Proofcraft

seL4® Multikernel Roadmap and Concurrency Verification

Corey Lewis @ Proofcraft

The world's most highly assured operating system kernel*

The world's most highly assured operating system kernel*

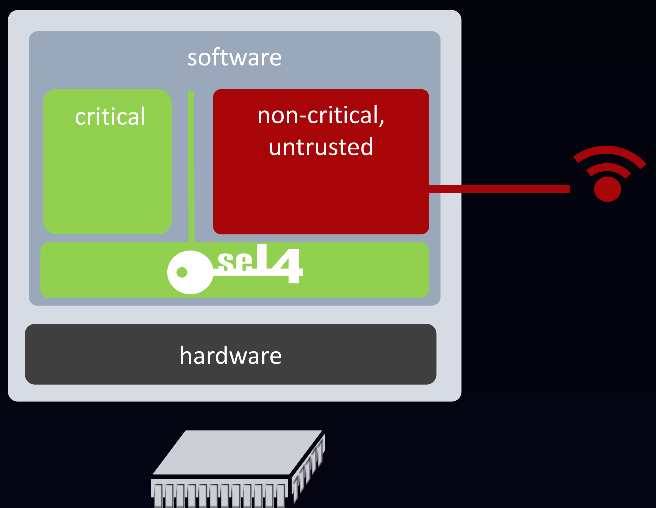
* only when running on a single core

The world's most highly assured operating system kernel*

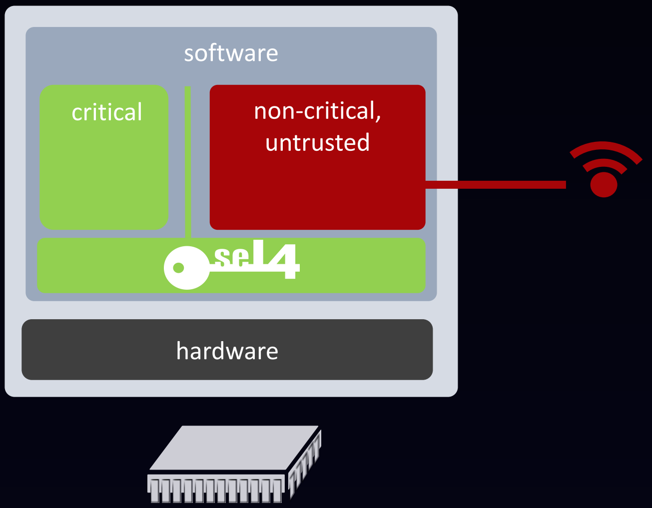
* when running sequentially, without interference

```
void kernel_call () {
        ...
        ...
        ...
}
```

✓erified

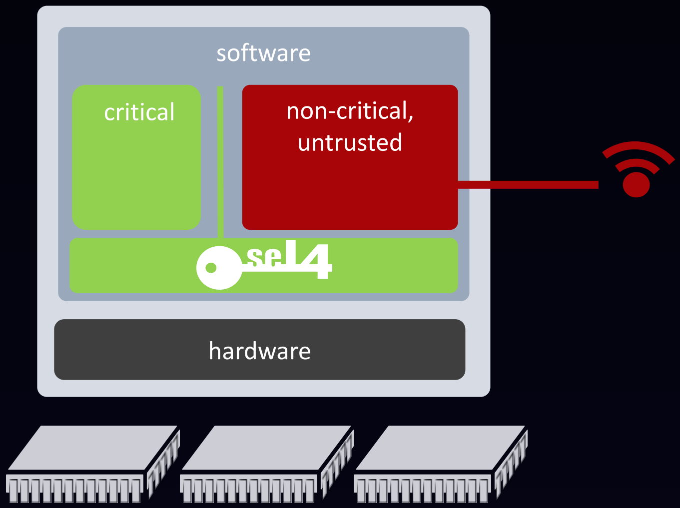


What do we want?



What do we want?

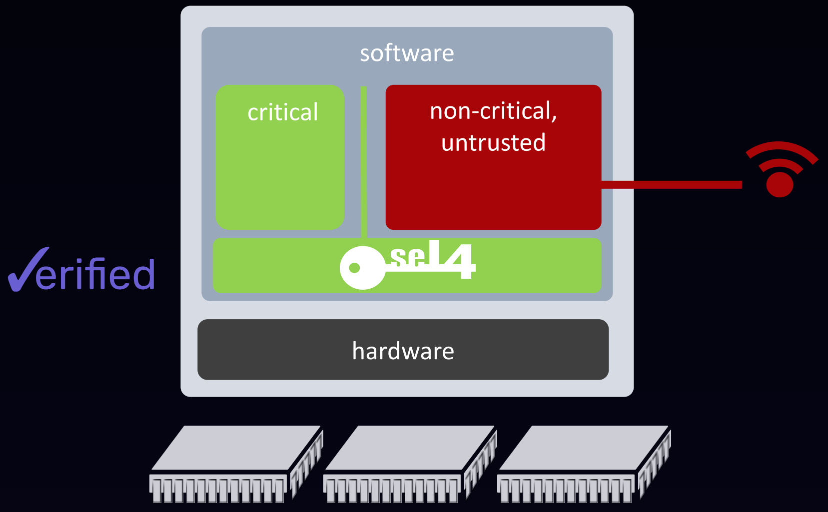
Better performance,
by using more cores



What do we want?

Better performance,
by using more cores

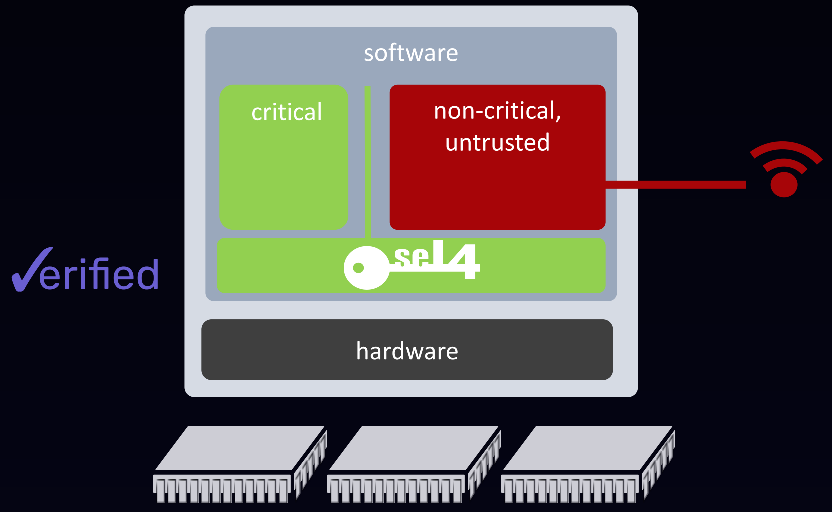
Still high assurance



What do we want?

Better performance, by using more cores → Concurrency

Still high assurance → Formal Verification

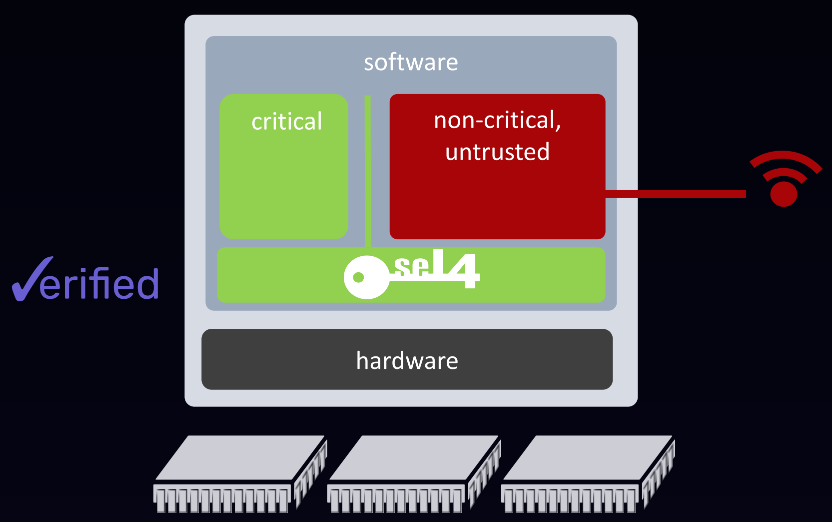


What do we want?

Better performance, by using more cores → Concurrency

Still high assurance → Formal Verification

Concurrency + Formal Verification → Very hard!



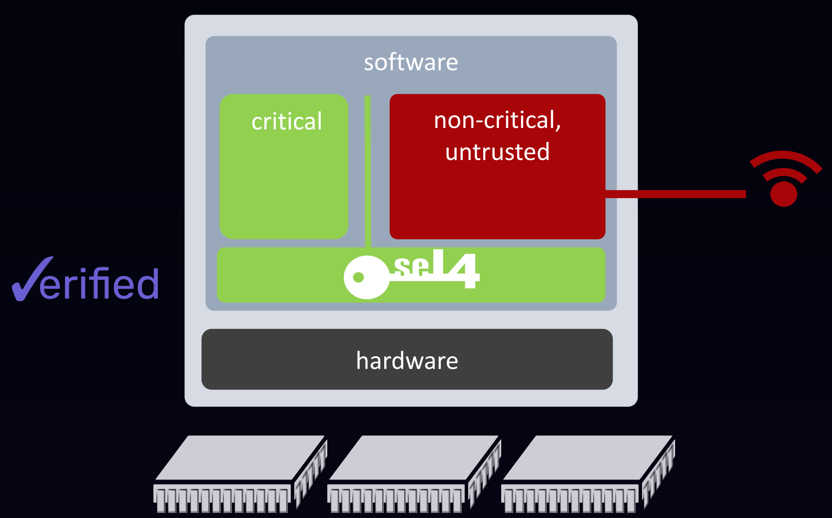
What do we want?

Better performance, by using more cores → Concurrency

Still high assurance → Formal Verification

Concurrency + Formal Verification → Very hard!

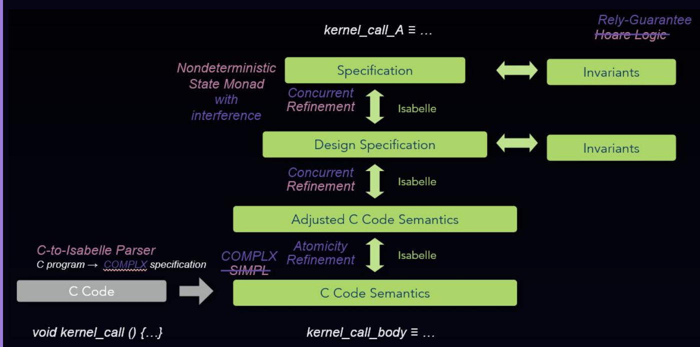
Goal:
Allow use of multiple cores as soon as possible,
with incrementally stronger and stronger assurance



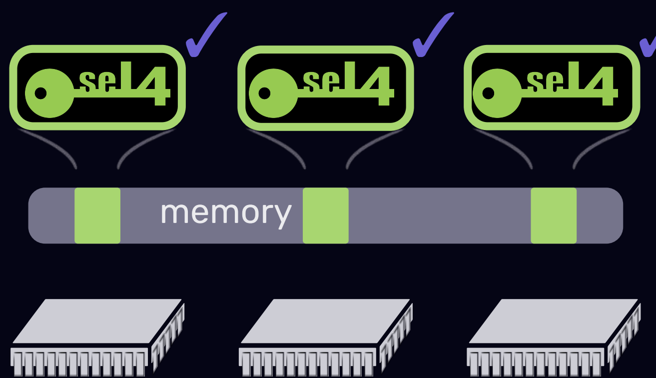
Overview

Goal:
Allow use of multiple cores as soon as possible,
With incrementally stronger and stronger assurance

What's hard?
What have we got so far?



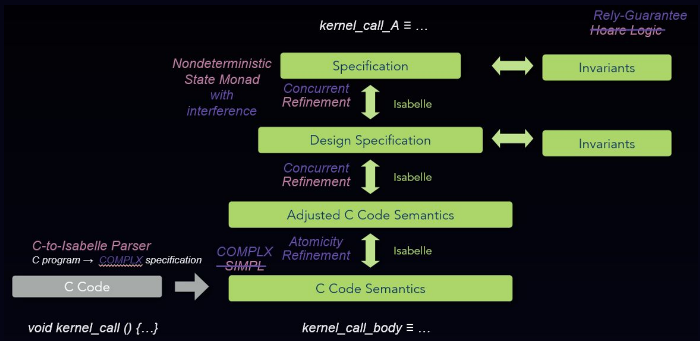
Towards a verified static multikernel seL4



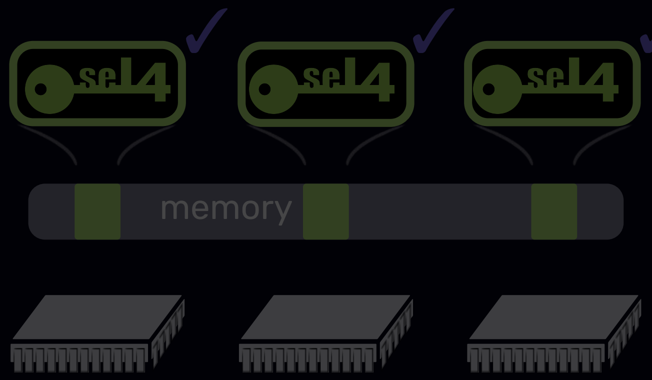
Overview

Goal:
Allow use of multiple cores as soon as possible,
With incrementally stronger and stronger assurance

What's hard?
What have we got so far?



Towards a verified static multikernel seL4



What’s hard?

Concurrency

Formal Verification

Very hard!

Isn’t that solved?

There exist approaches for concurrency verification
that work for small / self-contained algorithms

But:

What's hard?

Concurrency

Formal Verification

Very hard!

Isn't that solved?

There exist approaches for concurrency verification
that work for small / self-contained algorithms

But:

seL4 is neither small nor high-level nor modular
(because it's a microkernel and it is fast)

What's hard?

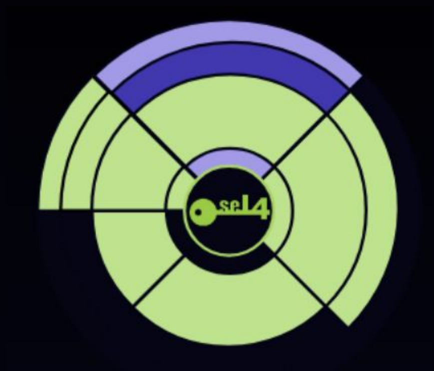
Plus:

seL4's existing verification framework is complex
(because it's doing formal proof of low-level complex code)

What's hard?

Plus:

seL4's existing verification framework is complex
(because it's doing formal proof of low-level complex code)

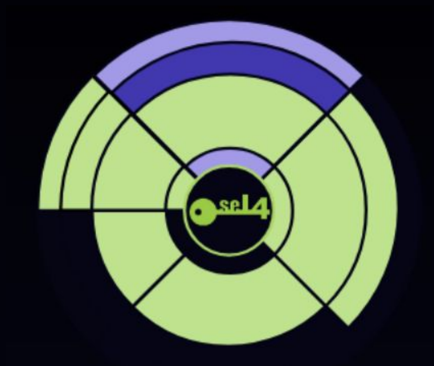


- > 1 million lines of proof
  - Developed over 15 years
- Three levels of specifications
  - Two very different specification languages
  - Needs to capture a lot of detail
- Many different configurations
  - Multiple architectures, multiple features, MCS

What's hard?

Plus:

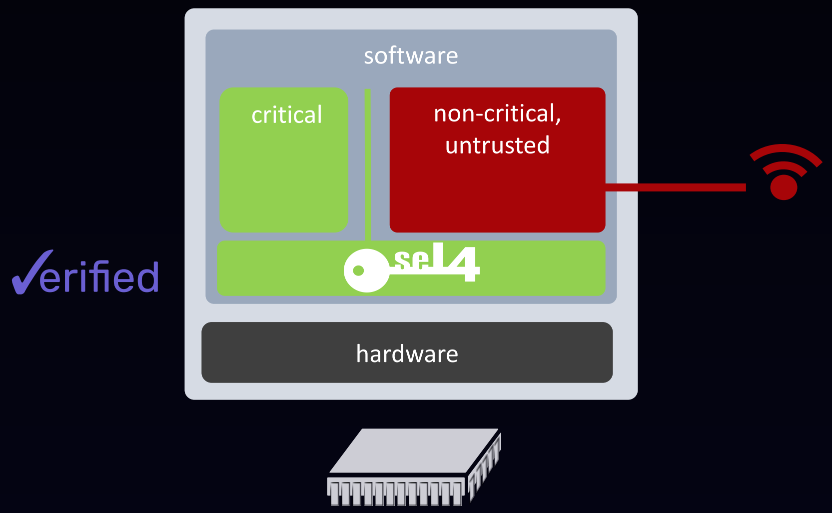
seL4's existing verification framework is complex
(because it's doing formal proof of low-level complex code)



- > 1 million lines of proof
  - Developed over 15 years
- Three levels of specifications
  - Two very different specification languages
  - Needs to capture a lot of detail
- Many different configurations
  - Multiple architectures, multiple features, MCS

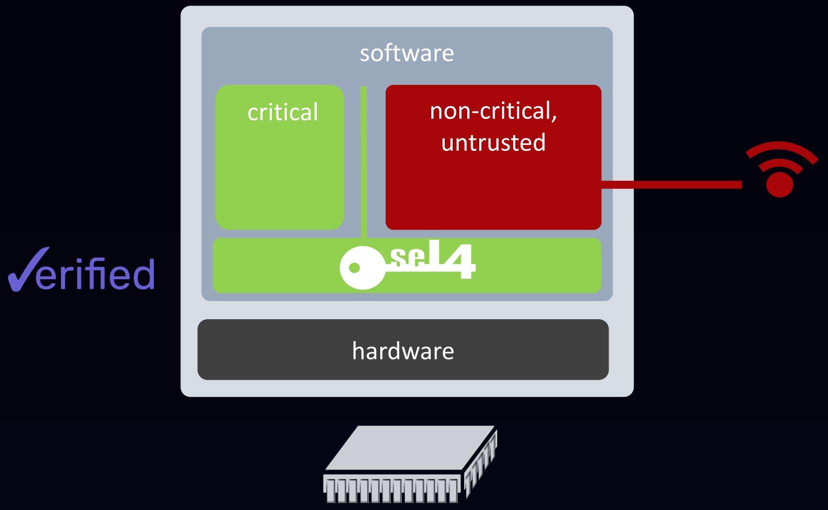
We want to maximise reuse of existing proofs

The unicore situation



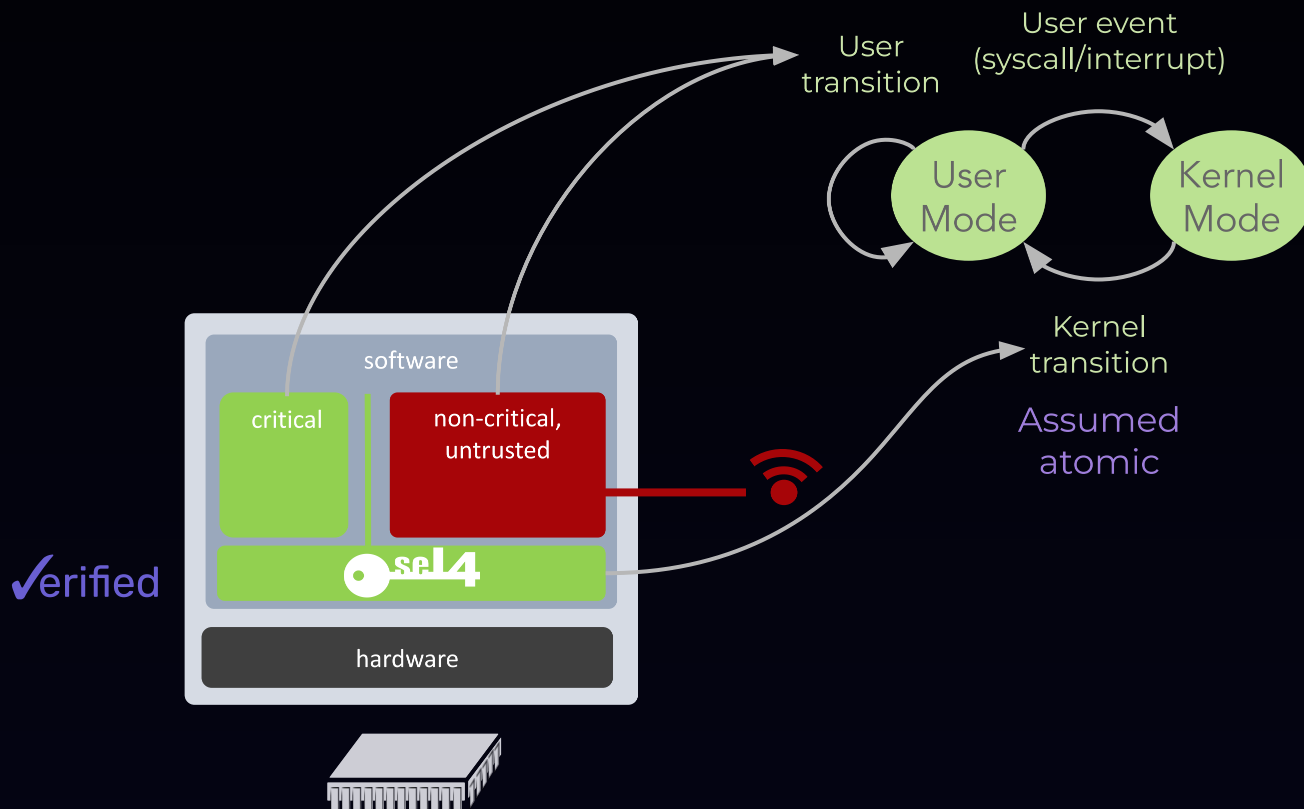
The unicore situation

Verified = the C code is correct (w.r.t its specification)
(+security, binary, etc. Ignored here for simplicity)



The unicore situation

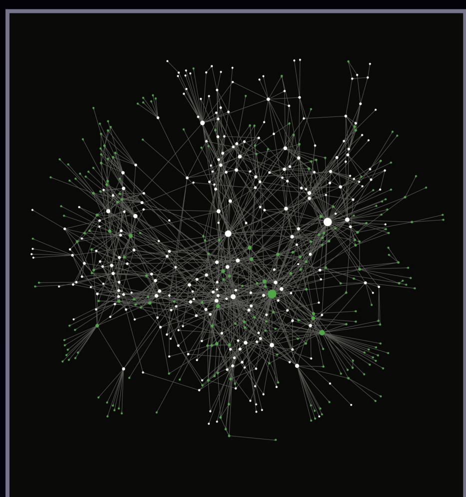
Verified = the C code is correct (w.r.t its specification)
(+security, binary, etc. Ignored here for simplicity)



The unicore situation

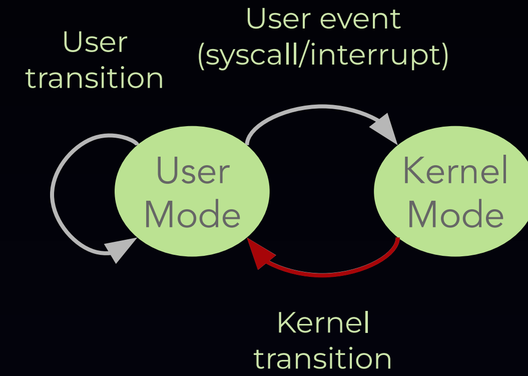
Verified = the C code is correct (w.r.t its specification)
(+security, binary, etc. Ignored here for simplicity)

~10,000 LOC
>500 functions



void kernel_call () {
…
…
…
}

C Code

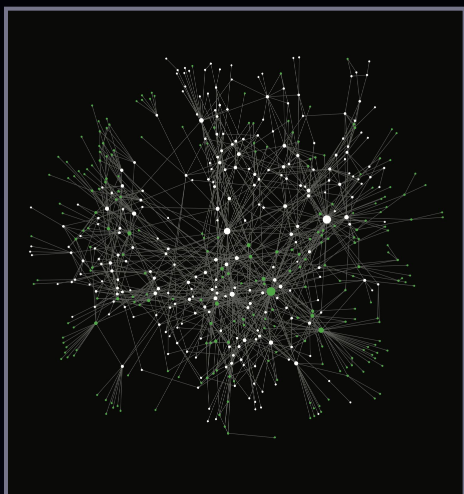


Assumed atomic

The unicore situation

Verified = the C code is correct (w.r.t its specification)
(+security, binary, etc. Ignored here for simplicity)

~10,000 LOC
>500 functions



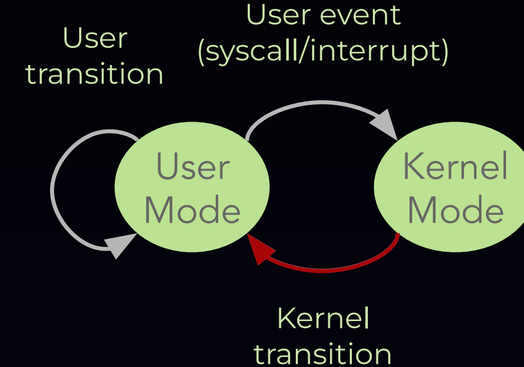
void kernel_call () {
...
...
...
}

Functional Correctness

Specification

↕

C Code

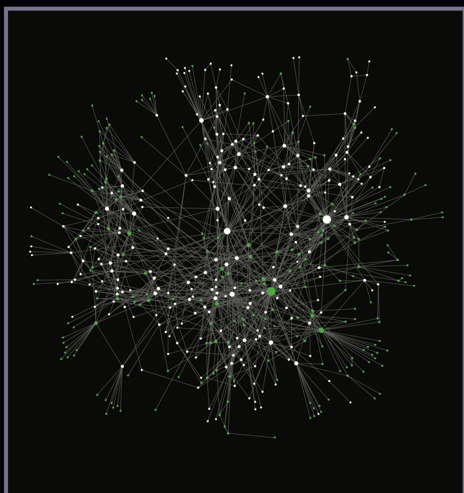


Assumed atomic

The multicore situation

~10,000 LOC

>500 functions



void kernel_call () {

…

…

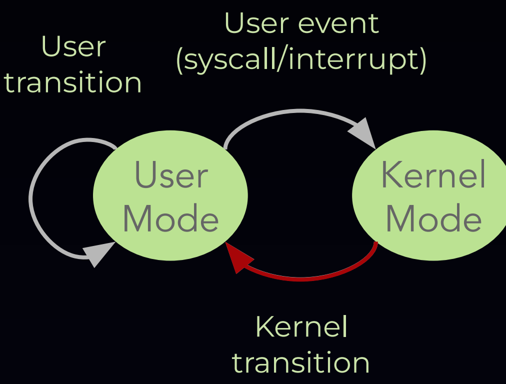
…

}

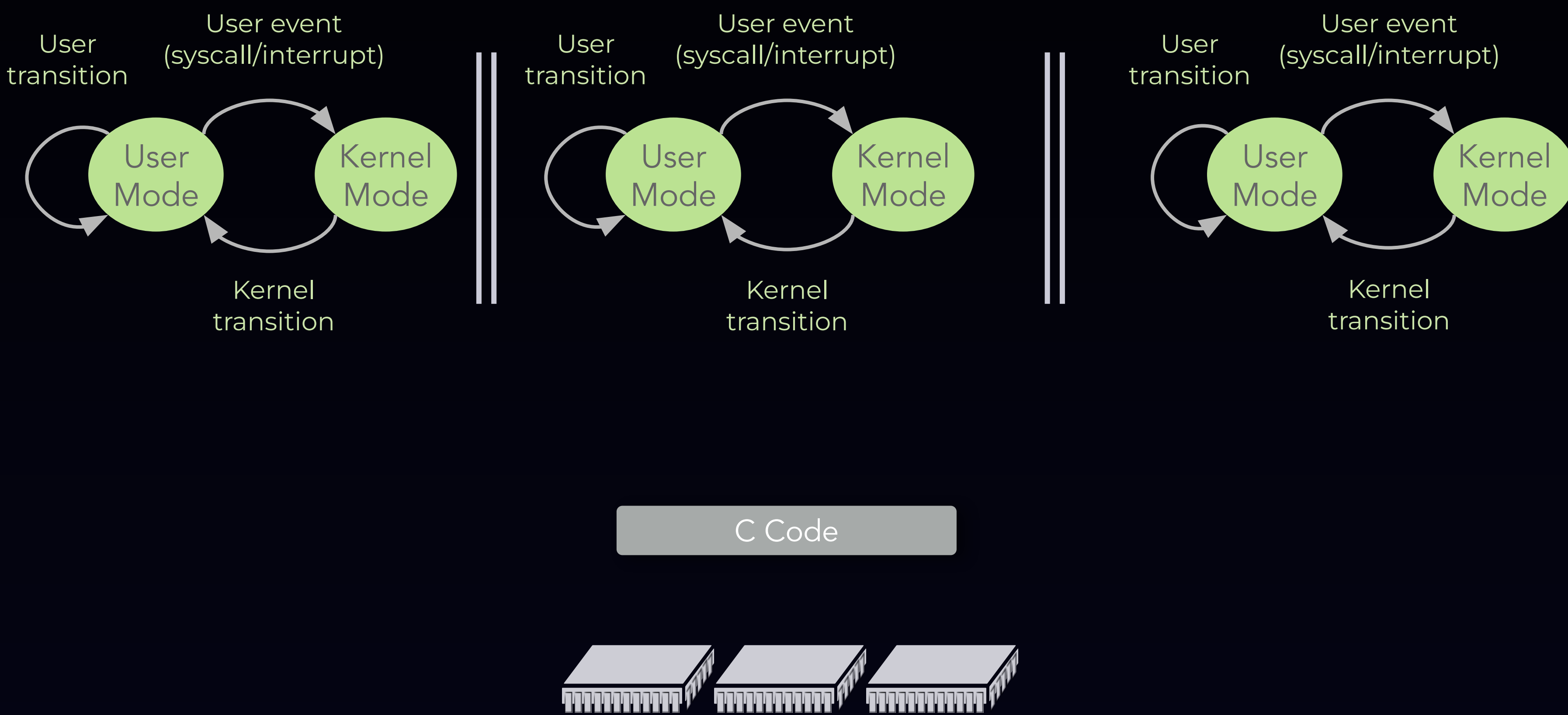
Functional Correctness

Specification

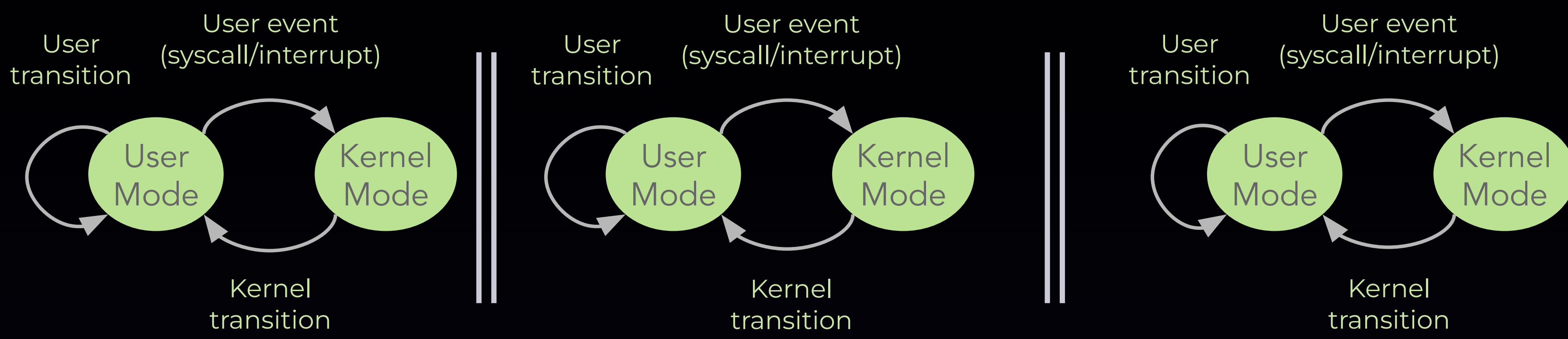
C Code



The multicore situation

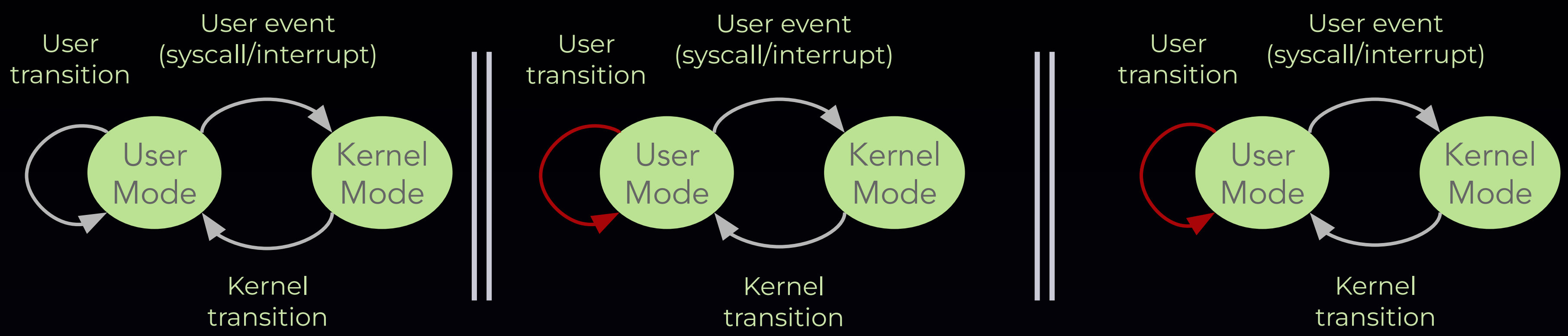


The multicore situation



Introduces three types of concurrency

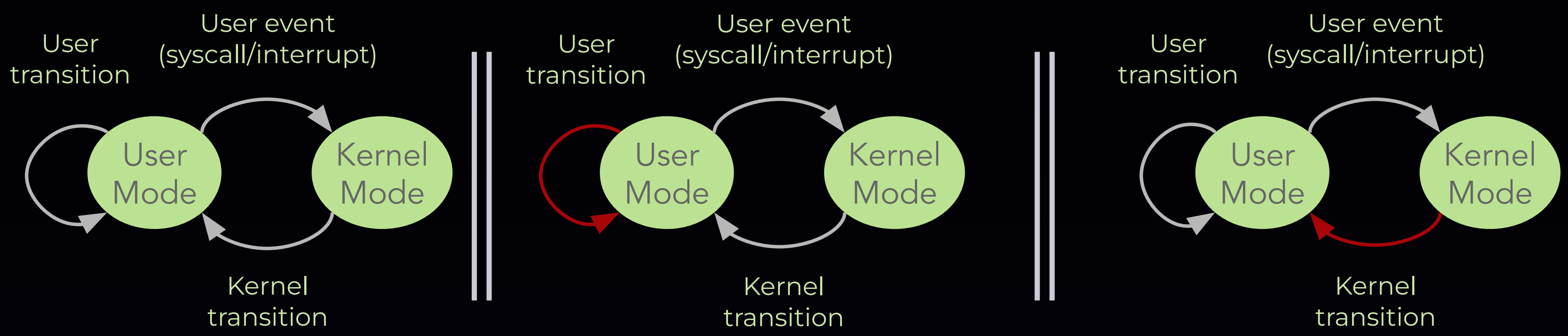
The multicore situation



Introduces three types of concurrency

1. User and User
   - Part of overall system design
   - Out of scope of kernel verification
   - Must reason about this for whole-system proofs

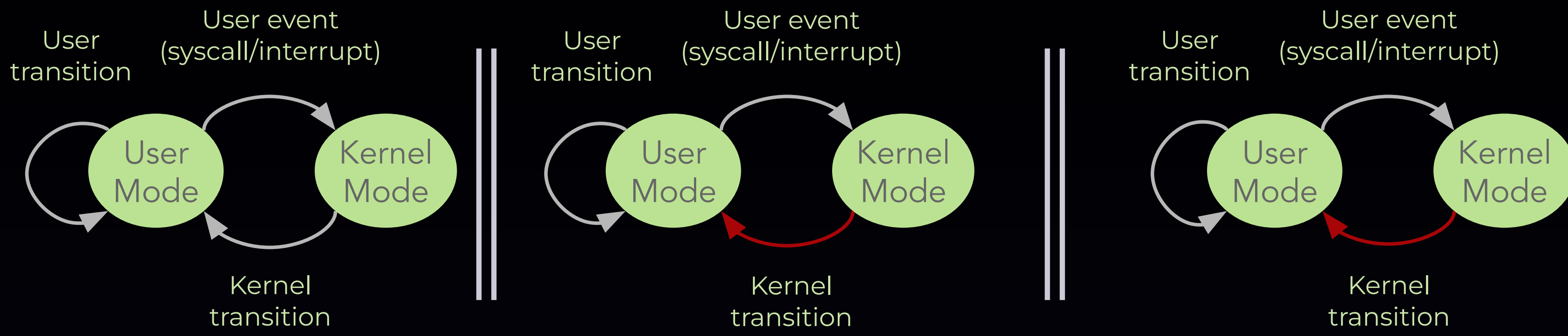
The multicore situation



Introduces three types of concurrency

2. User and Kernel
   - Must prove that the kernel does not depend on what the user has access to

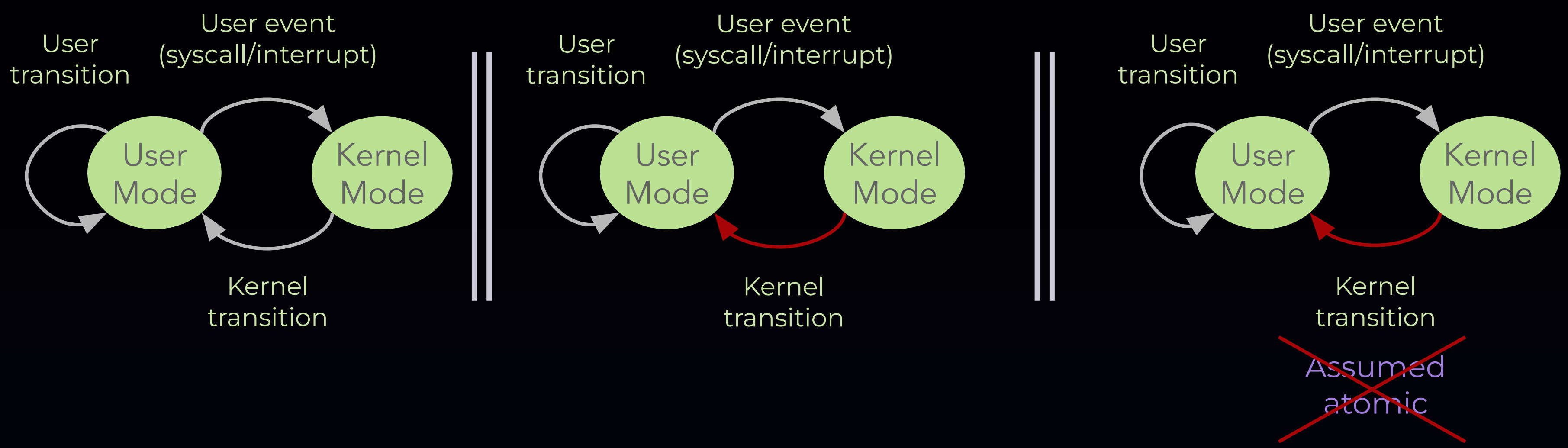
The multicore situation



Introduces three types of concurrency

3. Kernel and Kernel
   - Must prove that the kernel itself correctly handles this
   - SMP seL4 does this with locks, the static multikernel uses separation of resources

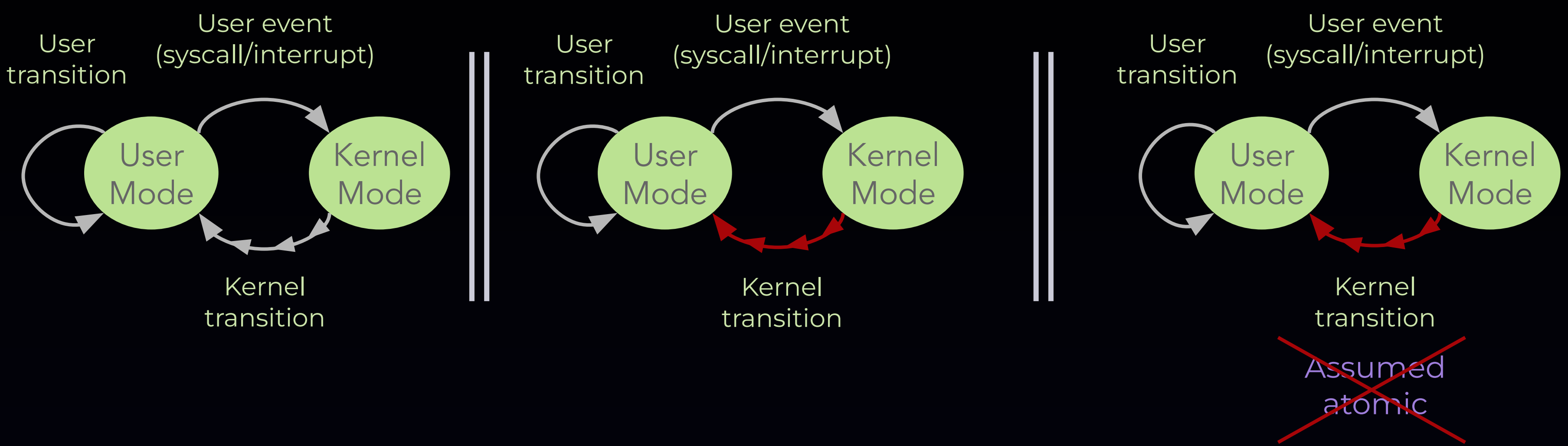
The multicore situation



Introduces three types of concurrency

3. Kernel and Kernel
   - Must prove that the kernel itself correctly handles this
   - SMP seL4 does this with locks, the static multikernel uses separation of resources

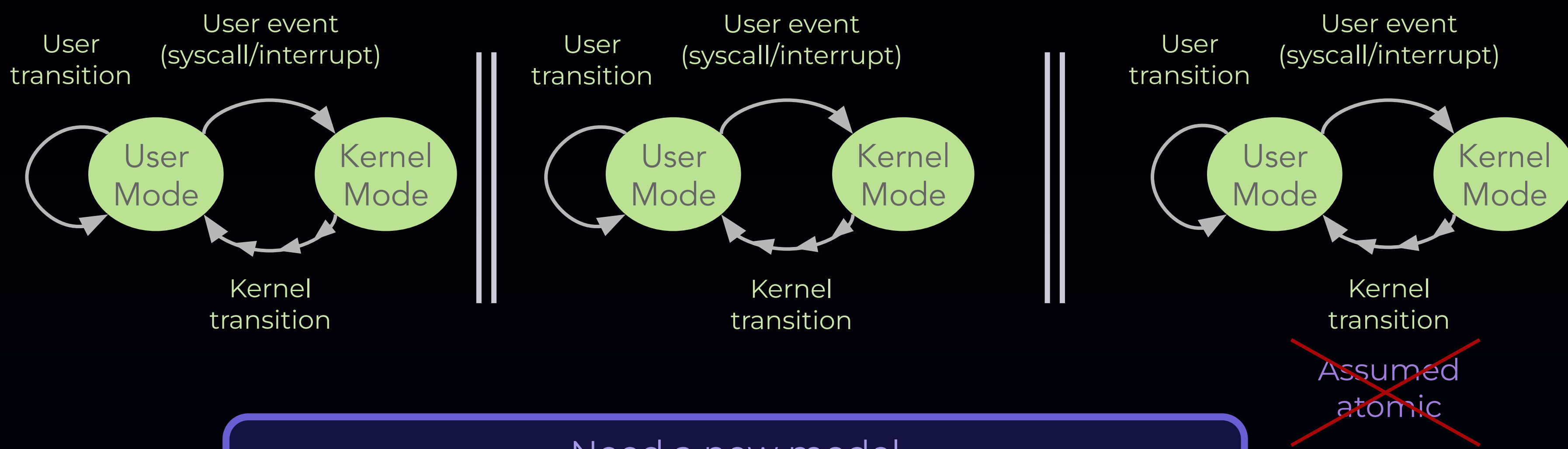
The multicore situation



Introduces three types of concurrency

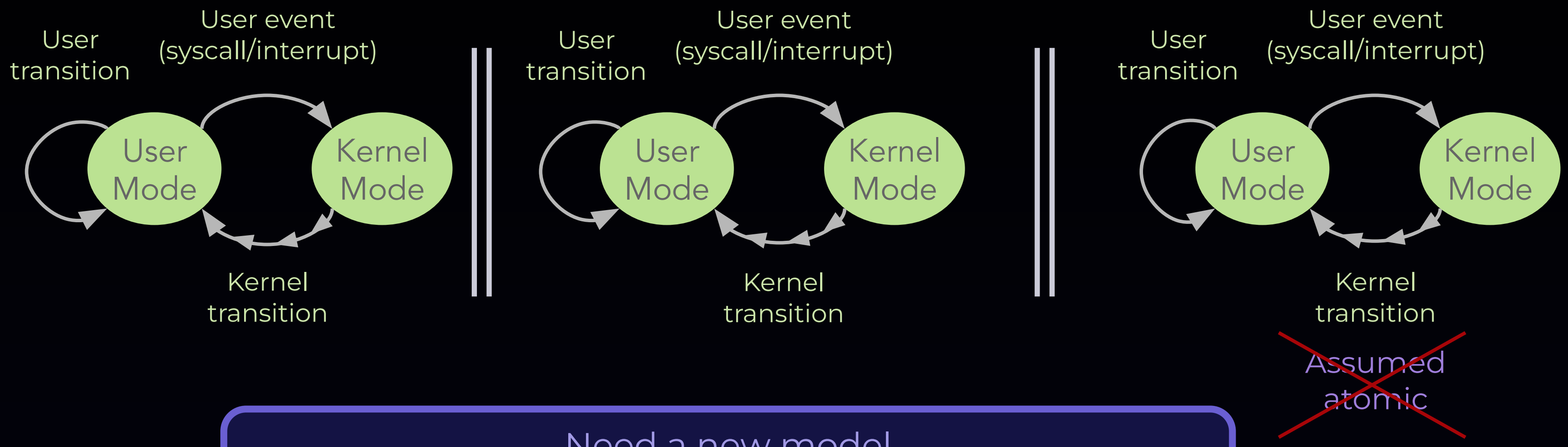
3. Kernel and Kernel
   - Must prove that the kernel itself correctly handles this
   - SMP seL4 does this with locks, the static multikernel uses separation of resources

The multicore situation



Need a new model
and verification framework

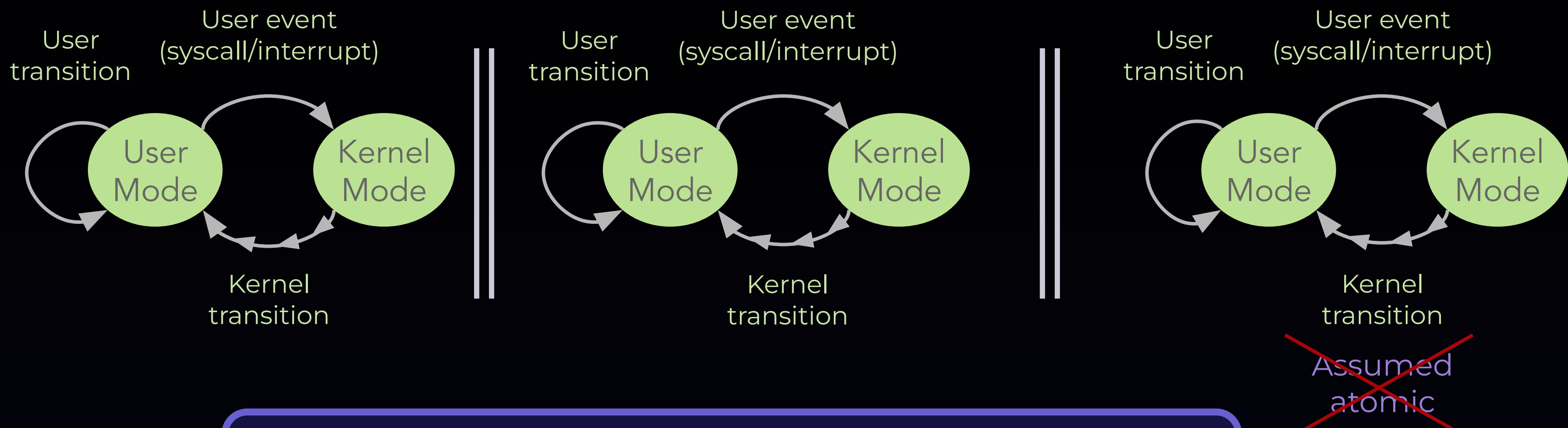
The multicore situation



Need a new model
and verification framework

We want to maximise reuse of existing sequential proofs
where concurrency is controlled

The multicore situation

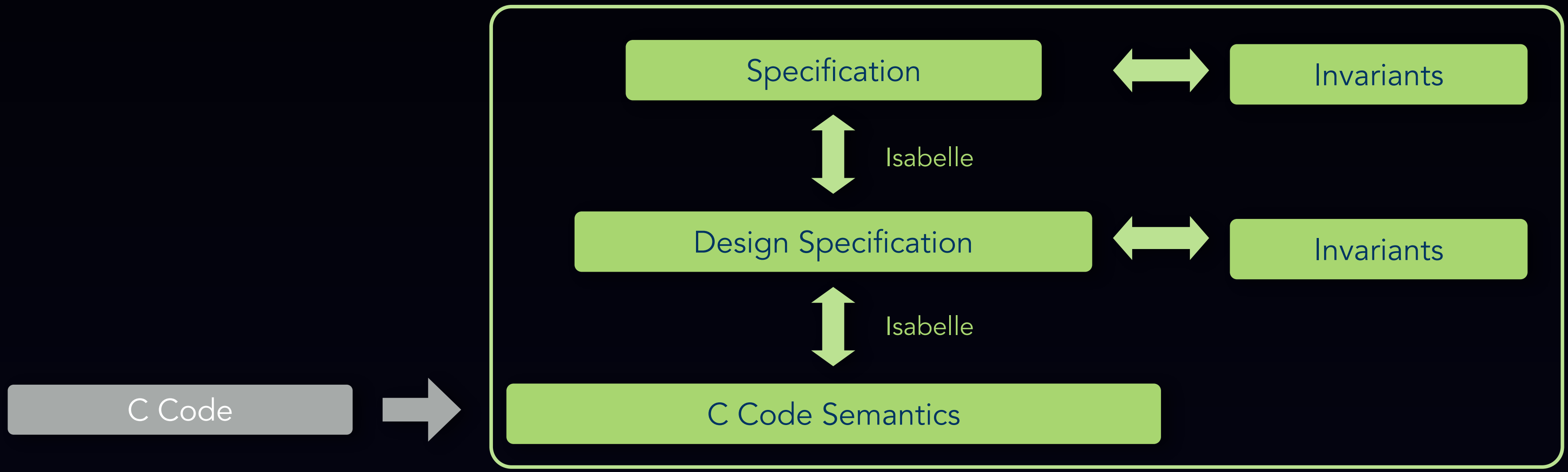


Need a new model
and verification framework

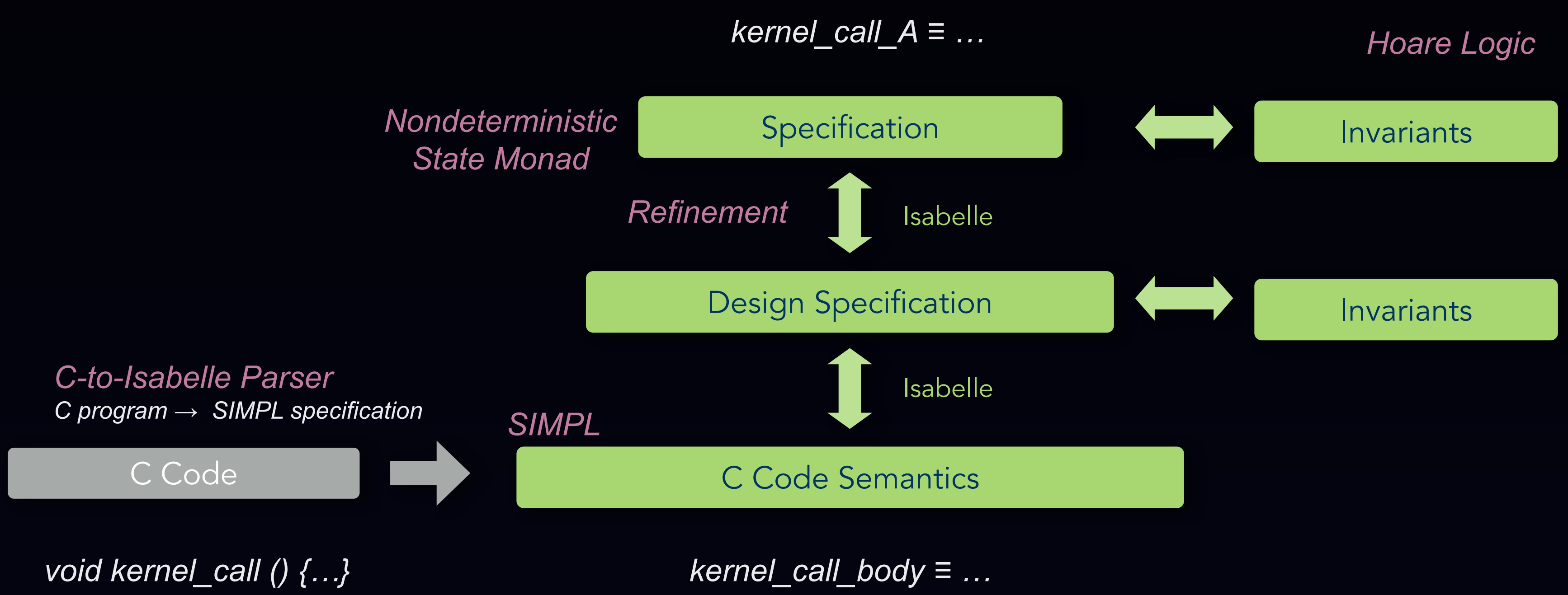
We want to maximise reuse of existing sequential proofs
where concurrency is controlled

We have developed a proof-of-concept framework
for concurrent reasoning for seL4 with maximum reuse

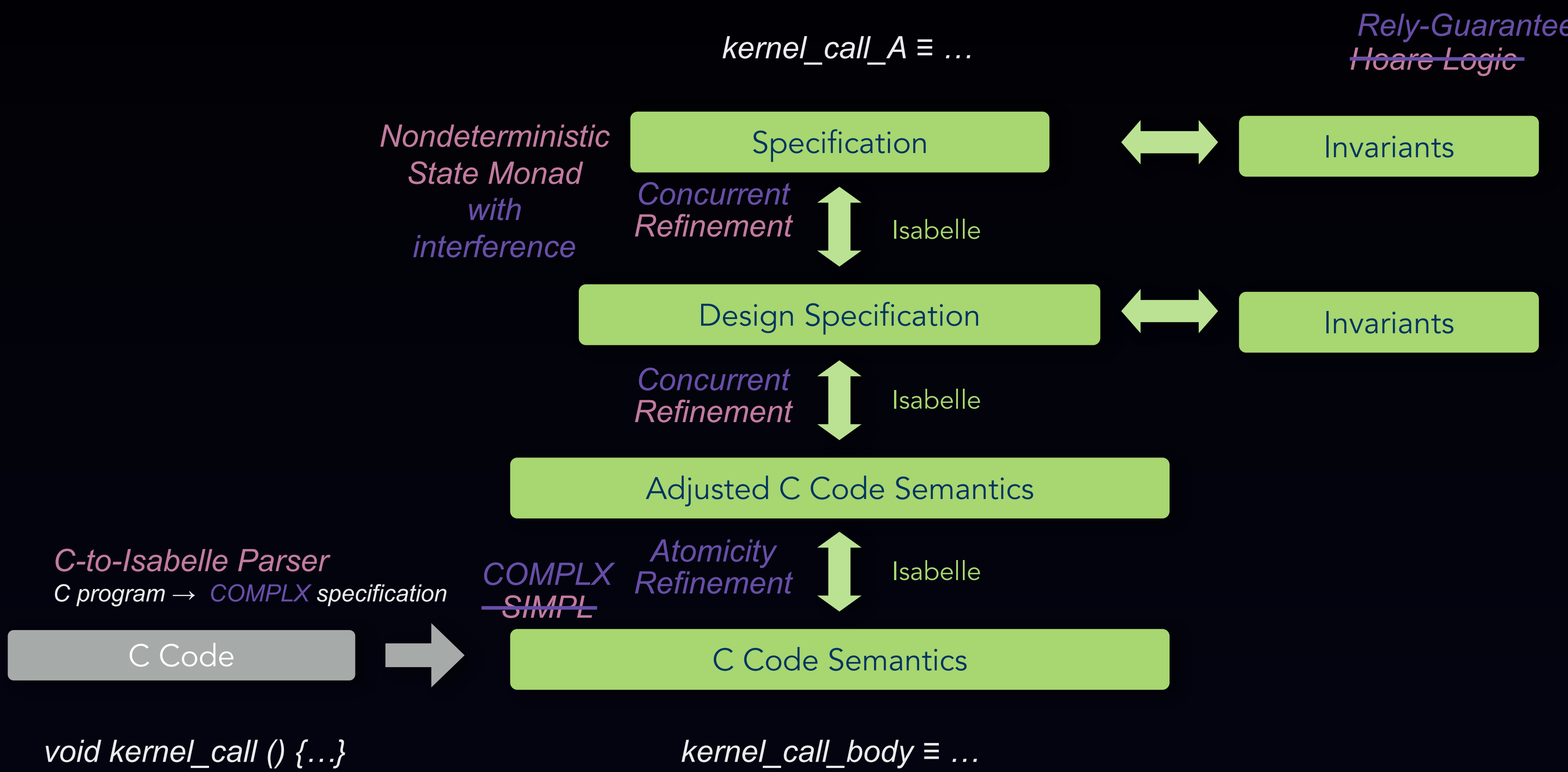
The existing sequential framework (for unicore)



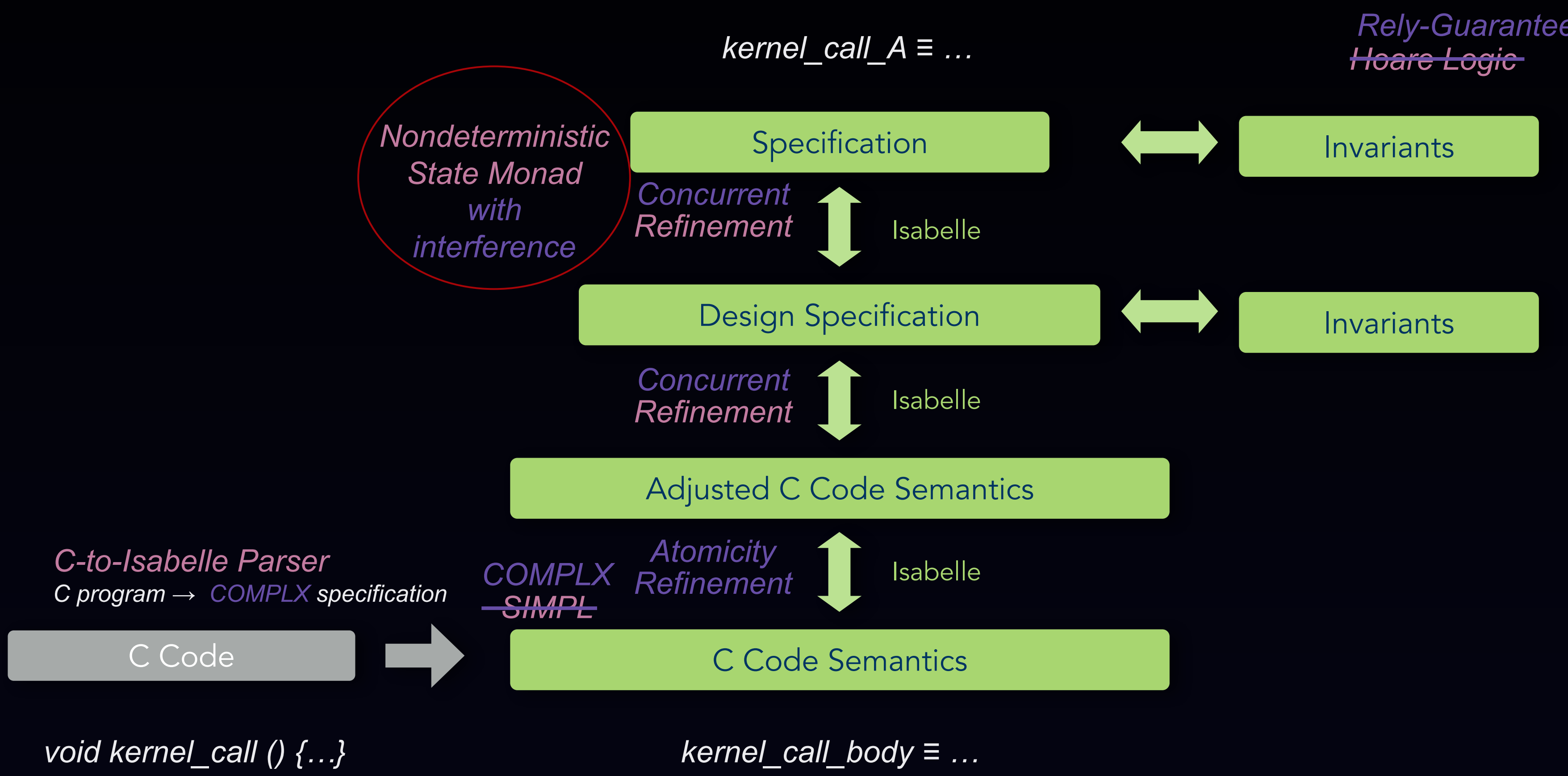
The existing sequential framework (for unicore)



Proof-of-concept concurrent framework



Small dive: interference monad (to maximize reuse)



Small dive:

Sequential: Nondeterministic State Monad

$$\text{state} \longrightarrow (\text{result}, \text{state})\ \text{set}$$

Small dive:

Sequential: Nondeterministic State Monad

$$\text{state} \longrightarrow (\text{result}, \text{state}) \text{ set}$$

```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
        Some f ⇒ return f
      | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
 od"
```

Small dive:

Sequential: Nondeterministic State Monad

$$\text{state} \longrightarrow (\text{result}, \text{state})\ \text{set}$$

```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
          Some f ⇒ return f
        | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
 od"
```

Small dive:

Nondeterministic State Monad
With concurrency?

```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
        Some f ⇒ return f
      | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
 od"
```

Small dive:

Nondeterministic State Monad
With concurrency?

```
"do_fault_transfer badge sender receiver buf ≡ do
  fault ← thread_get tcb_fault sender;
  f ← (case fault of
        Some f ⇒ return f
      | None ⇒ fail);
  (label, msg) ← make_fault_msg f sender;
  sent ← set_mrs receiver buf msg;
  set_message_info receiver $ MI sent 0 0 label;
  as_user receiver $ setRegister badge_register badge
od"
```

Small dive:

Nondeterministic State Monad
With concurrency?

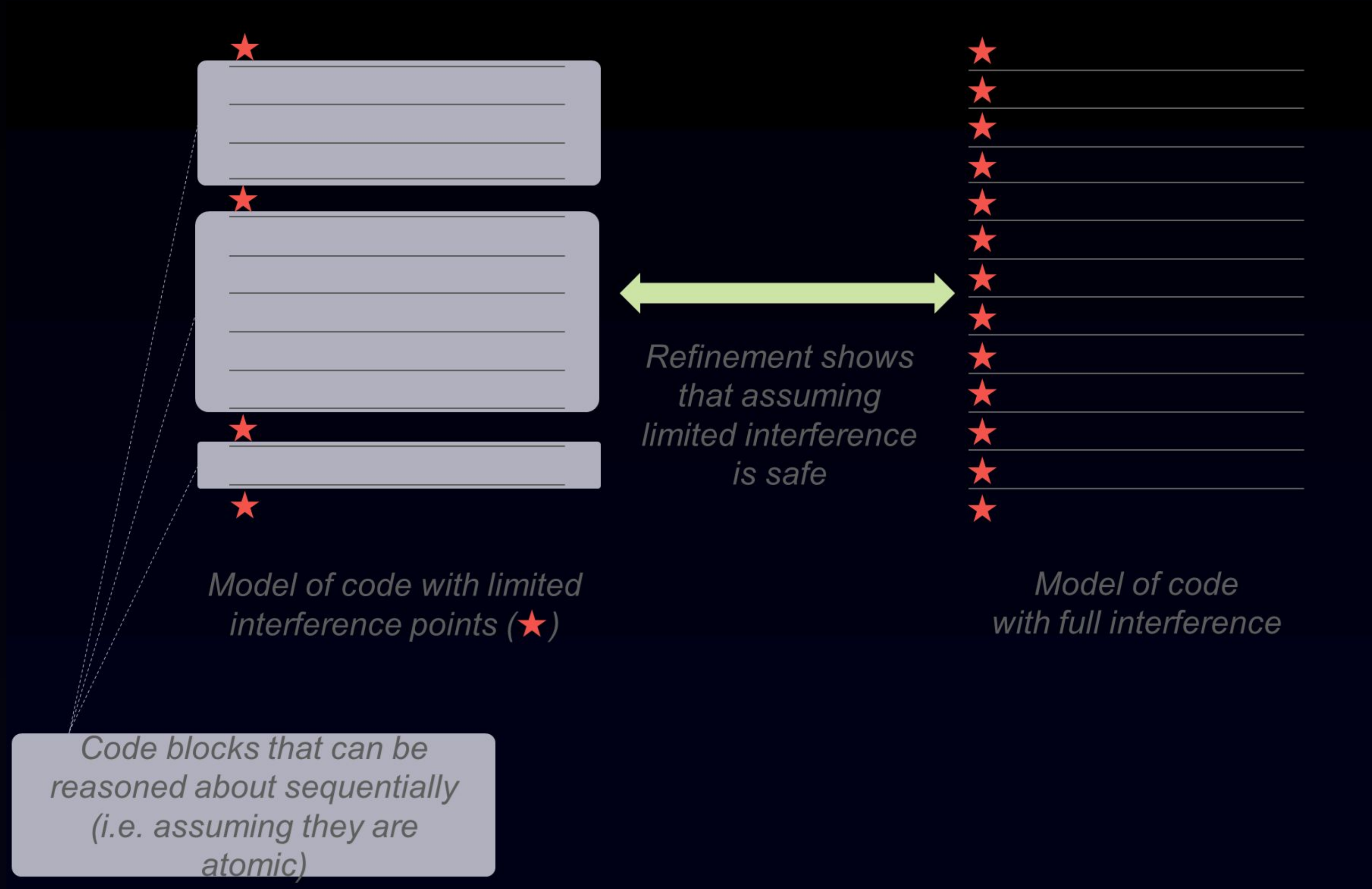
```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
        Some f ⇒ return f
      | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
od"
```

Small dive:

Nondeterministic State Monad
With concurrency?

```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
          Some f ⇒ return f
        | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
od"
```

Limited interference



Small dive:

Concurrent: Interference Trace Monad

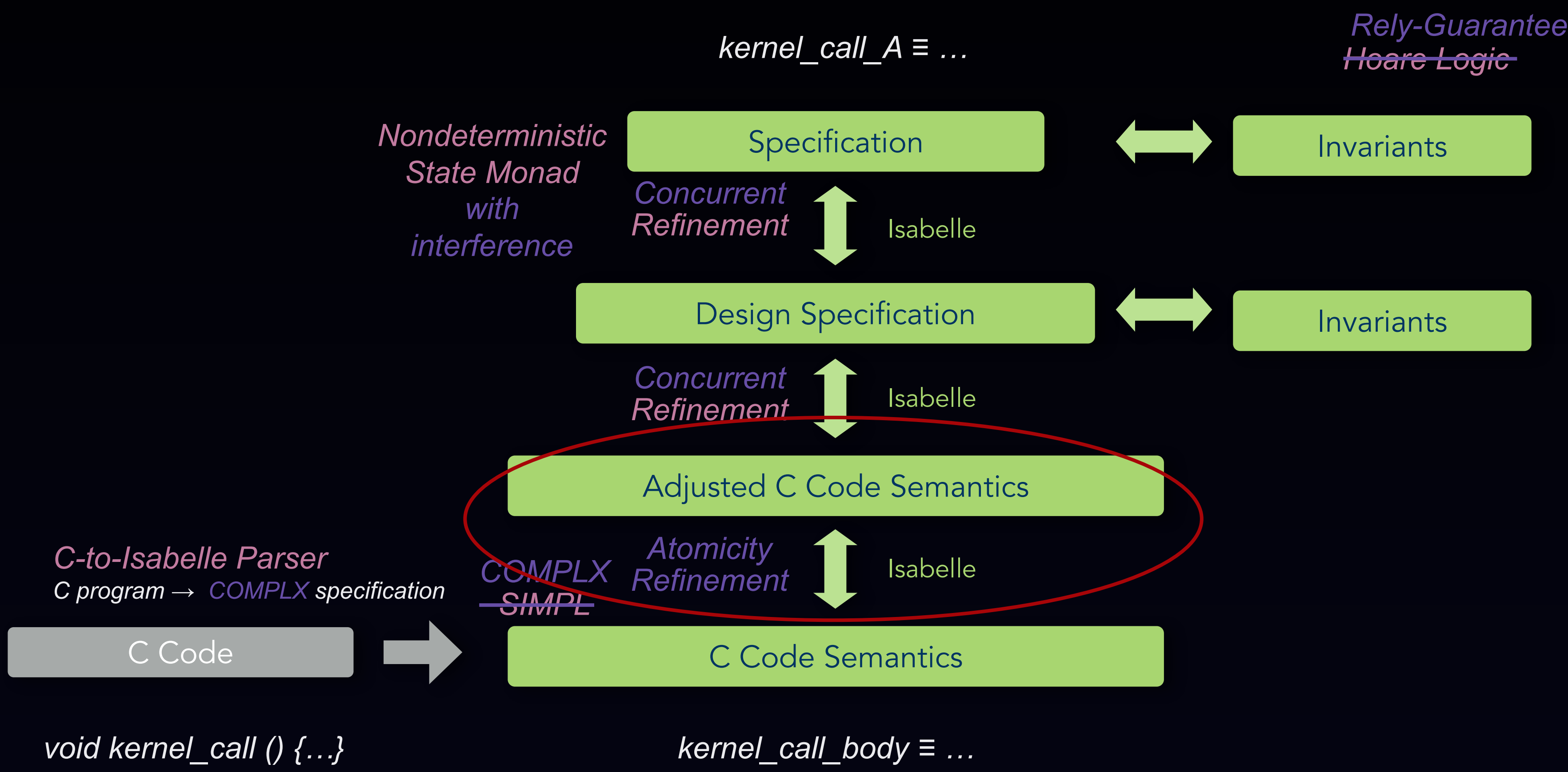
Small dive:

Concurrent: Interference Trace Monad

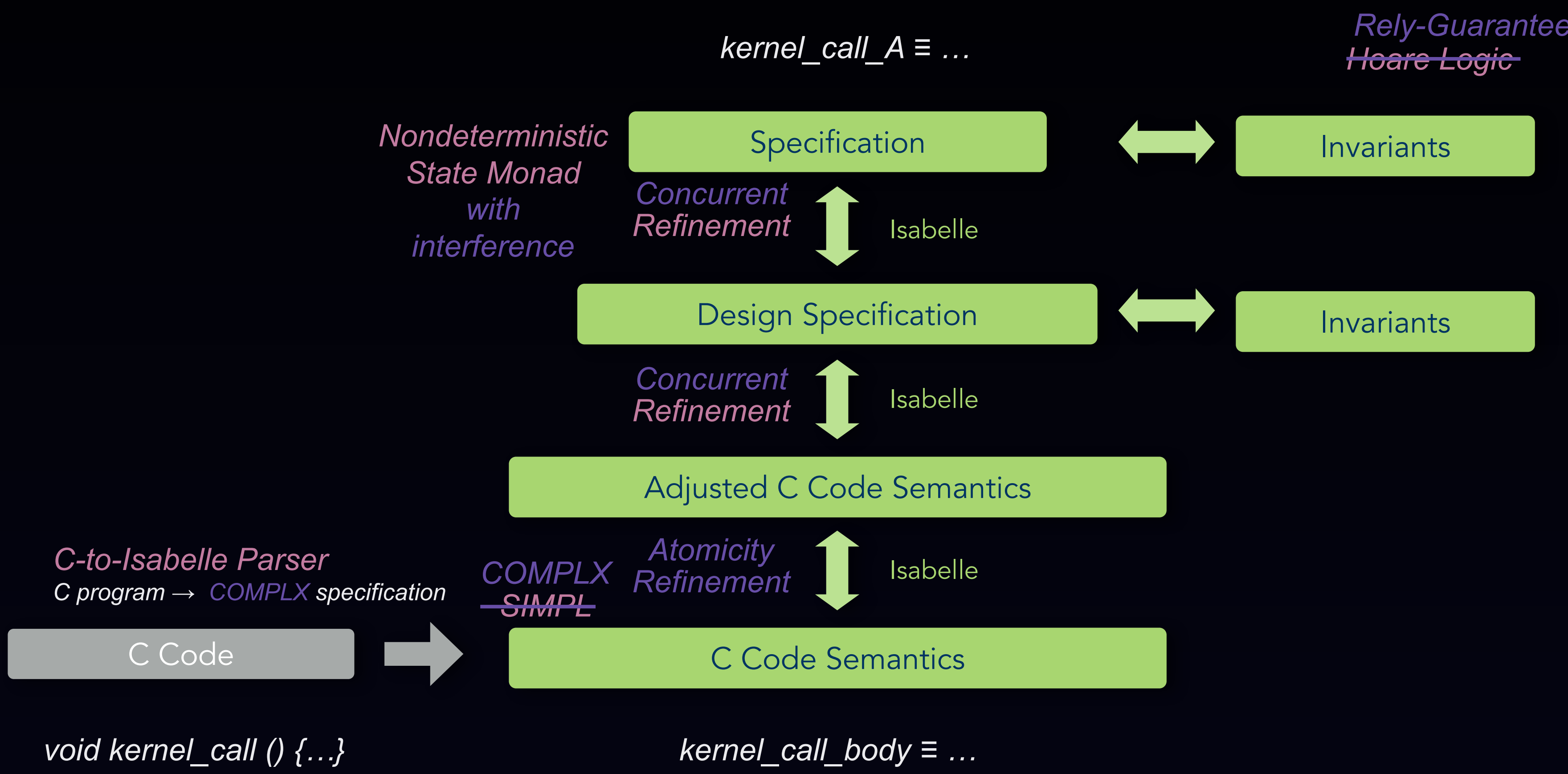
$$state \longrightarrow (trace, (result, state))\ set$$

```
"do_fault_transfer badge sender receiver buf ≡ do
   fault ← thread_get tcb_fault sender;
   f ← (case fault of
         Some f ⇒ return f
       | None ⇒ fail);
   (label, msg) ← make_fault_msg f sender;
   sent ← set_mrs receiver buf msg;
   interference;
   set_message_info receiver $ MI sent 0 0 label;
   as_user receiver $ setRegister badge_register badge
 od"
```

Proof-of-concept concurrent framework



Proof-of-concept concurrent framework



Proof-of-concept concurrent framework

Rely-Guarantee
~~oare Logic~~

riants

Now how do we apply this to update all of the seL4 proofs?

riants

Adjusted C Code Semantics

C-to-Isabelle Parser
C program → COMPLX specification

COMPLX
~~*SIMPL*~~

Atomicity Refinement

Isabelle

C Code

C Code Semantics

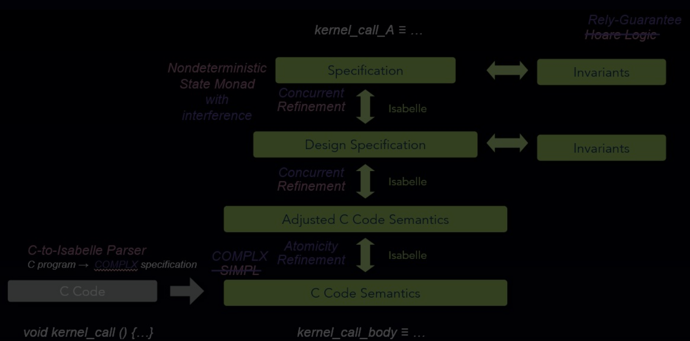
void kernel_call () {…}

kernel_call_body ≡ …

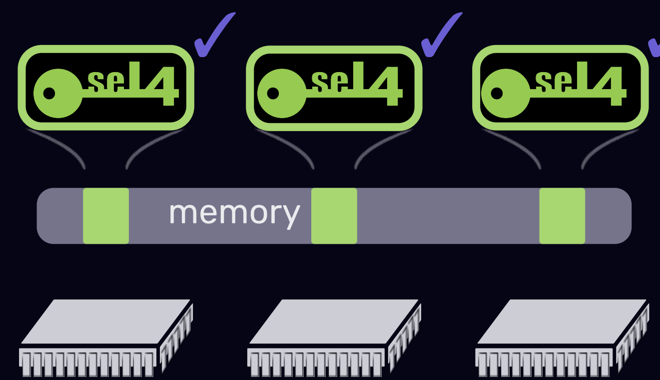
Overview

<u>Goal</u>:
Allow use of multiple cores as soon as possible,
With incrementally stronger and stronger assurance

What's hard?
What have we got so far?



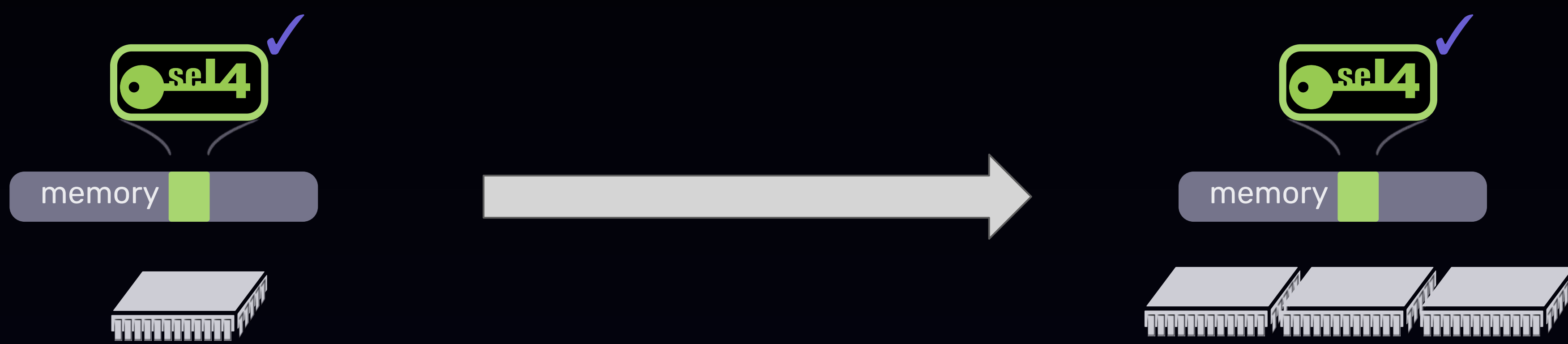
Towards a verified static multikernel seL4



Progressive roadmap

Single core

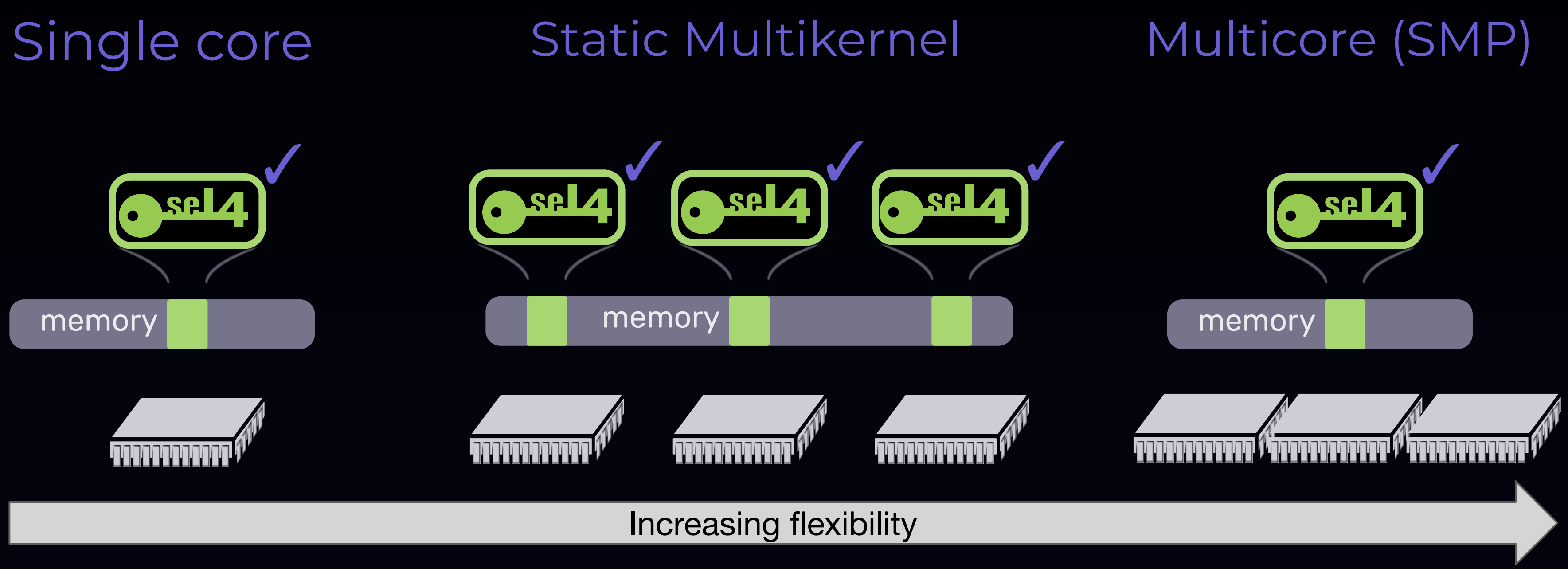
Multicore (SMP)



Need full concurrency on Day 1

No assurance until done

Progressive roadmap: via static multikernel



One seL4 per core

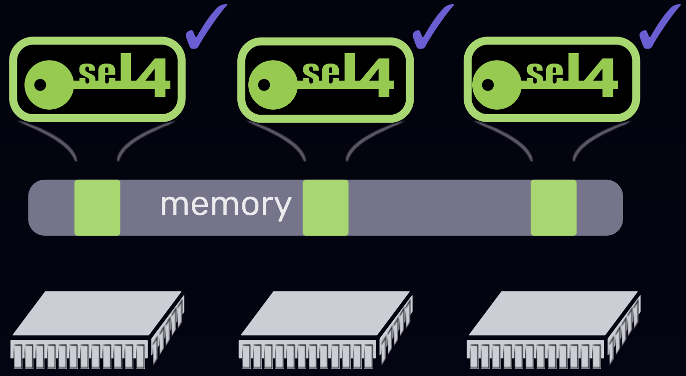
Progressively building stronger assurance from Day 1

Static multikernel configuration of seL4

- Each core runs a copy of the kernel
  - Each copy has separate resources and data structures
  - No kernel-kernel interactions

- User code communicates via shared memory and inter-processor interrupts
  - seL4 API remains nearly identical

- Static partition of memory simplifies verification
  - Still provides increased utility and performance



Multikernel seL4 verification roadmap

Assurance

Multikernel seL4 verification roadmap

Verify sequentially

- Verify code changes sequentially
  - Add IPI API

Sequentially correct

Assurance

Multikernel seL4 verification roadmap

✔ Verify sequentially

- Verify code changes sequentially
  - Add IPI API

Sequentially correct

Assurance

Multikernel seL4 verification roadmap

✔ Verify sequentially

List obligations

- Identify required proof obligations
  - e.g. separation of resources between kernel instances

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

. . .

Assurance

Multikernel seL4 verification roadmap

✔ Verify sequentially

List obligations — Prove obligations

- Prove required obligations in isolation
  - Proofs would still be sequential

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores
…

Assurance

Multikernel seL4 verification roadmap

✔ Verify sequentially

List obligations — Prove obligations

Parametrise specifications

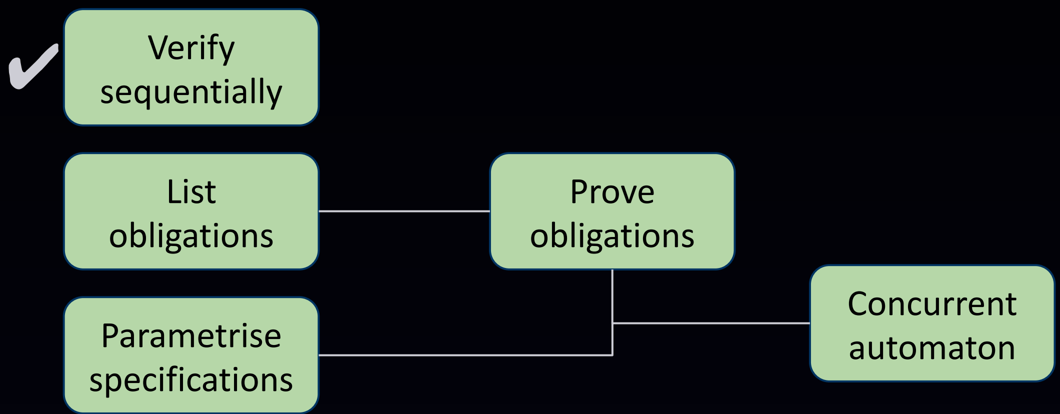
- Parametrise specifications to allow multiple instances of the kernel
  - Parameters such as physical memory location

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

…

Assurance

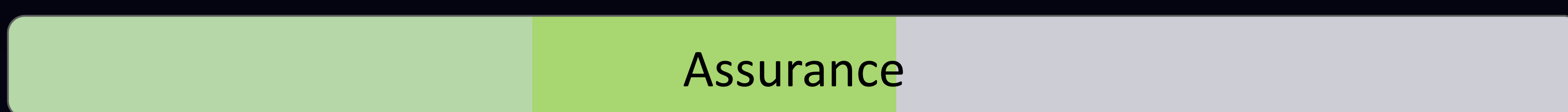
Multikernel seL4 verification roadmap



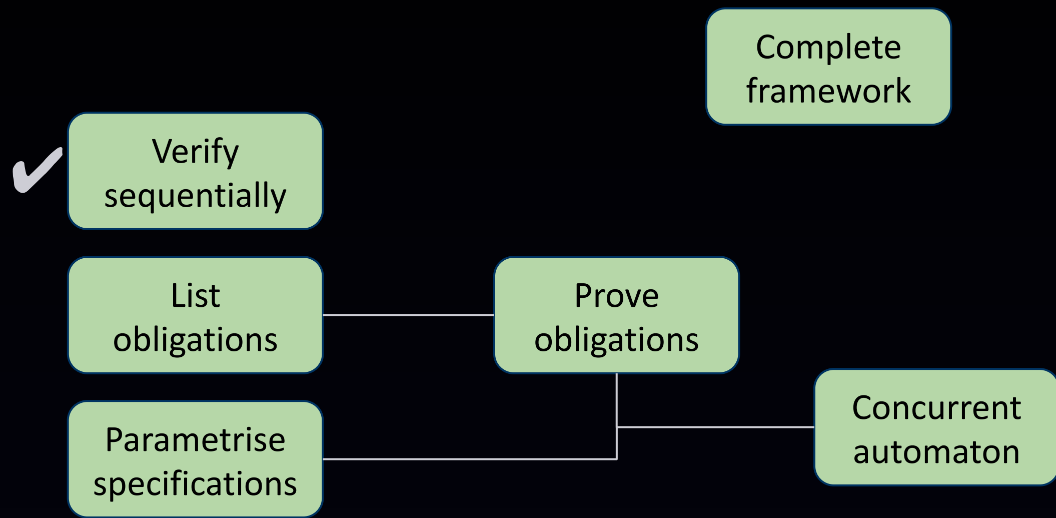
- Add coarse-grained concurrency to the automaton
  - Transitions are still atomic, some obligations will be validated

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

…
More proof obligations?



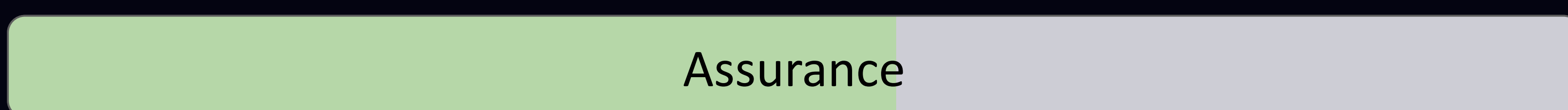
Multikernel seL4 verification roadmap



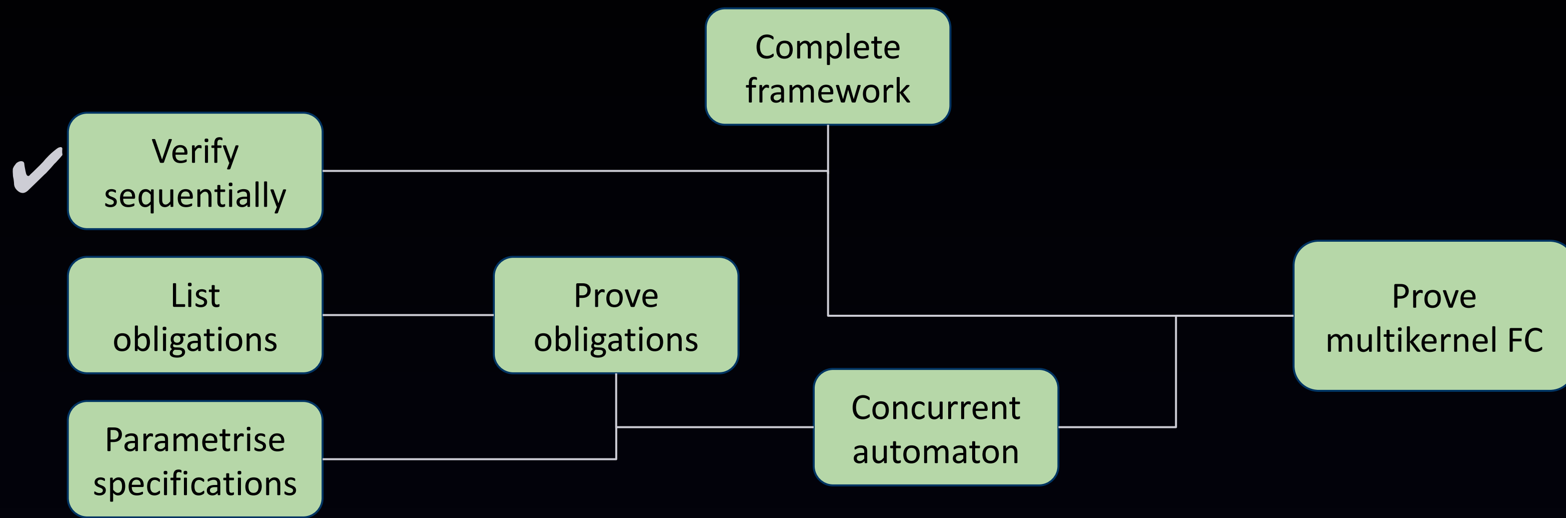
- Exercise and complete concurrency framework
  - Monad rulesets, haskell translator, atomicity refinement, C-Parser, …

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

…
More proof obligations?



Multikernel seL4 verification roadmap



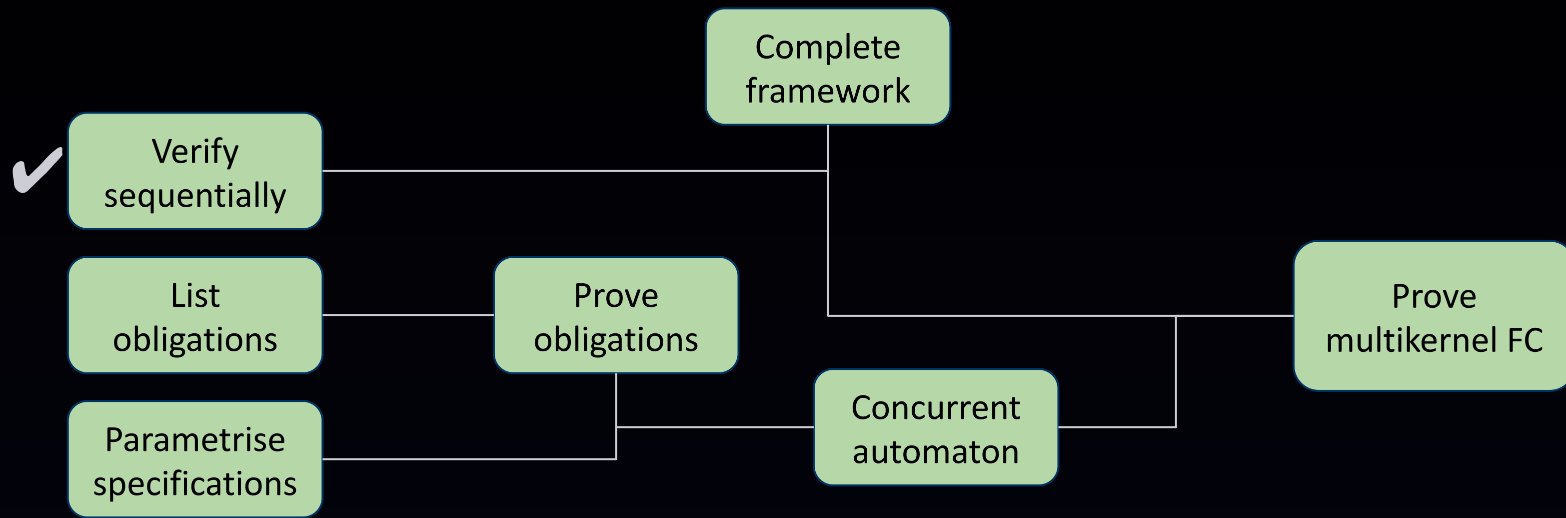
- Prove functional correctness for multikernel
  - This is where full concurrency is introduced

Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

…
More proof obligations?

Assurance

Multikernel seL4 verification roadmap



- Prove functional correctness for multikernel
  - This is where full concurrency is introduced

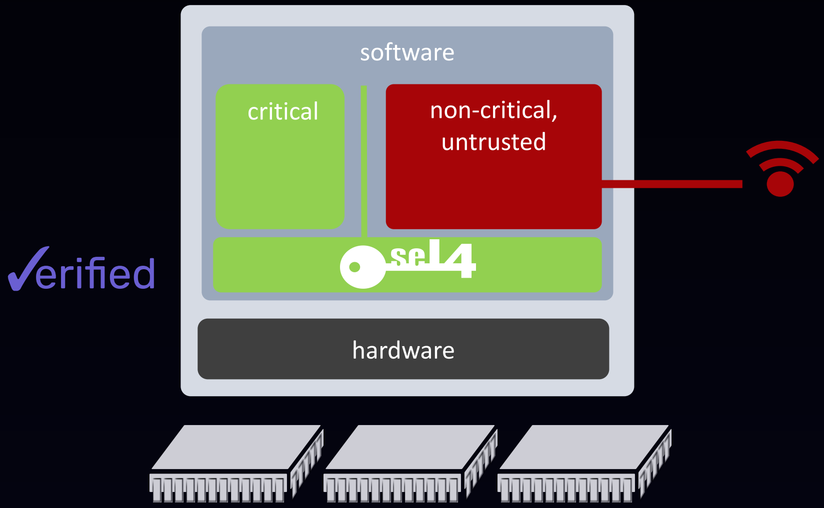
Sequentially correct
Separation of resources maintained
Isolation of kernels on different cores

…
More proof obligations?
Functional correctness!

Assurance

What do we want?

Goal:
Allow use of multiple cores as soon as possible,
with incrementally stronger and stronger assurance



Assurance

Thank you

Proofcraft
Corey Lewis
Principal Proof Engineer