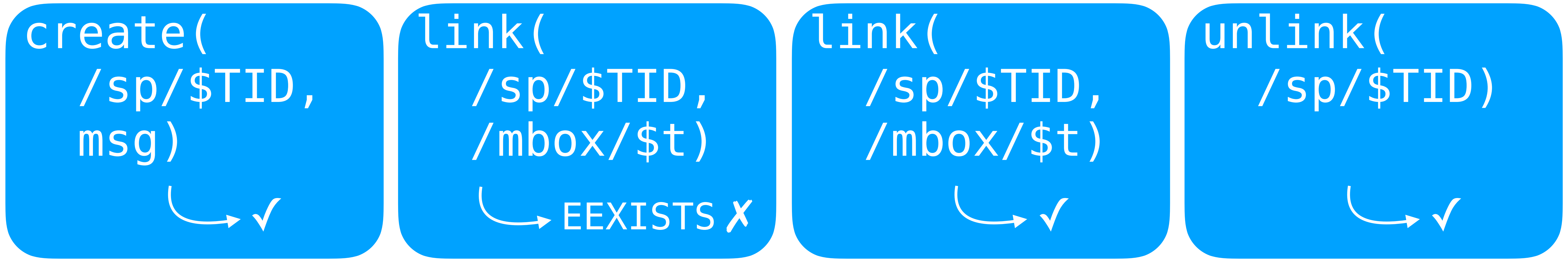
file-system spec

CSPEC

CSPEC provides supporting definitions and theorems

Proof engineer reasons about file-system operations

```
def deliver(msg):
  create("/spool/$TID", msg)
  while True:
    t = time.time()
    if link("/spool/$TID",
            "/mbox/$t"):
      break
  unlink("/spool/$TID")
```



```
create(
  /sp/$TID,
  msg)
      ↳ ✓
```

```
link(
  /sp/$TID,
  /mbox/$t)
    ↳ EEXISTS ✗
```

```
link(
  /sp/$TID,
  /mbox/$t)
      ↳ ✓
```

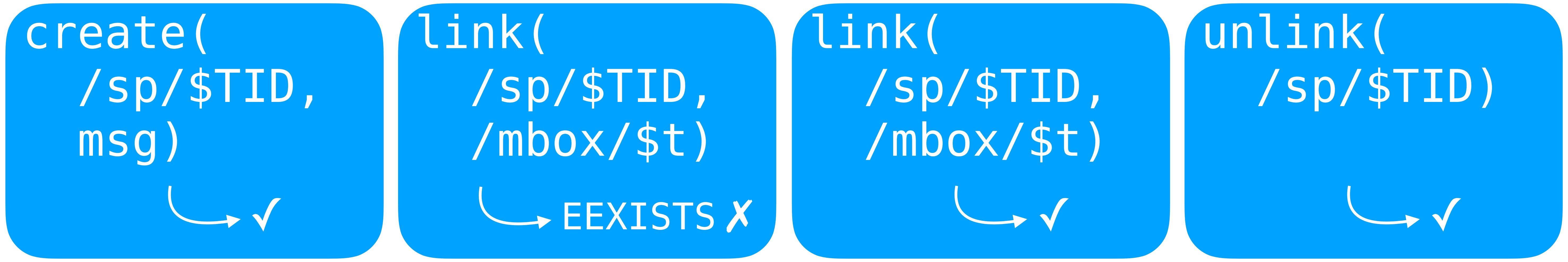
```
unlink(
  /sp/$TID)

      ↳ ✓
```

Proof engineer reasons about file-system operations

```
def deliver(msg):
  create(“/spool/$TID”, msg)
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  unlink(“/spool/$TID”)
```

collapsed to
one operation ←

```
create(“/spool/$TID”)
write(“/spool/$TID”, msg)
```



```
create(
  /sp/$TID,
  msg)
      ↳ ✓
```

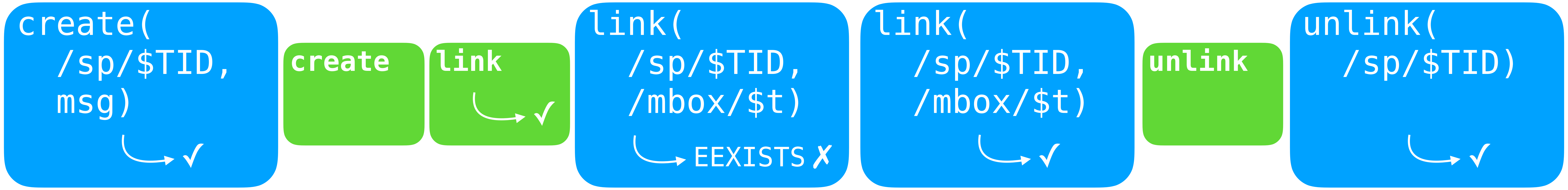
```
link(
  /sp/$TID,
  /mbox/$t)
      ↳ EEXISTS ✗
```

```
link(
  /sp/$TID,
  /mbox/$t)
      ↳ ✓
```

```
unlink(
  /sp/$TID)
      ↳ ✓
```

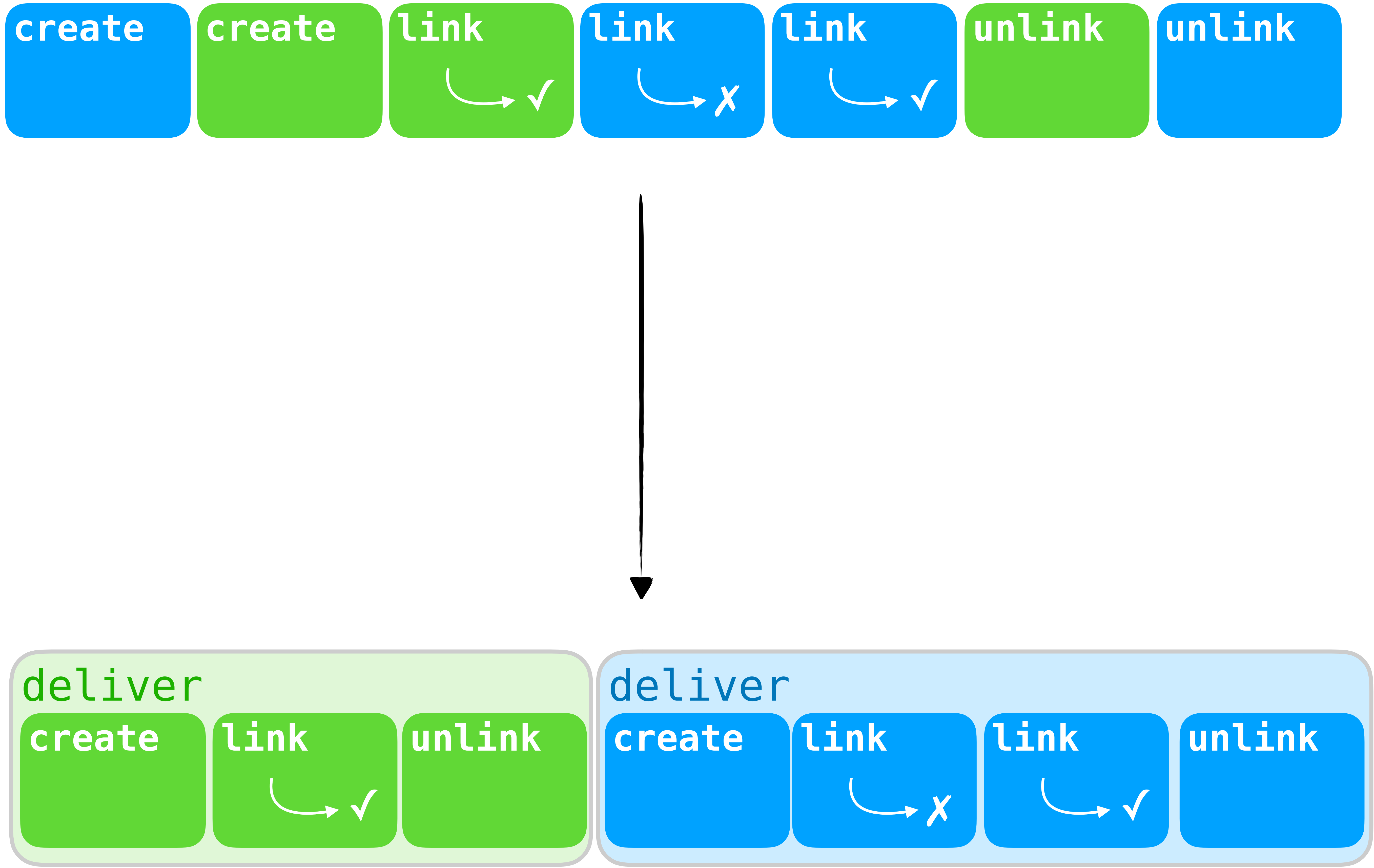
Proof engineer reasons about interleaving of file-system operations

```
def deliver(msg):
  create(“/spool/$TID”, msg)
  while True:
    t = time.time()
    if link(“/spool/$TID”,
            “/mbox/$t”):
      break
  unlink(“/spool/$TID”)
```



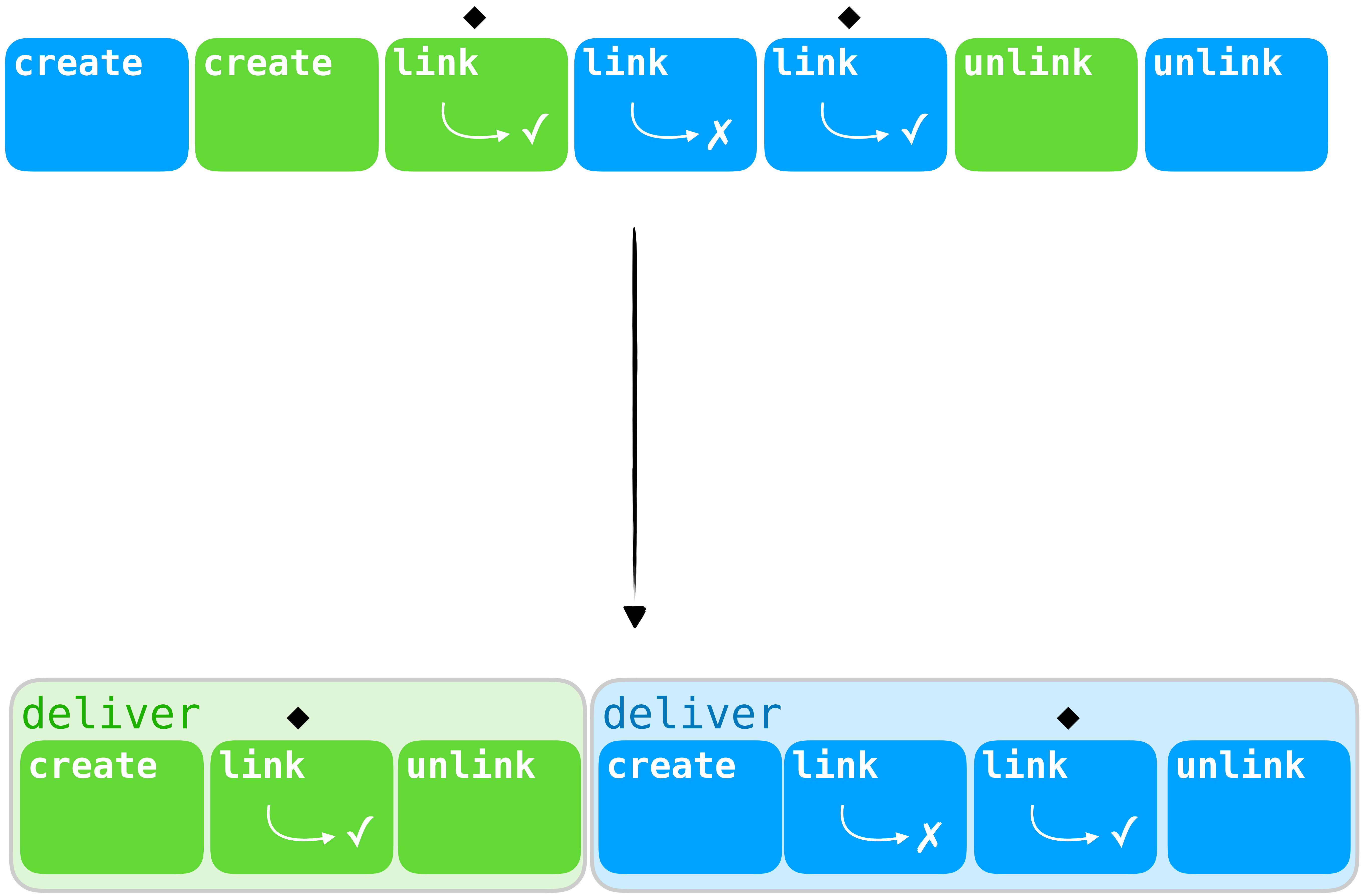
We assume file-system operations are atomic

Proving atomicity of delivery



atomicity: concurrent deliveries appear to execute all at once (in some order)

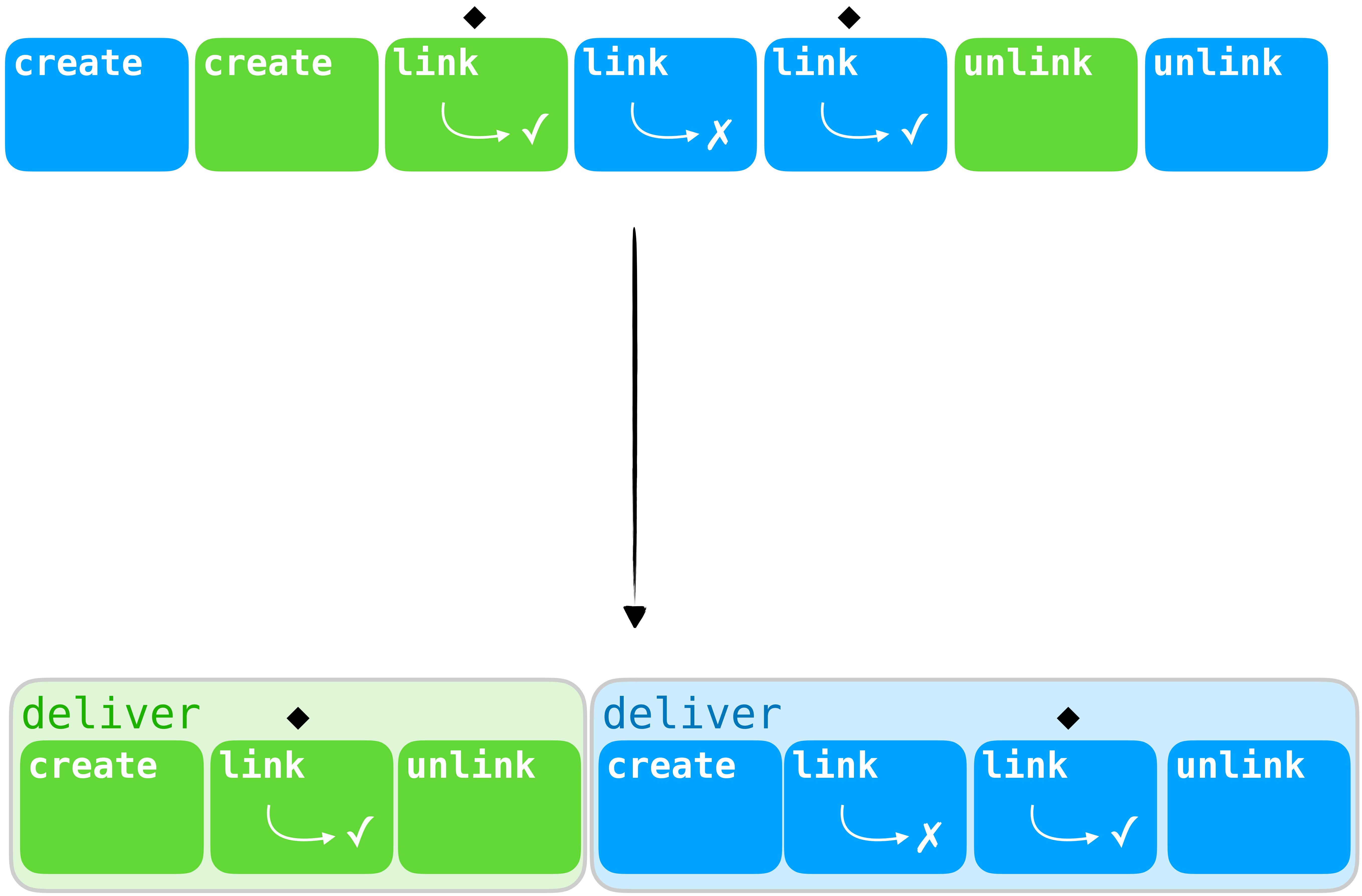
Proving atomicity of delivery



atomicity: concurrent deliveries appear to execute all at once (in some order)

Step 1: developer identifies commit point

Proving atomicity of delivery

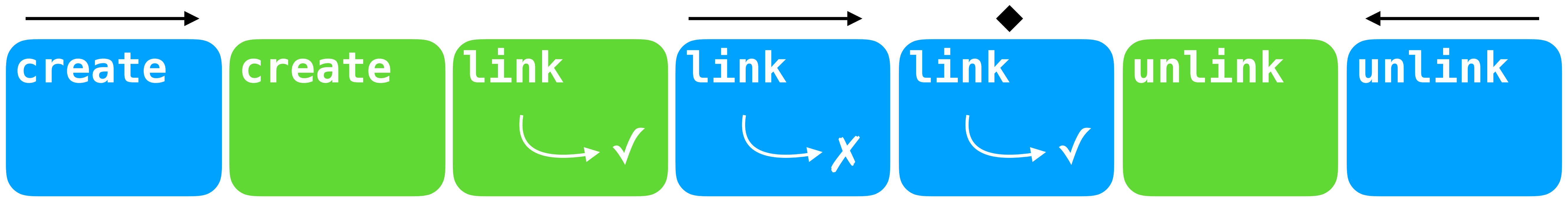


atomicity: concurrent deliveries appear to execute all at once (in some order)

Step 1: developer identifies commit point

Step 2: prove operation occurs logically at commit point

Example of movers for this execution



create

create

link ✓

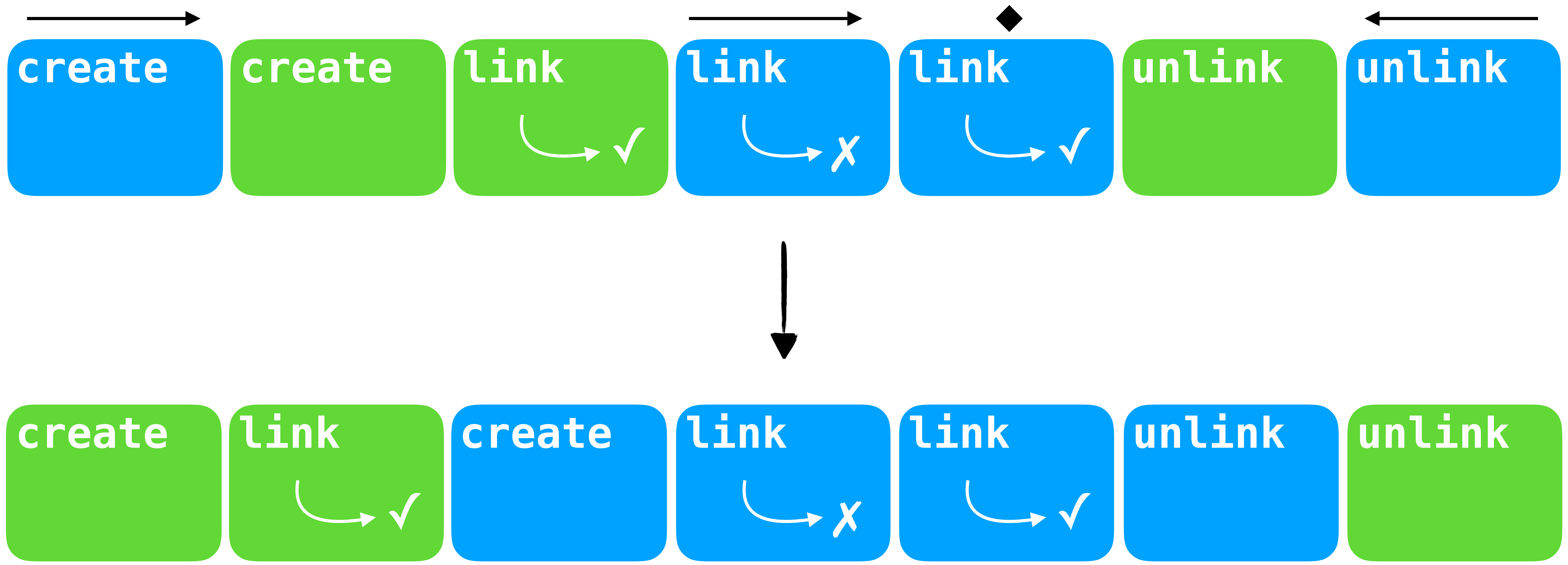
link ✗

link ✓

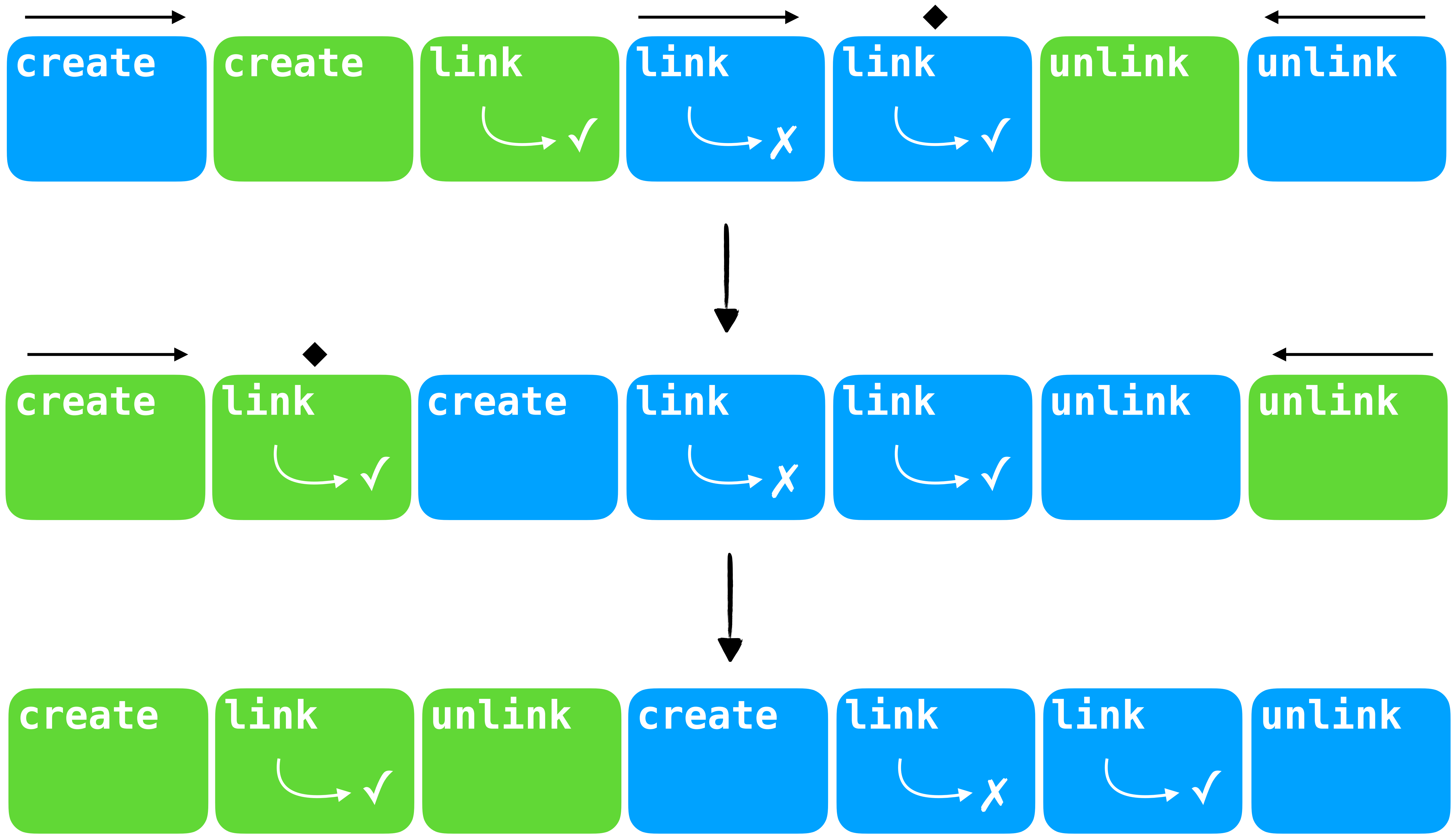
unlink

unlink

Example of movers for this execution

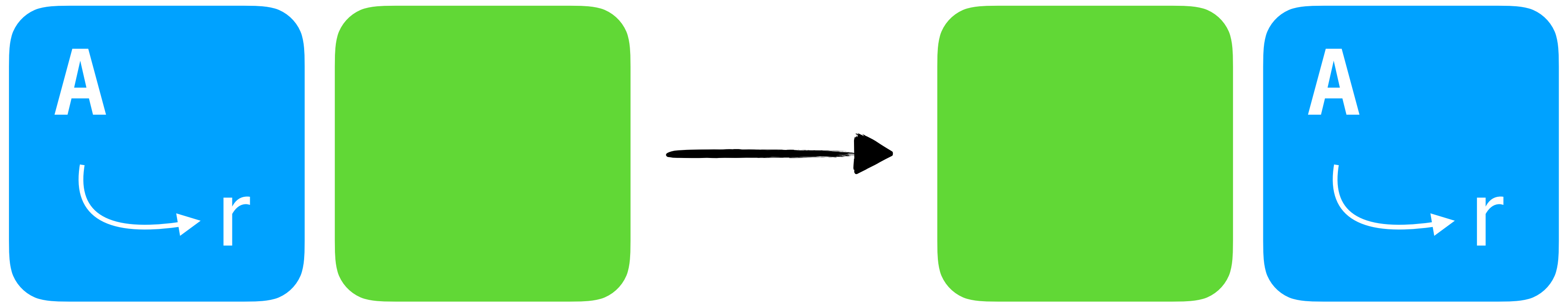


Example of movers for this execution

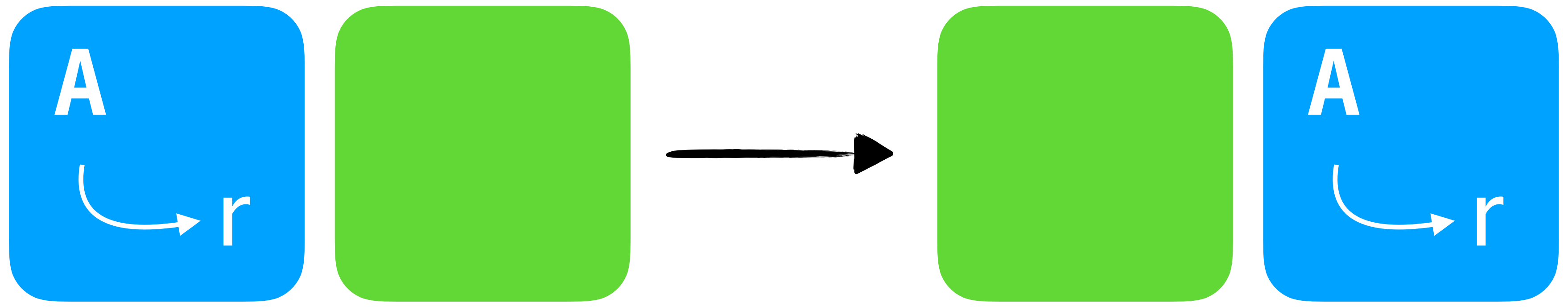


→			→	◆		←
create	create	link ✓	link ✗	link ✓	unlink	unlink
→	◆					←
create	link ✓	create	link ✗	link ✓	unlink	unlink
create	link ✓	unlink	create	link ✗	link ✓	unlink

Right mover can be reordered after any green thread operation



Right mover can be reordered after any green thread operation

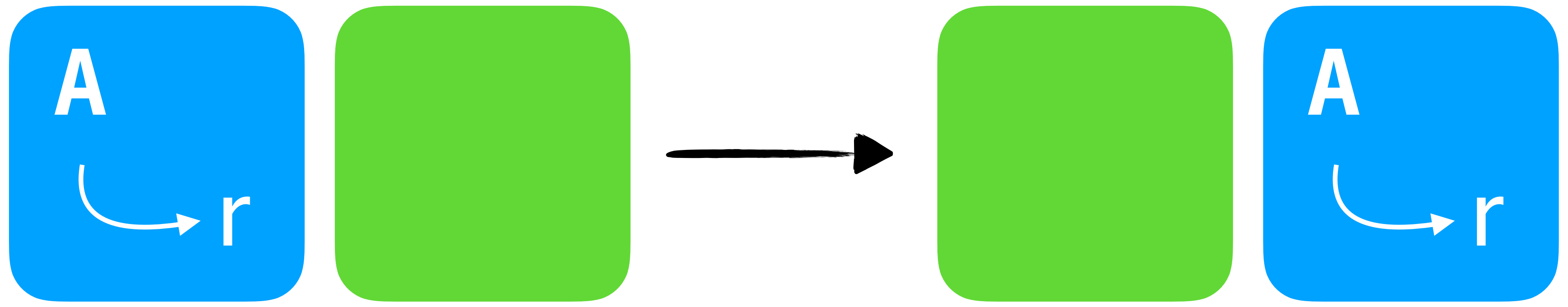


left movers are the converse

Movers need to consider only *possible* operations from other threads

A is a *right mover* if

for all green operations 🟩,



left movers are the converse

🟩 is one of

```
create(
  /sp/$TID,
  msg)
```

```
link(
  /sp/$TID,
  /mbox/$t)
```

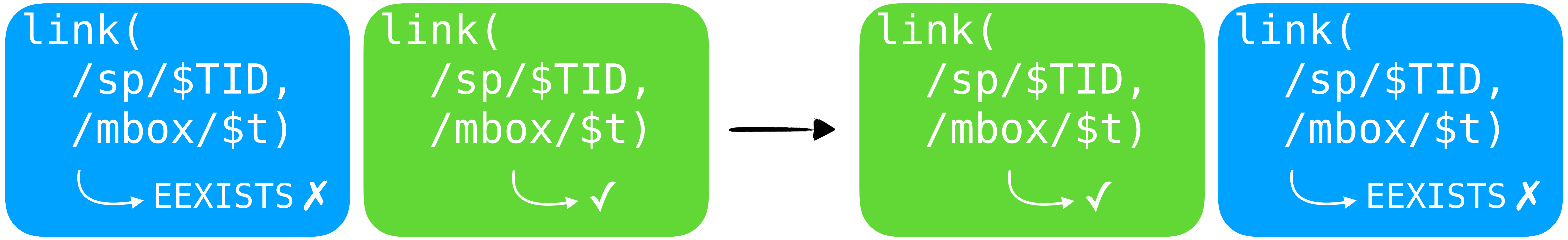
↪ EEXISTS ✗

```
link(
  /sp/$TID,
  /mbox/$t)
```

```
unlink(
  /sp/$TID)
```

Example mover proof: failing `link` is a *right mover*

Proof sketch (only `link` case):



link(
  /sp/$TID,
  /mbox/$t)
  ↳ EEXISTS ✗

link(
  /sp/$TID,
  /mbox/$t)
  ↳ ✓

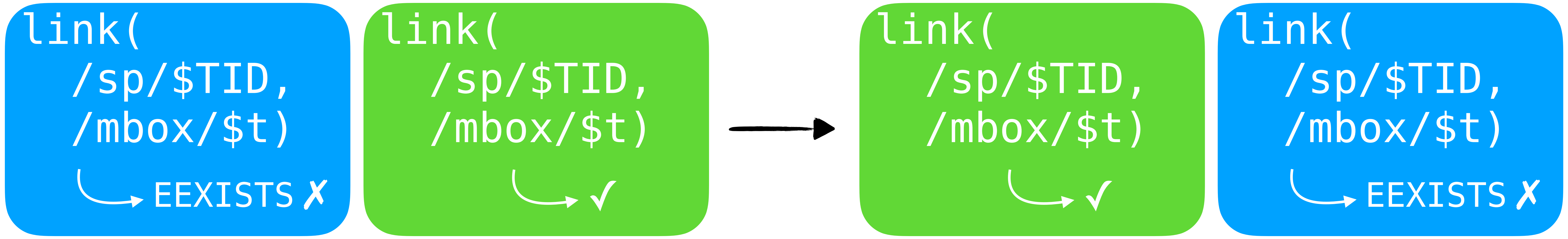
→

link(
  /sp/$TID,
  /mbox/$t)
  ↳ ✓

link(
  /sp/$TID,
  /mbox/$t)
  ↳ EEXISTS ✗

Example mover proof: failing `link` is a *right mover*

Proof sketch (only `link` case):

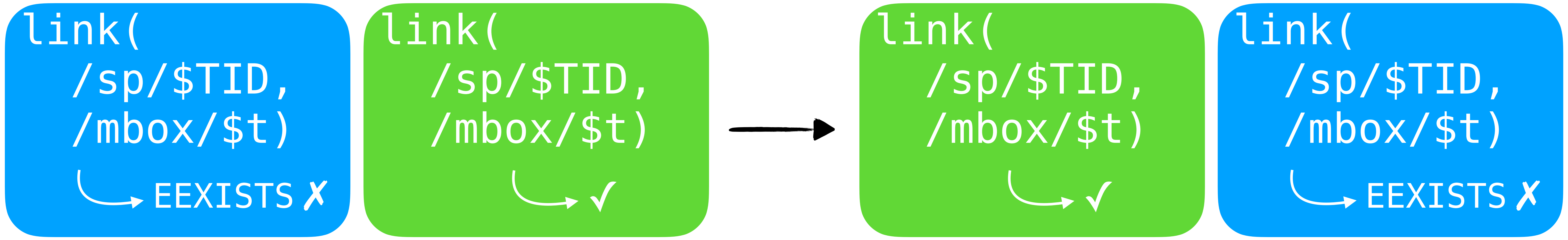


```
link(                link(                       link(                link(
  /sp/$TID,            /sp/$TID,                   /sp/$TID,            /sp/$TID,
  /mbox/$t)            /mbox/$t)        →          /mbox/$t)            /mbox/$t)
   ↪ EEXISTS ✗           ↪ ✓                         ↪ ✓                 ↪ EEXISTS ✗
```

`$t` ≠ `$t` (otherwise `link` ↪ ✗ then `link` ↪ ✓ is impossible)

Example mover proof: failing `link` is a *right mover*

Proof sketch (only `link` case):



```
link(
  /sp/$TID,
  /mbox/$t)
  ↪ EEXISTS ✗
```

```
link(
  /sp/$TID,
  /mbox/$t)
    ↪ √
```

⟶

```
link(
  /sp/$TID,
  /mbox/$t)
    ↪ √
```

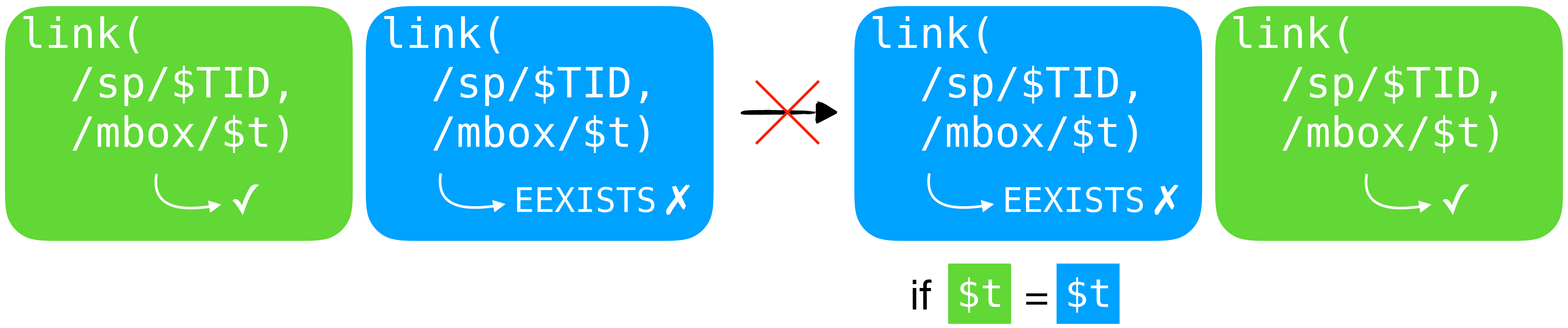
```
link(
  /sp/$TID,
  /mbox/$t)
  ↪ EEXISTS ✗
```

`$t` ≠ `$t` (otherwise `link` ↪ ✗ then `link` ↪ √ is impossible)

⟹ `link` operations are independent

Failing `link` does not move left

Failing `link` does not move left



```
link(
  /sp/$TID,
  /mbox/$t)
     ⤷ ✓
```

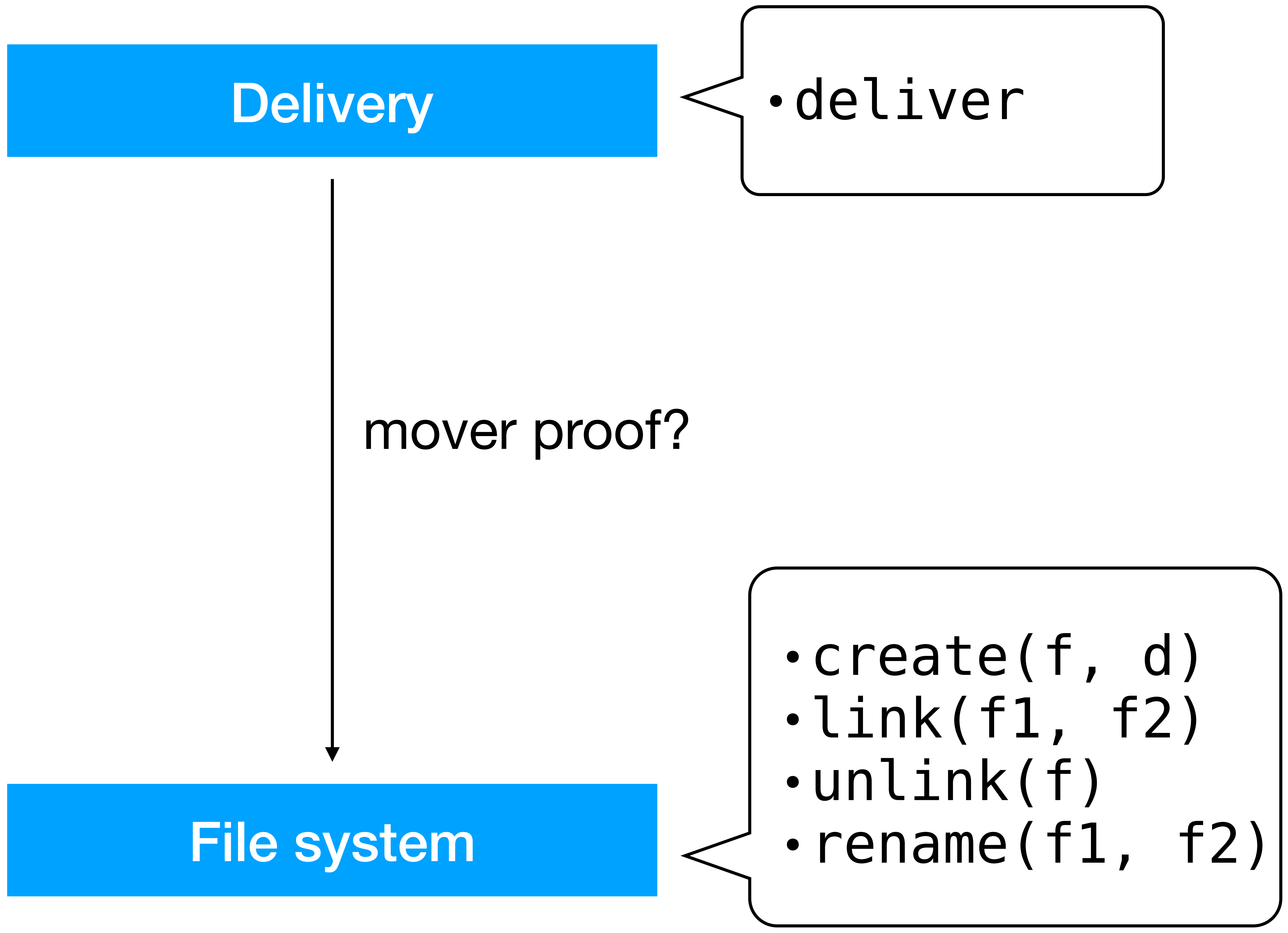
```
link(
  /sp/$TID,
  /mbox/$t)
   ⤷ EEXISTS ✗
```

```
link(
  /sp/$TID,
  /mbox/$t)
   ⤷ EEXISTS ✗
```

```
link(
  /sp/$TID,
  /mbox/$t)
     ⤷ ✓
```

if `$t` = `$t`

Challenge: how to limit what other operations to consider in mover proofs?



Delivery

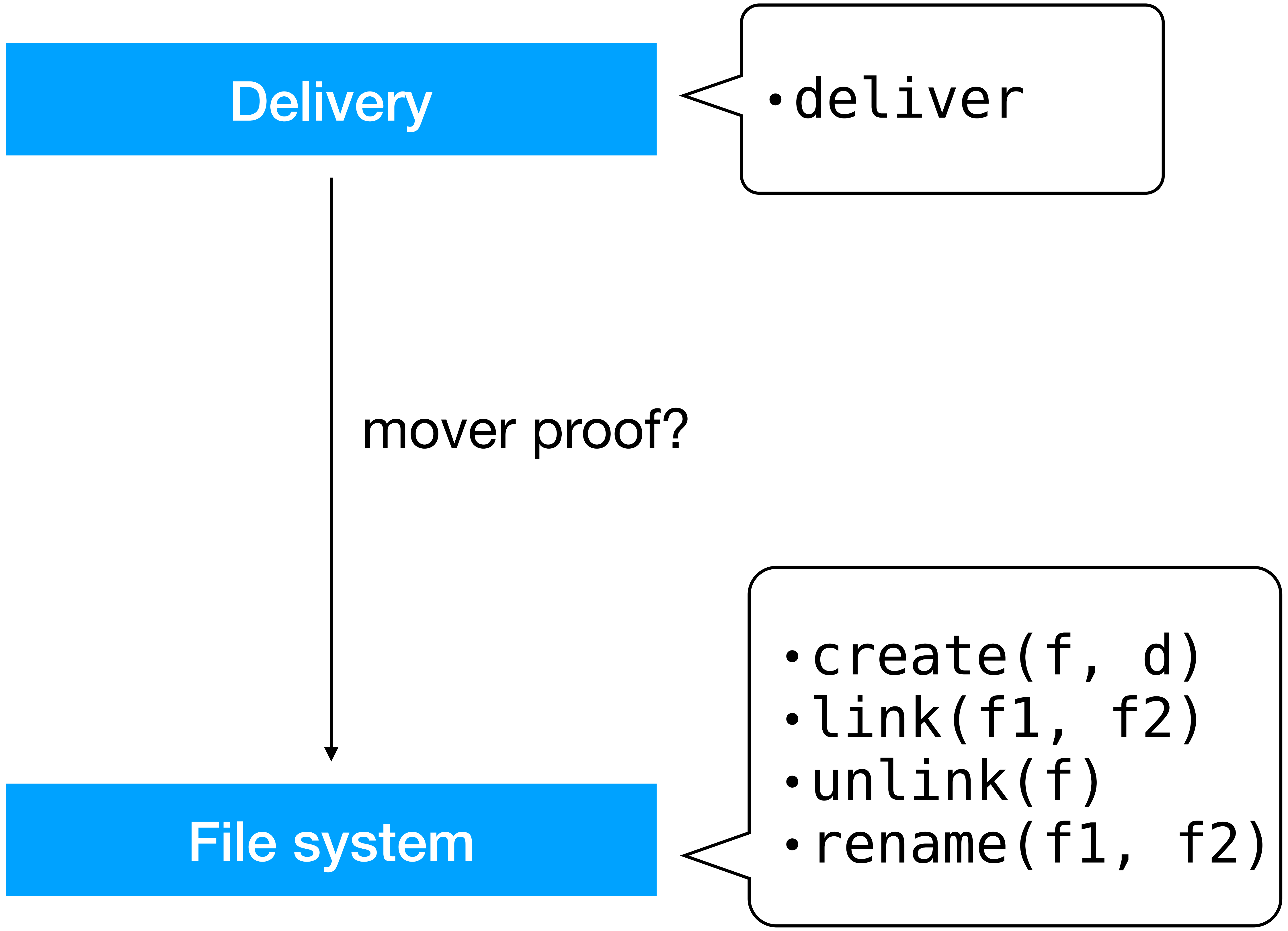
- `deliver`

mover proof?

File system

- `create(f, d)`
- `link(f1, f2)`
- `unlink(f)`
- `rename(f1, f2)`

Challenge: how to limit what other operations to consider in mover proofs?



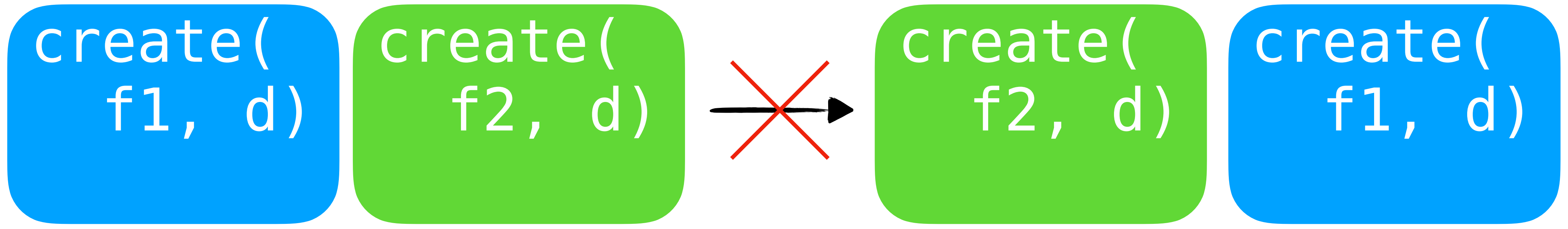
Delivery

- deliver

mover proof?

File system

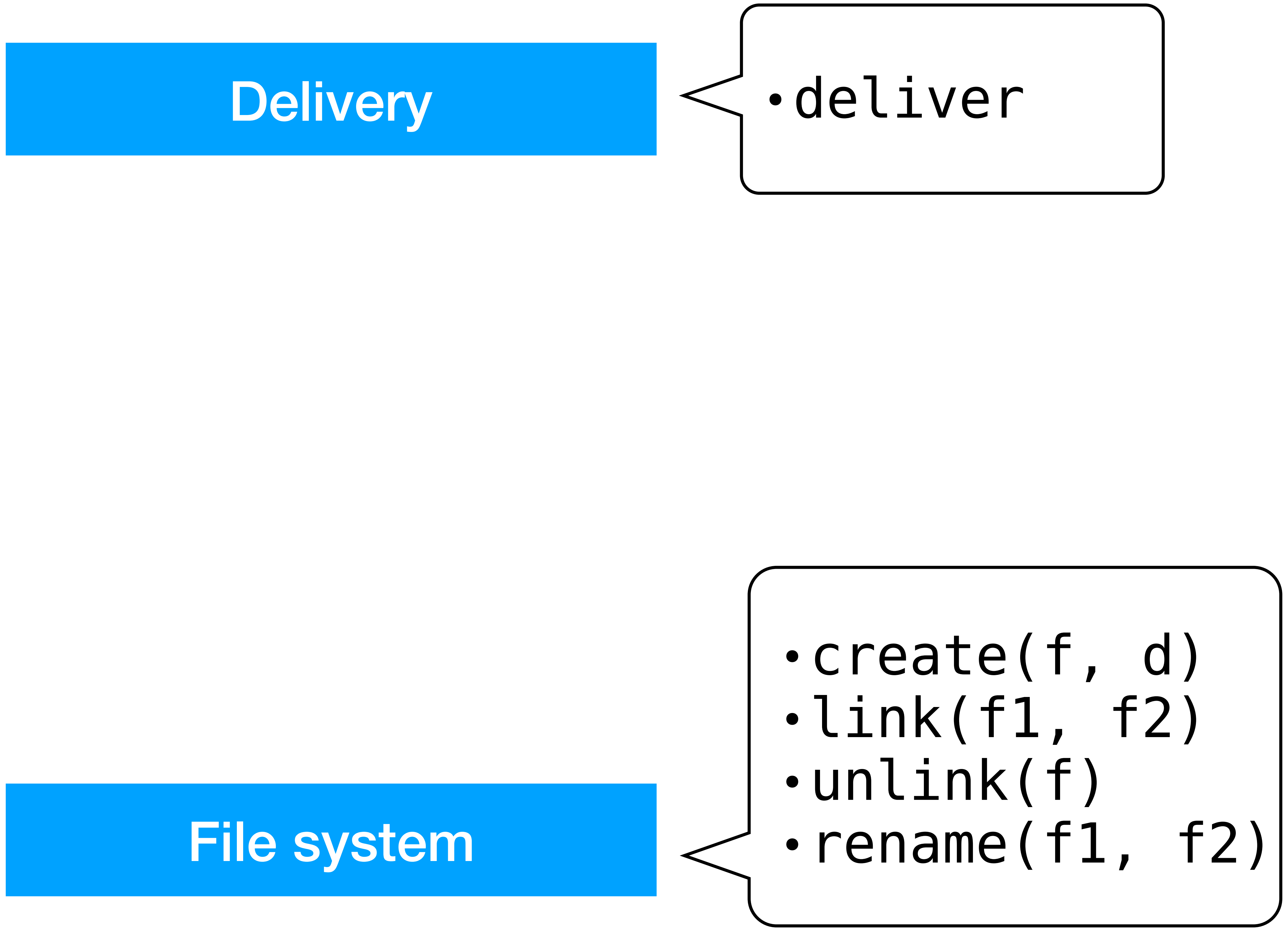
- create(f, d)
- link(f1, f2)
- unlink(f)
- rename(f1, f2)



if filenames are identical

Layers enable mover reasoning

Layers **limit** what operations are available
⟹ use **multiple layers** to make operations movers



Delivery

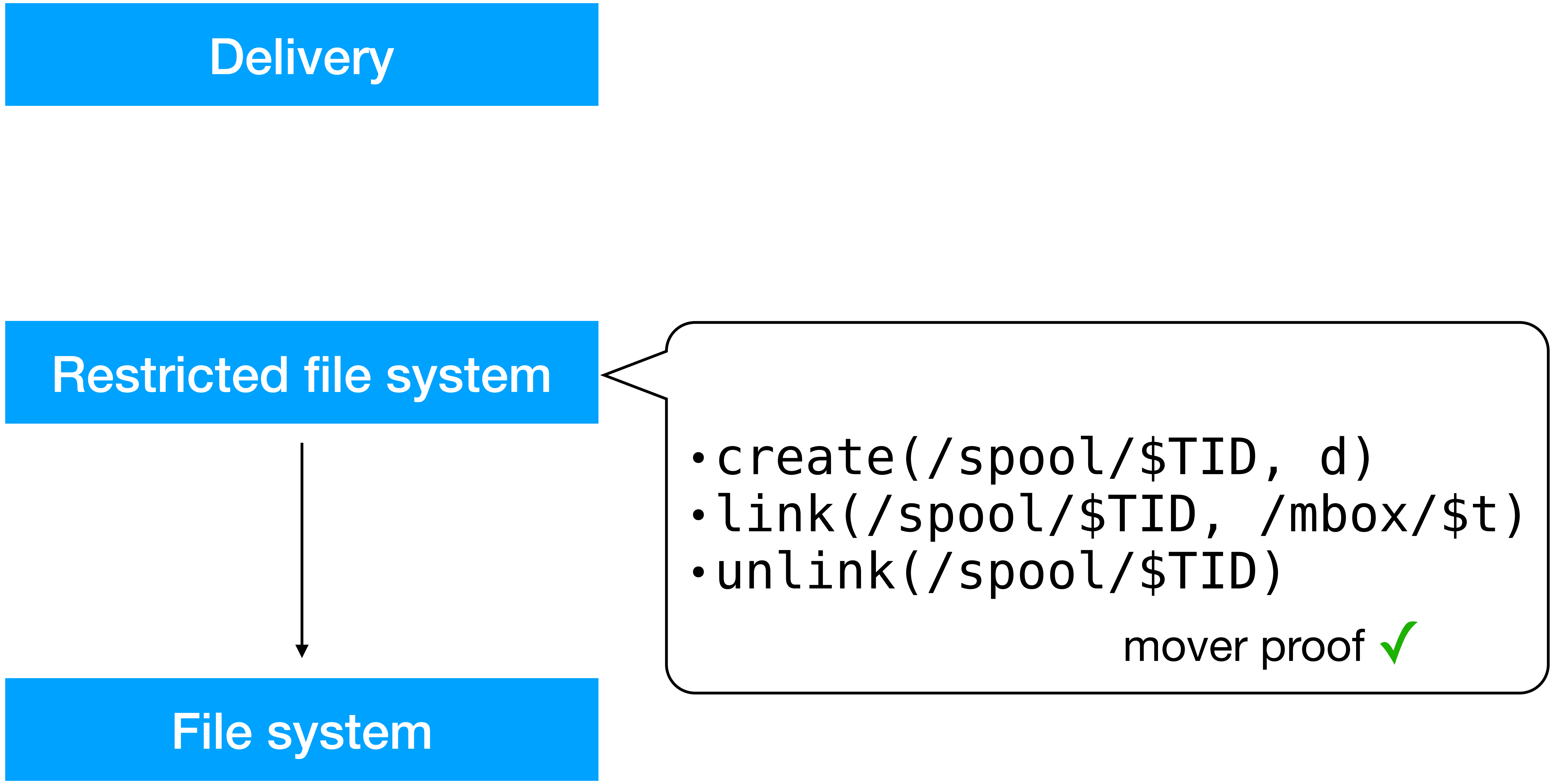
```
•deliver
```

File system

```
•create(f, d)
•link(f1, f2)
•unlink(f)
•rename(f1, f2)
```

Layers enable mover reasoning

Layers **limit** what operations are available
⟹ use **multiple layers** to make operations movers



Delivery

Restricted file system

File system

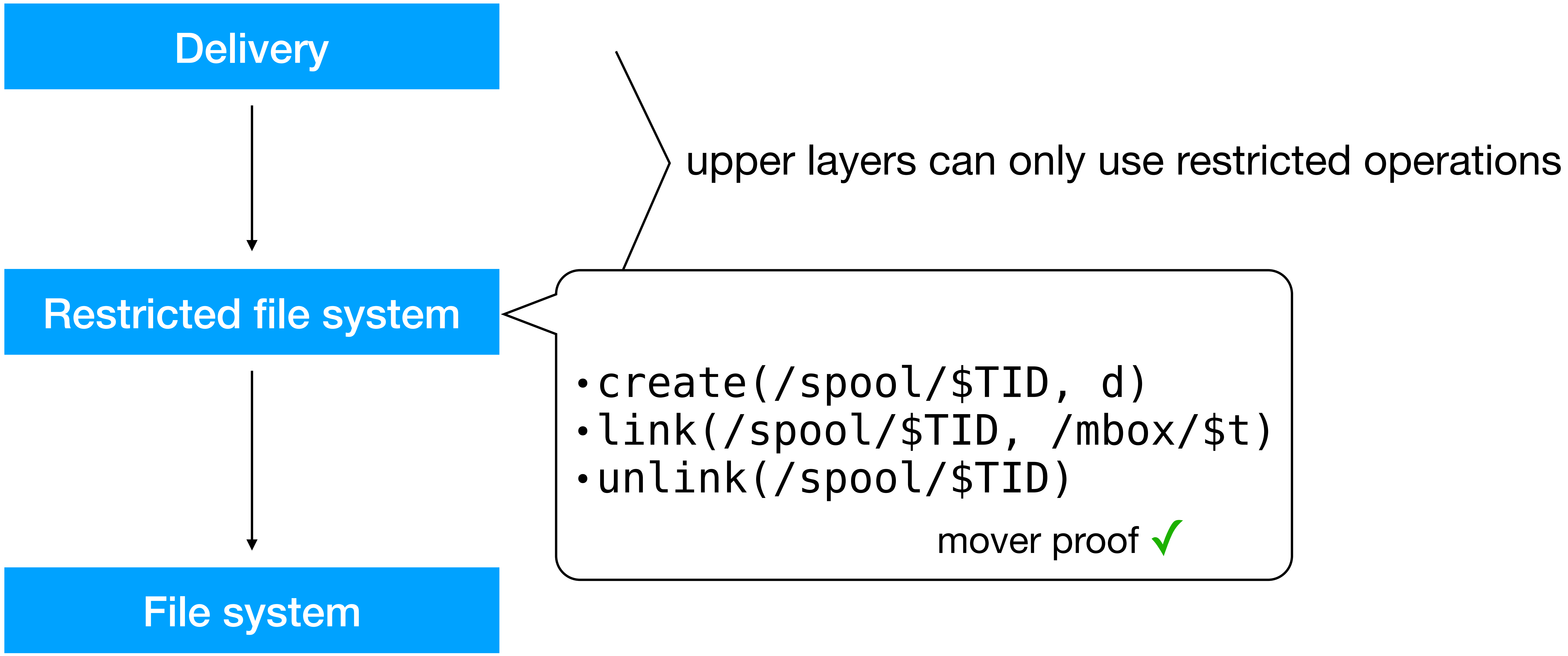
```
•create(/spool/$TID, d)
•link(/spool/$TID, /mbox/$t)
•unlink(/spool/$TID)
```

mover proof ✓

restrict arguments to include `$TID`

Layers enable mover reasoning

Layers **limit** what operations are available
⟹ use **multiple layers** to make operations movers

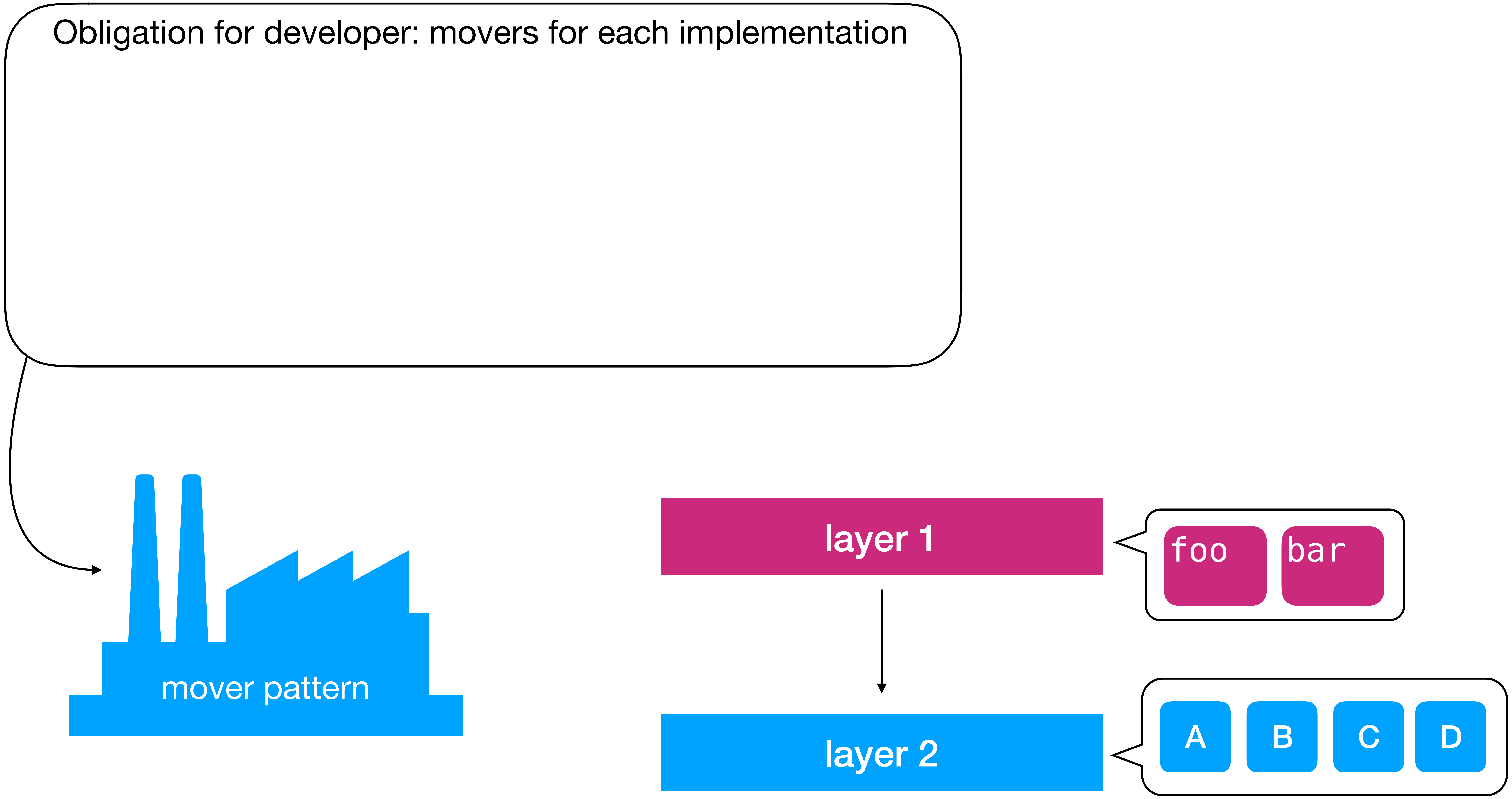


upper layers can only use restricted operations

```
•create(/spool/$TID, d)
•link(/spool/$TID, /mbox/$t)
•unlink(/spool/$TID)
```

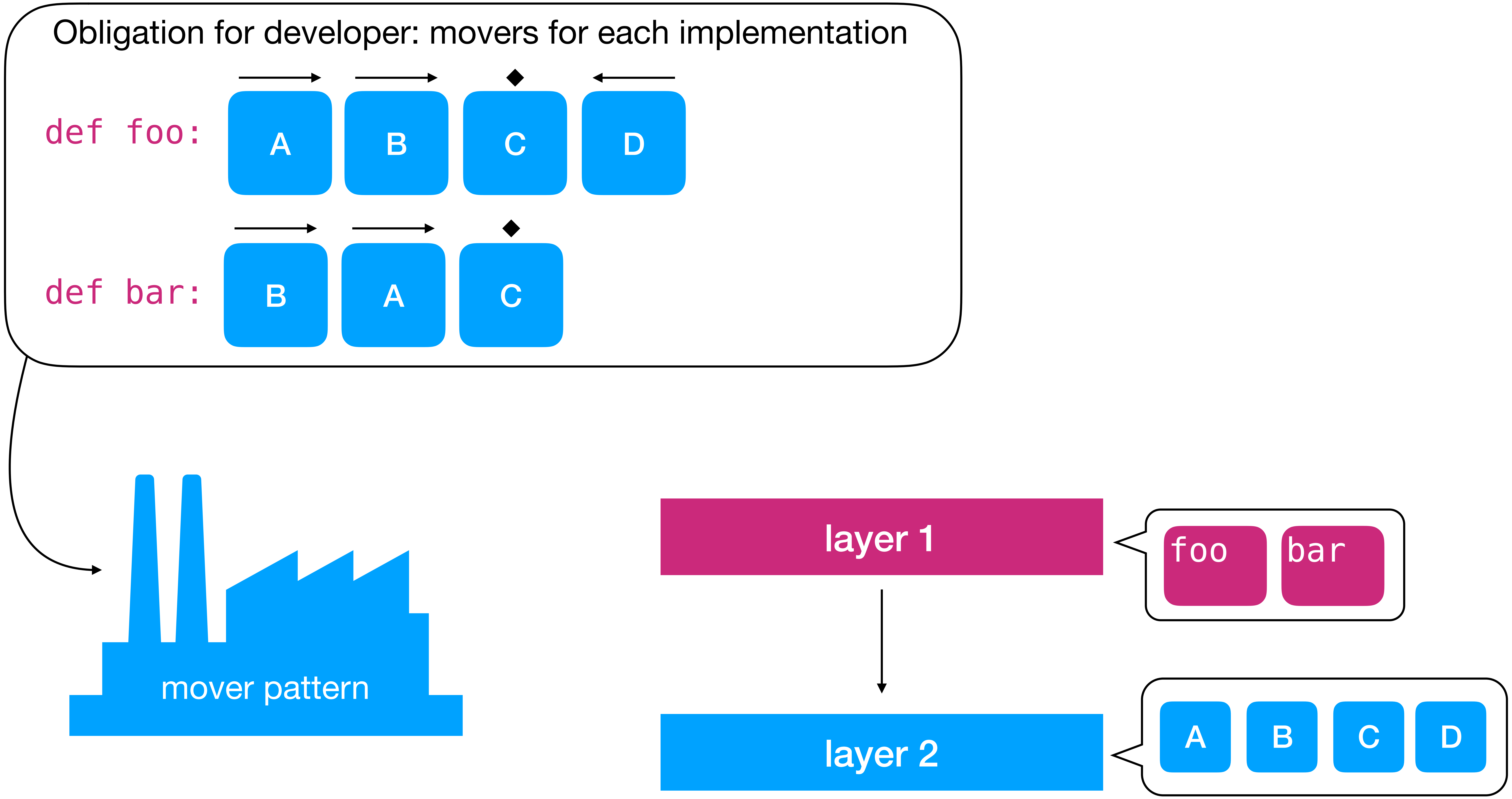
mover proof ✓

Movers are a layer proof *pattern*



Obligation for developer: movers for each implementation

Movers are a layer proof *pattern*



Obligation for developer: movers for each implementation

def foo: A → B → C ◆ D ←

def bar: B → A → C ◆

mover pattern

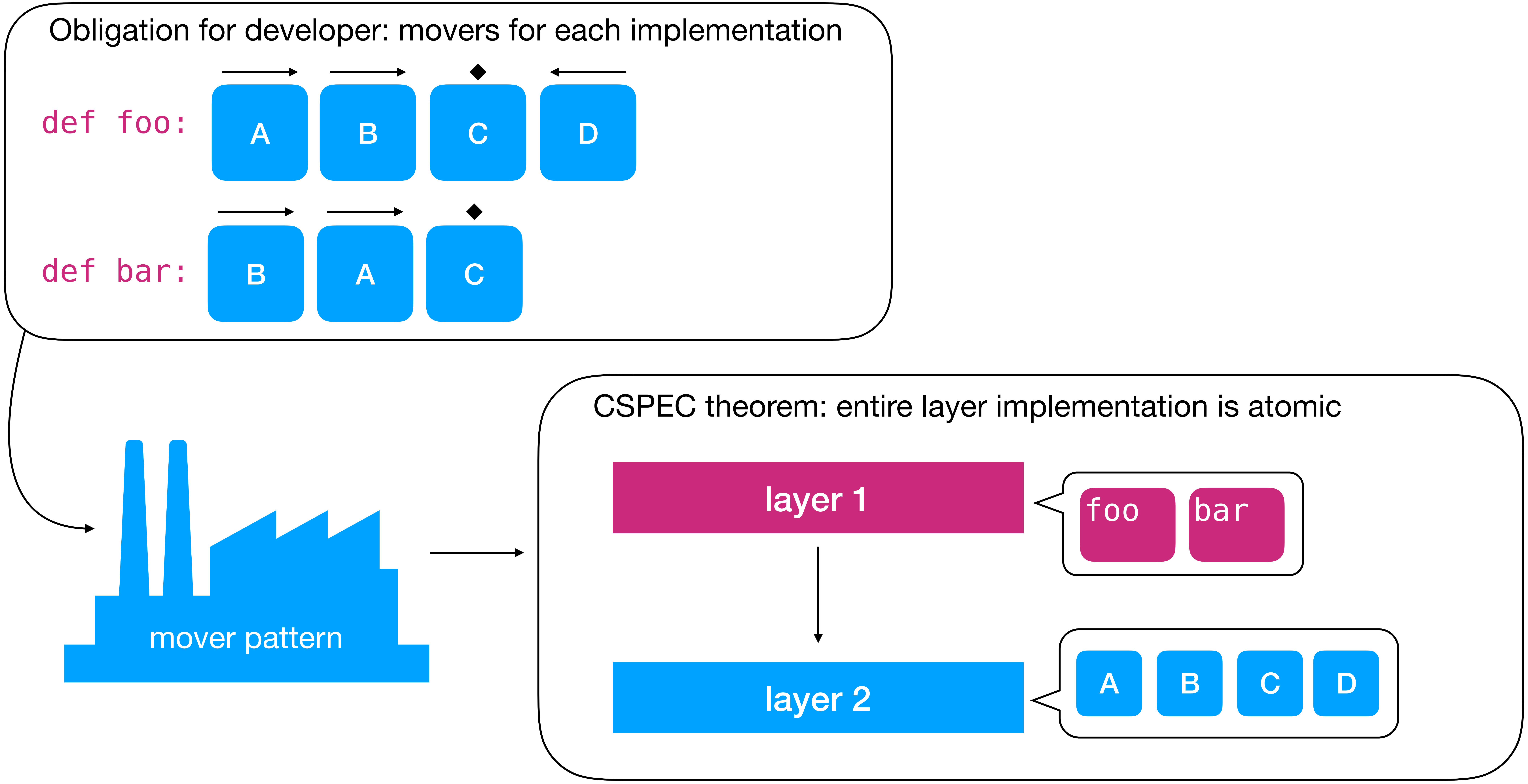
layer 1

foo bar

layer 2

A B C D

Movers are a layer proof *pattern*



Obligation for developer: movers for each implementation

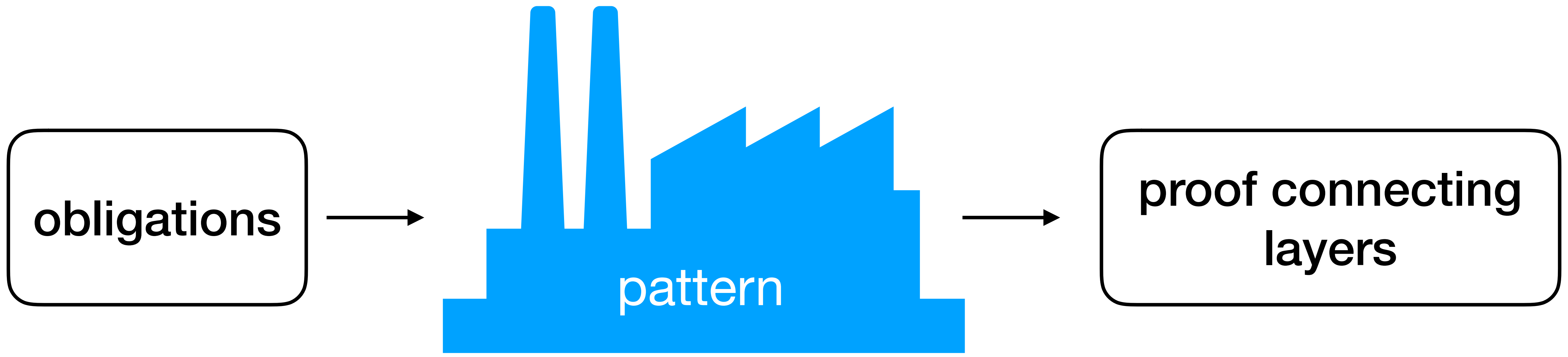
```
def foo:  A B C D
def bar:  B A C
```

CSPEC theorem: entire layer implementation is atomic

CSPEC provides other patterns to support mover reasoning

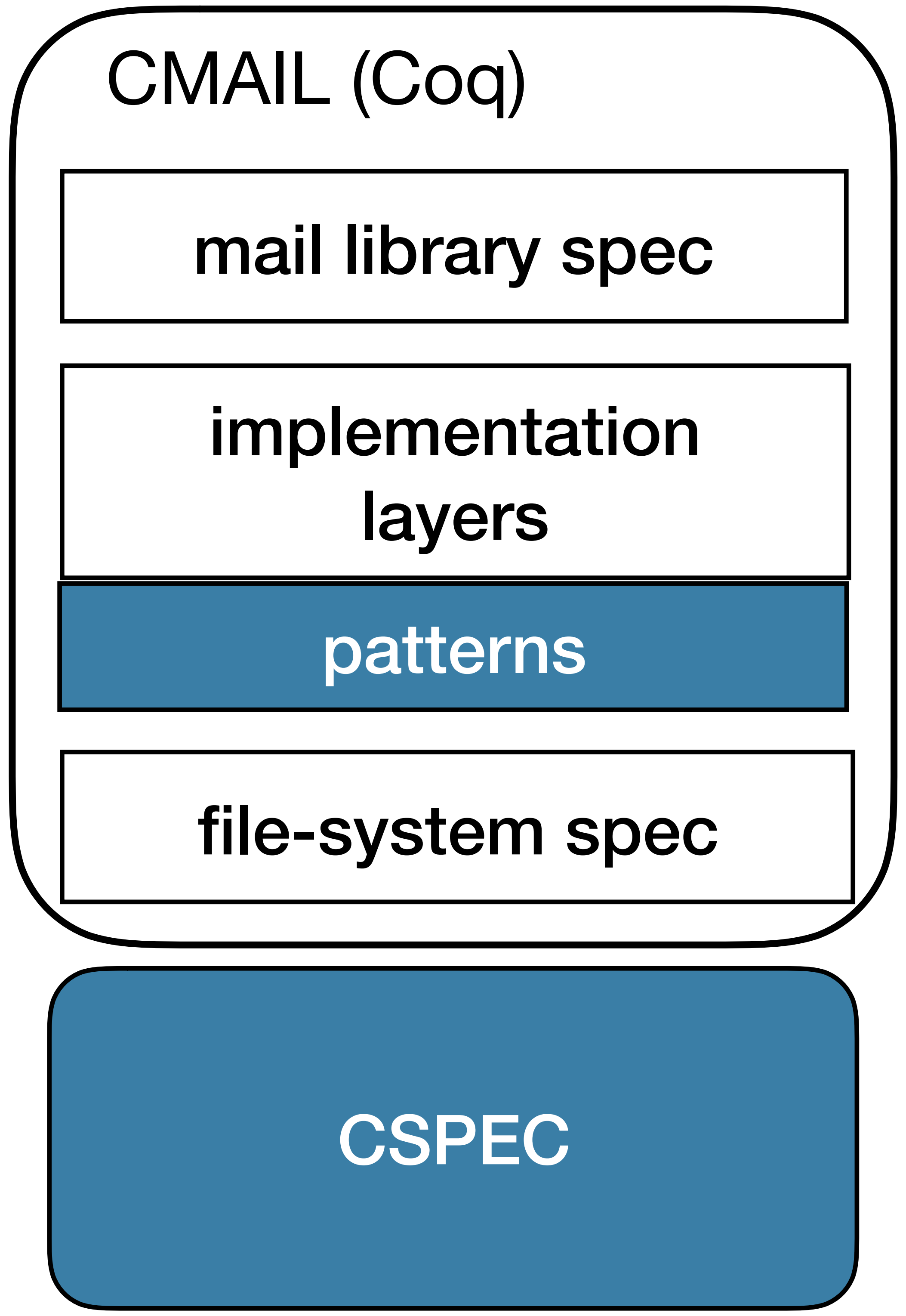
(see paper for details)

- Abstraction / forward simulation
  - Invariants
  - Error state
- Protocols
- Retry loops
- Partitioning



Using CSPEC to verify CMAIL

Coq



CMAIL (Coq)

mail library spec

implementation layers

patterns

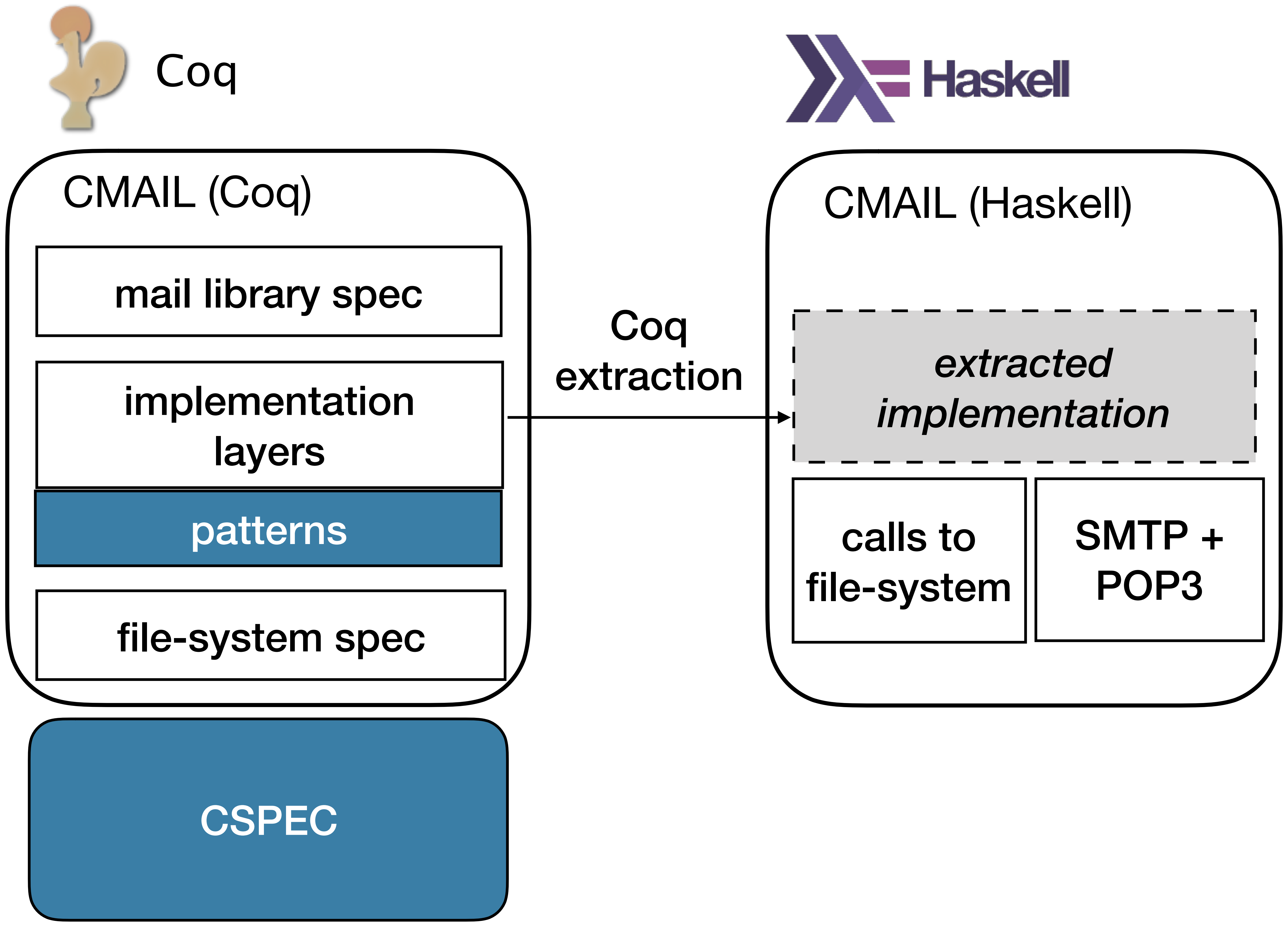
file-system spec

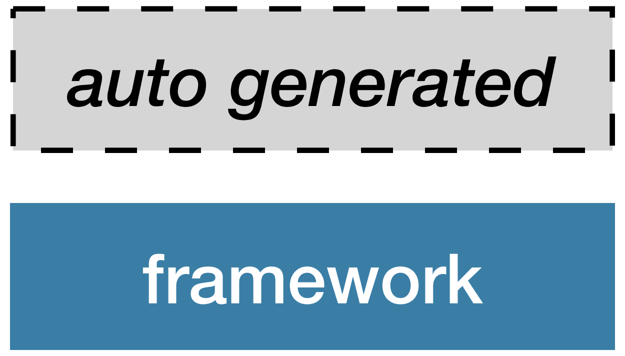
CSPEC

auto generated

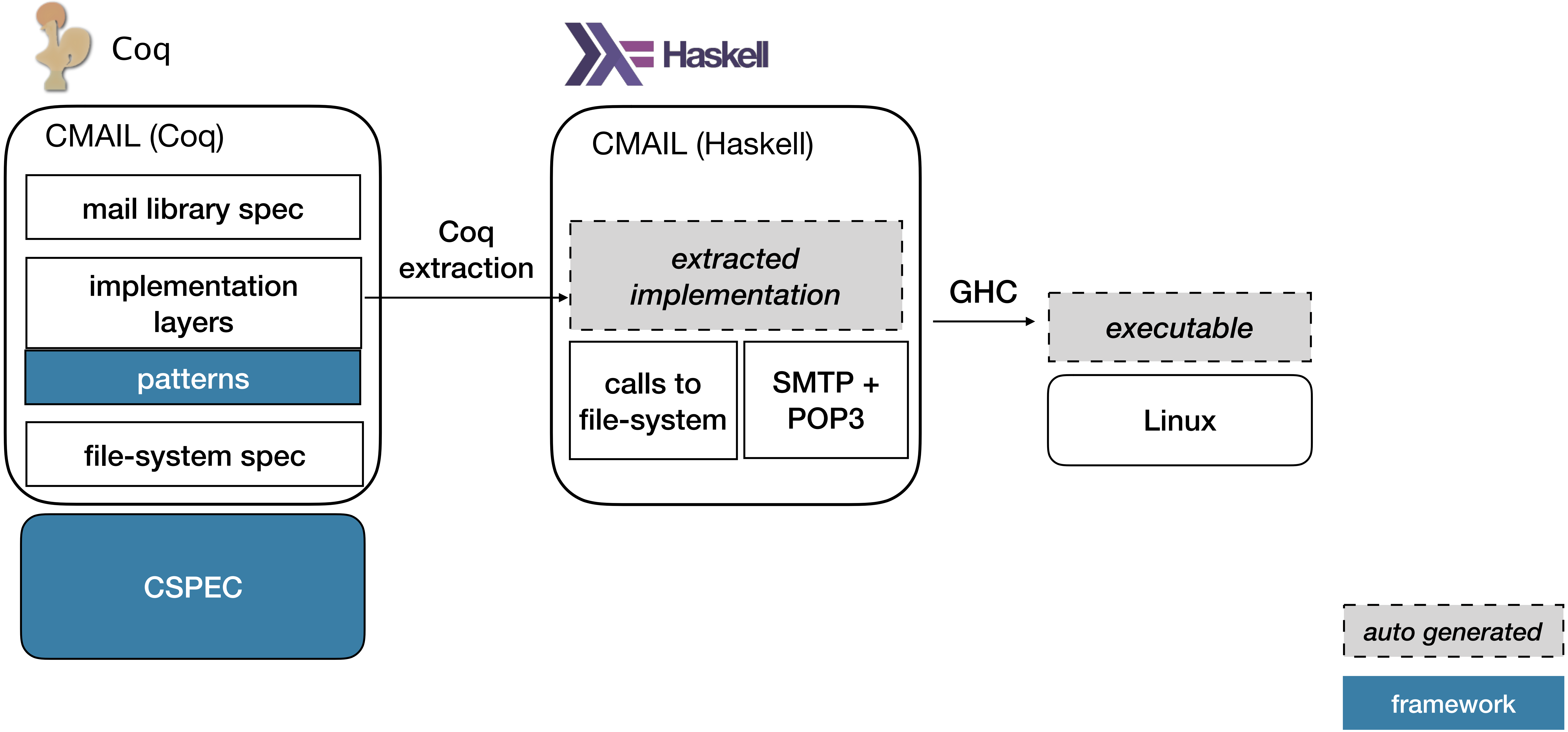
framework

Using CSPEC to verify CMAIL

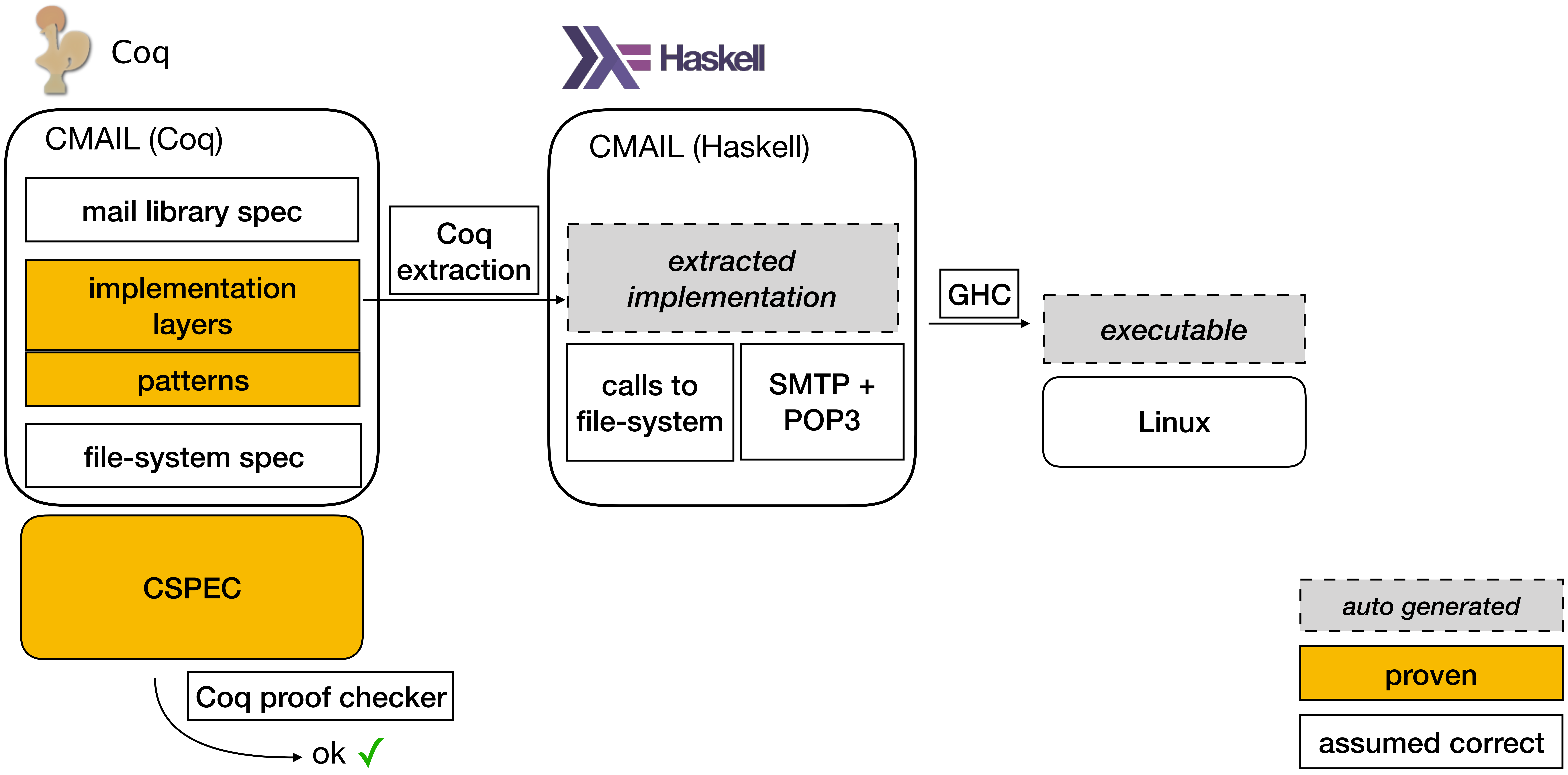




Using CSPEC to verify CMAIL



What is proven vs. assumed correct?



Coq

Haskell

CMAIL (Coq)

mail library spec

implementation layers

patterns

file-system spec

CSPEC

Coq proof checker

ok ✓

Coq extraction

CMAIL (Haskell)

extracted implementation

calls to file-system

SMTP + POP3

GHC

executable

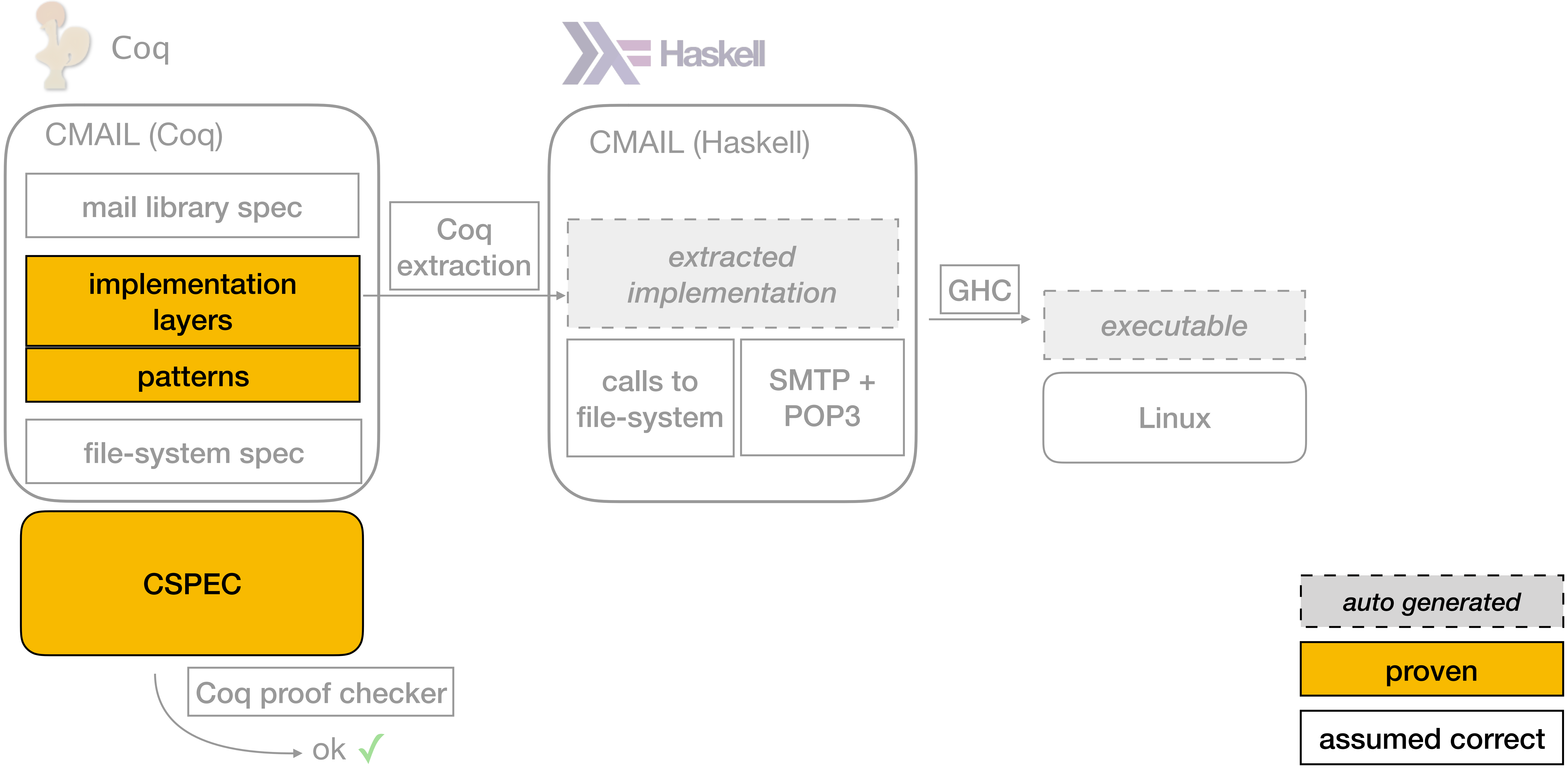
Linux

auto generated

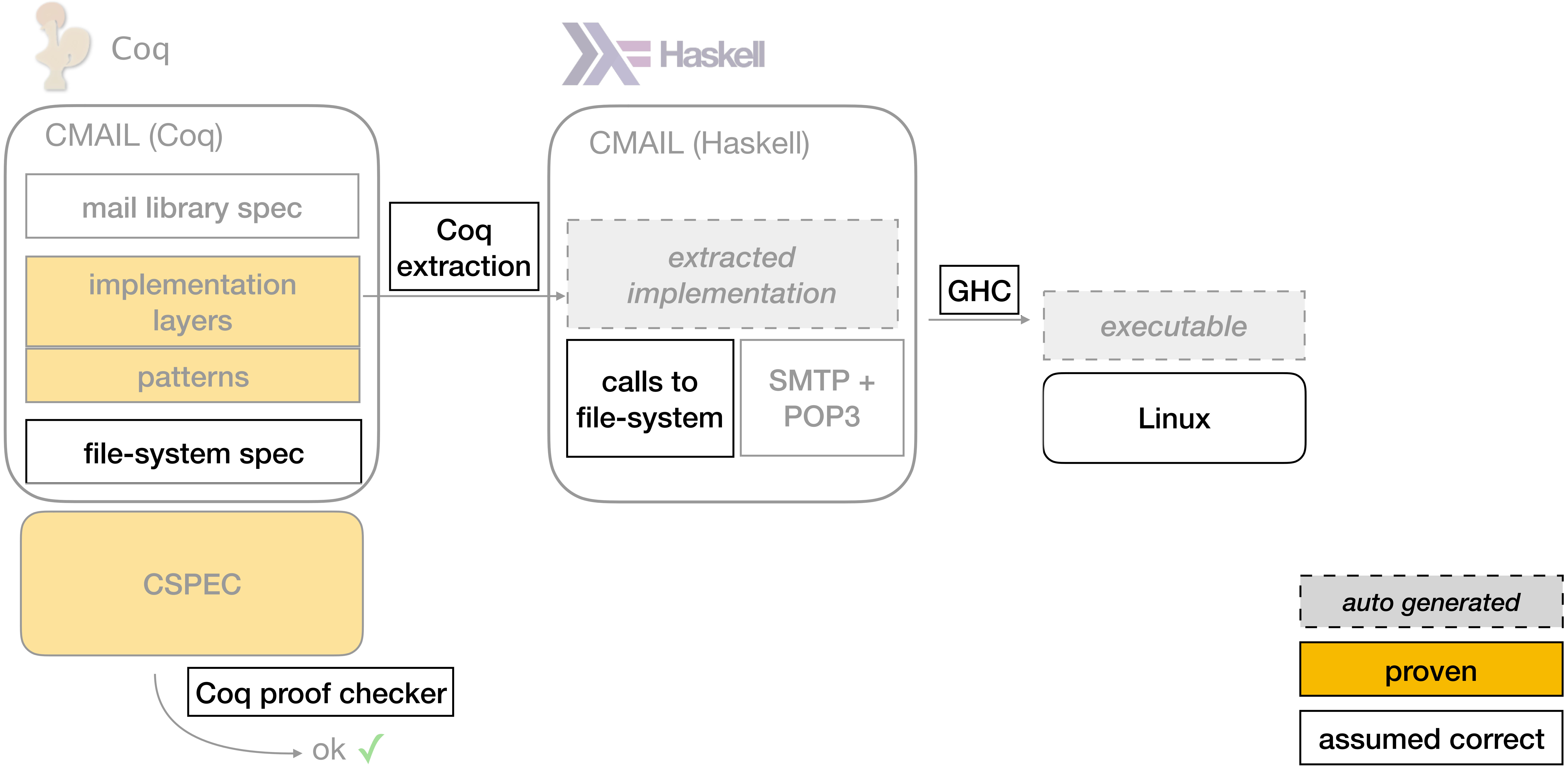
proven

assumed correct

Concurrency inside CMAIL is proven



Trust that the tools and OS are correct



Coq

Haskell

CMAIL (Coq)

mail library spec

implementation layers

patterns

file-system spec

CSPEC

Coq proof checker

ok ✓

Coq extraction

CMAIL (Haskell)

extracted implementation

calls to file-system

SMTP + POP3

GHC

executable

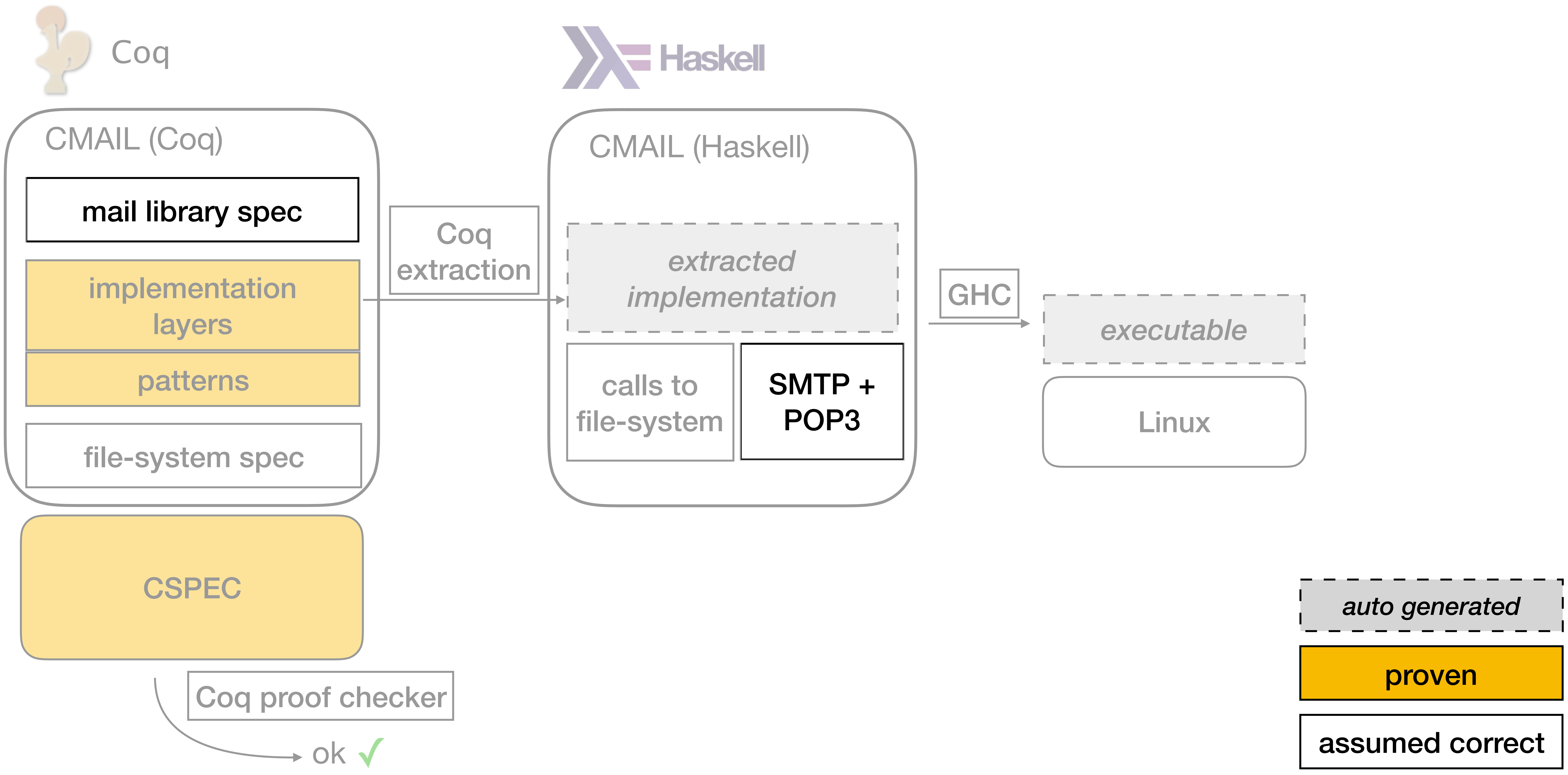
Linux

auto generated

proven

assumed correct

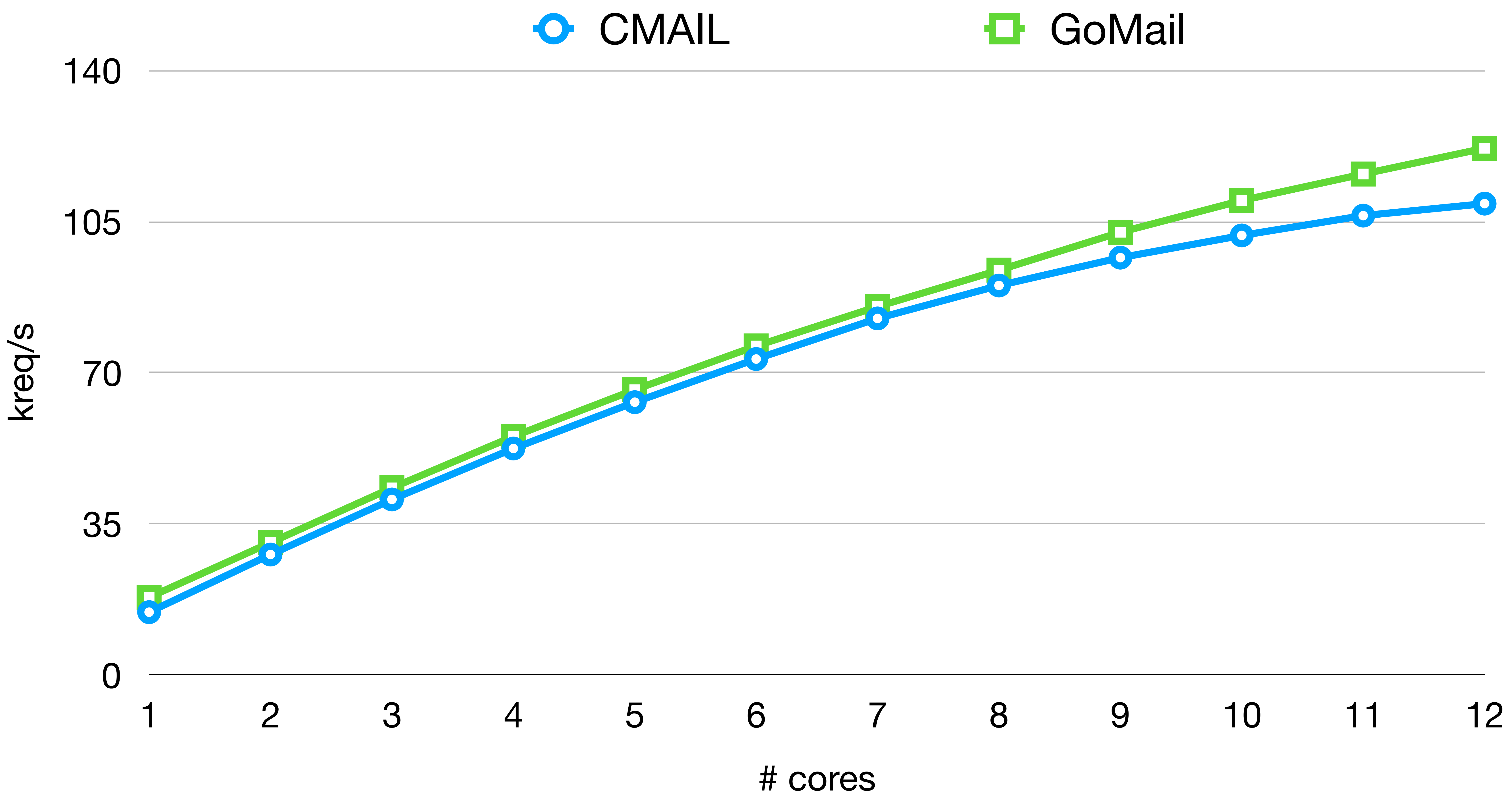
Mail server-specific assumptions



Evaluation

- Can CMAIL exploit file-system concurrency for speedup?
- How much effort was verifying CMAIL?
- What is the benefit of CSPEC's machine-checked proofs?

CMAIL achieves speedup with multiple cores



CMAIL was work but doable

		proof:code ratio
concurrent	**CMAIL**	**11.5x**
	CertiKOS	13.8x
	IronFleet	7.7x
sequential	**IronClad**	4.8x
	CompCert	4.6x

Took two authors 6 months

Machine-checked proofs give confidence in framework changes

Three anecdotes of changes to CSPEC:

Machine-checked proofs ensure soundness of entire system

Machine-checked proofs give confidence in framework changes

Three anecdotes of changes to CSPEC:

- Implemented **partitioning pattern** to support multiple users

Machine-checked proofs ensure soundness of entire system

Machine-checked proofs give confidence in framework changes

Three anecdotes of changes to CSPEC:

- Implemented **partitioning pattern** to support multiple users
- Improved **mover pattern** for a CMAIL left mover proof

Machine-checked proofs ensure soundness of entire system

Machine-checked proofs give confidence in framework changes

Three anecdotes of changes to CSPEC:

- Implemented **partitioning pattern** to support multiple users
- Improved **mover pattern** for a CMAIL left mover proof
- Implemented **error-state pattern** for the x86-TSO lock proof

Machine-checked proofs ensure soundness of entire system

CSPEC is a framework for verifying concurrency in systems software

- Layers and patterns (esp. movers) make proofs manageable
- Machine-checked framework supports adding new patterns
- Evaluated by verifying mail server and x86-TSO lock

github.com/mit-pdos/cspec

CSPEC is a framework for verifying concurrency in systems software

- Layers and patterns (esp. movers) make proofs manageable
- Machine-checked framework supports adding new patterns
- Evaluated by verifying mail server and x86-TSO lock

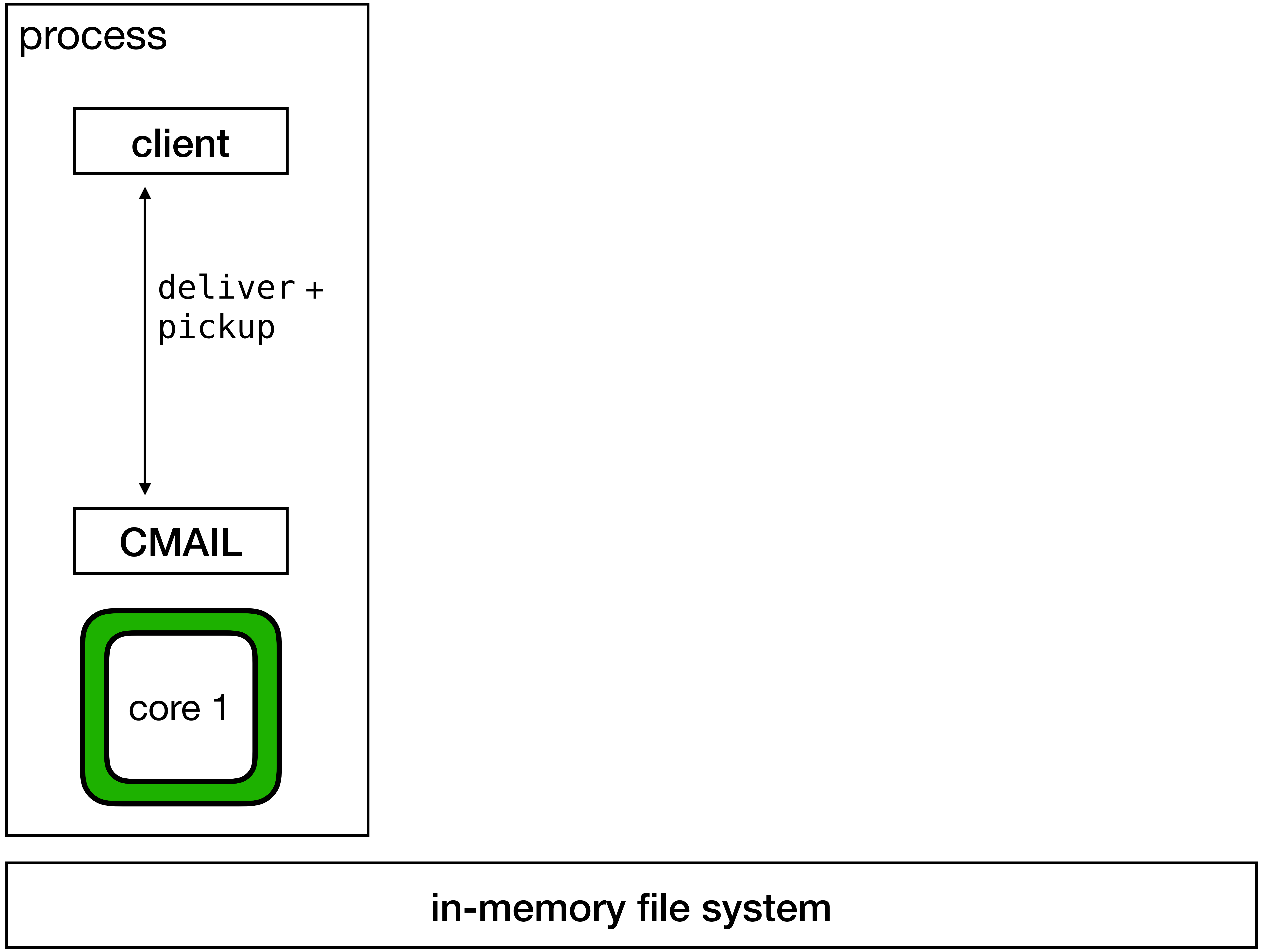
github.com/mit-pdos/cspec

poster #1

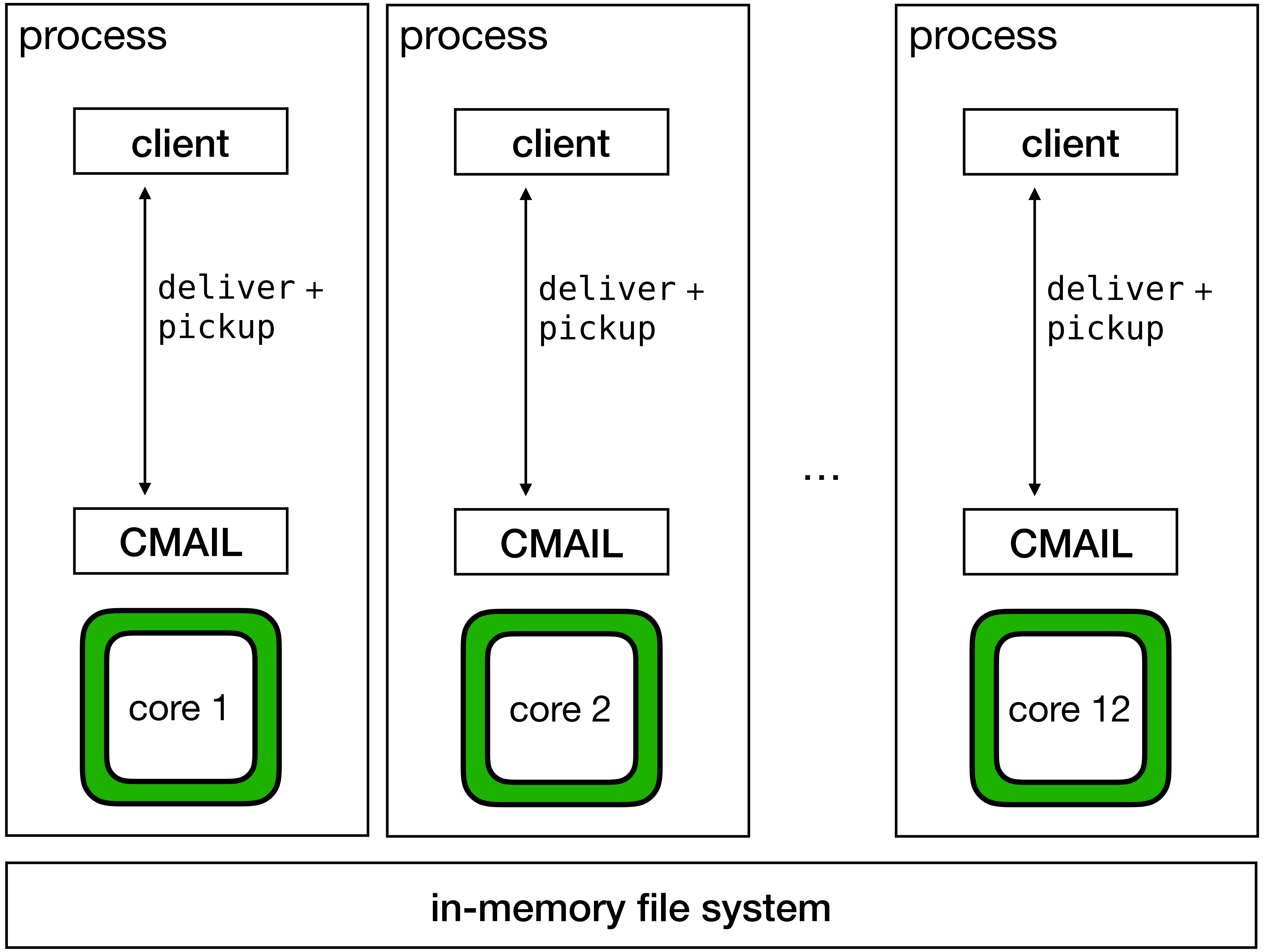
Backup slides

CMAIL perf experimental setup

Performance experiment setup for CMAIL



Performance experiment setup for CMAIL


process
client
deliver + pickup
CMAIL
core 1
process
client
deliver + pickup
CMAIL
core 2
...
process
client
deliver + pickup
CMAIL
core 12
in-memory file system