BedRock Systems Inc

Programming a Microkernel Specification in Separation Logic

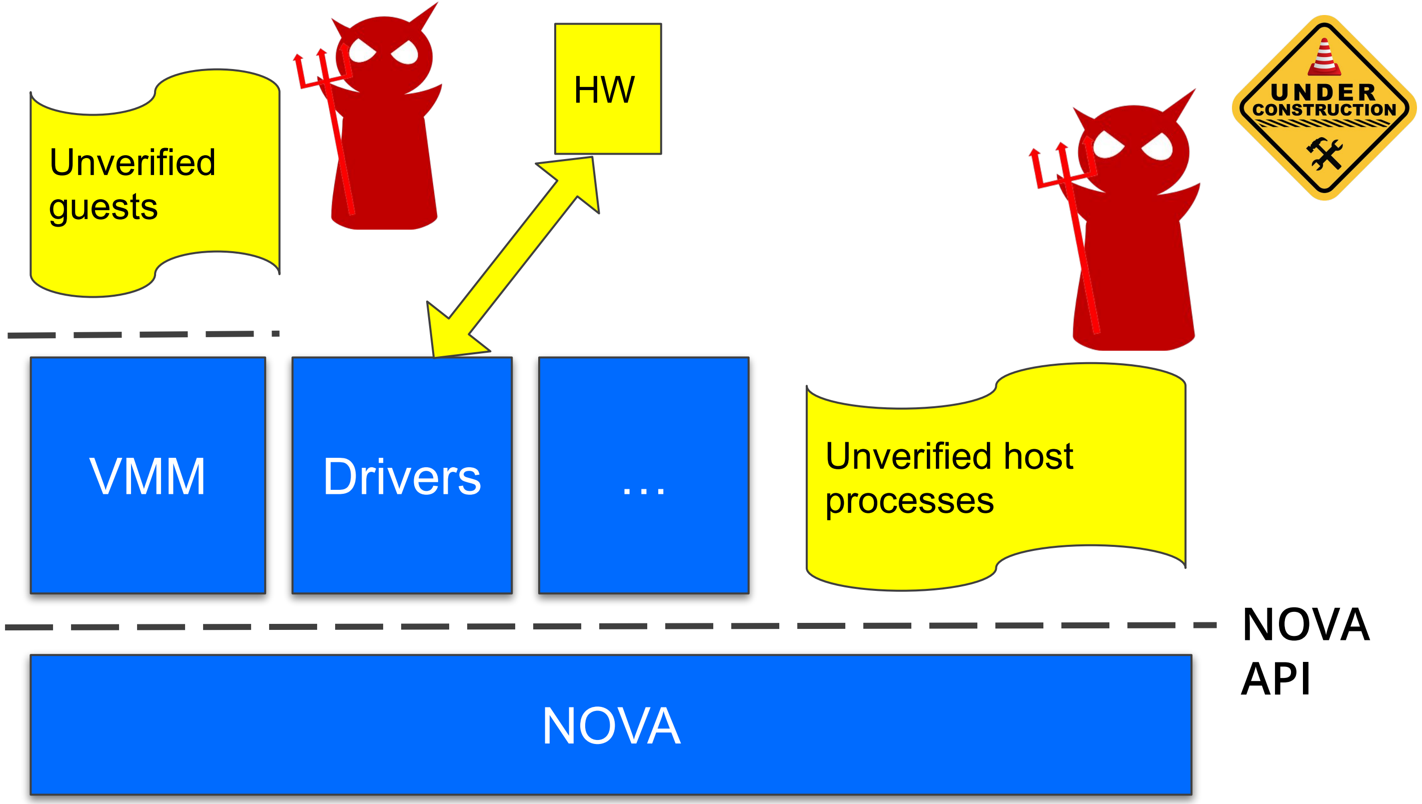
Paolo G. Giarrusso, Gregory Malecha, David Swasey, Yoichi Hirai

BedRock Systems, Inc.

Formal Verification @ Bedrock

Work-in-progress proof of **bare-metal property**: VMM refines bare-metal machine.

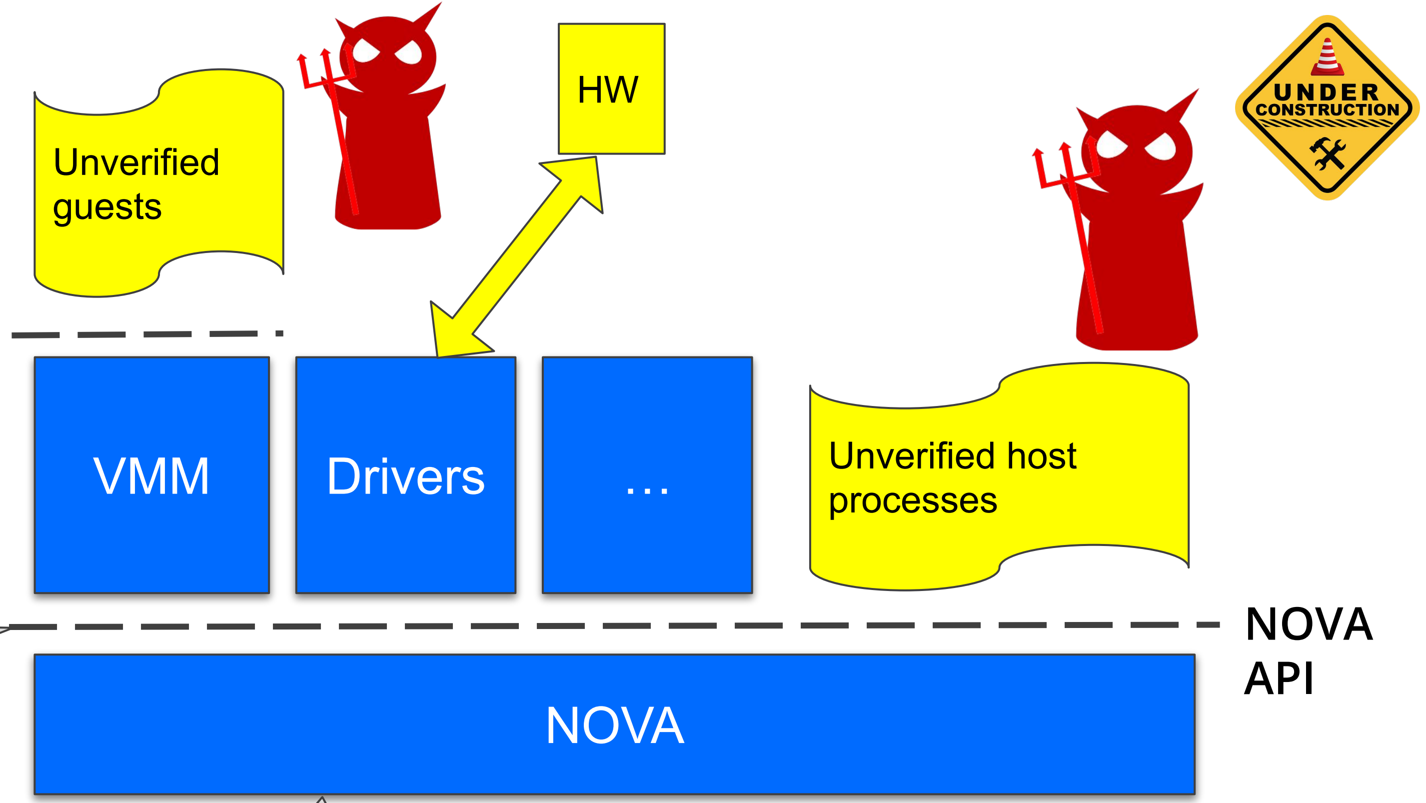
- Operational semantics "at the boundaries" — HW & unverified guests.



TCB:

- C++ compiler correctness
- C++ axiomatic semantics in Iris
- HW models

Formal Verification @ Bedrock

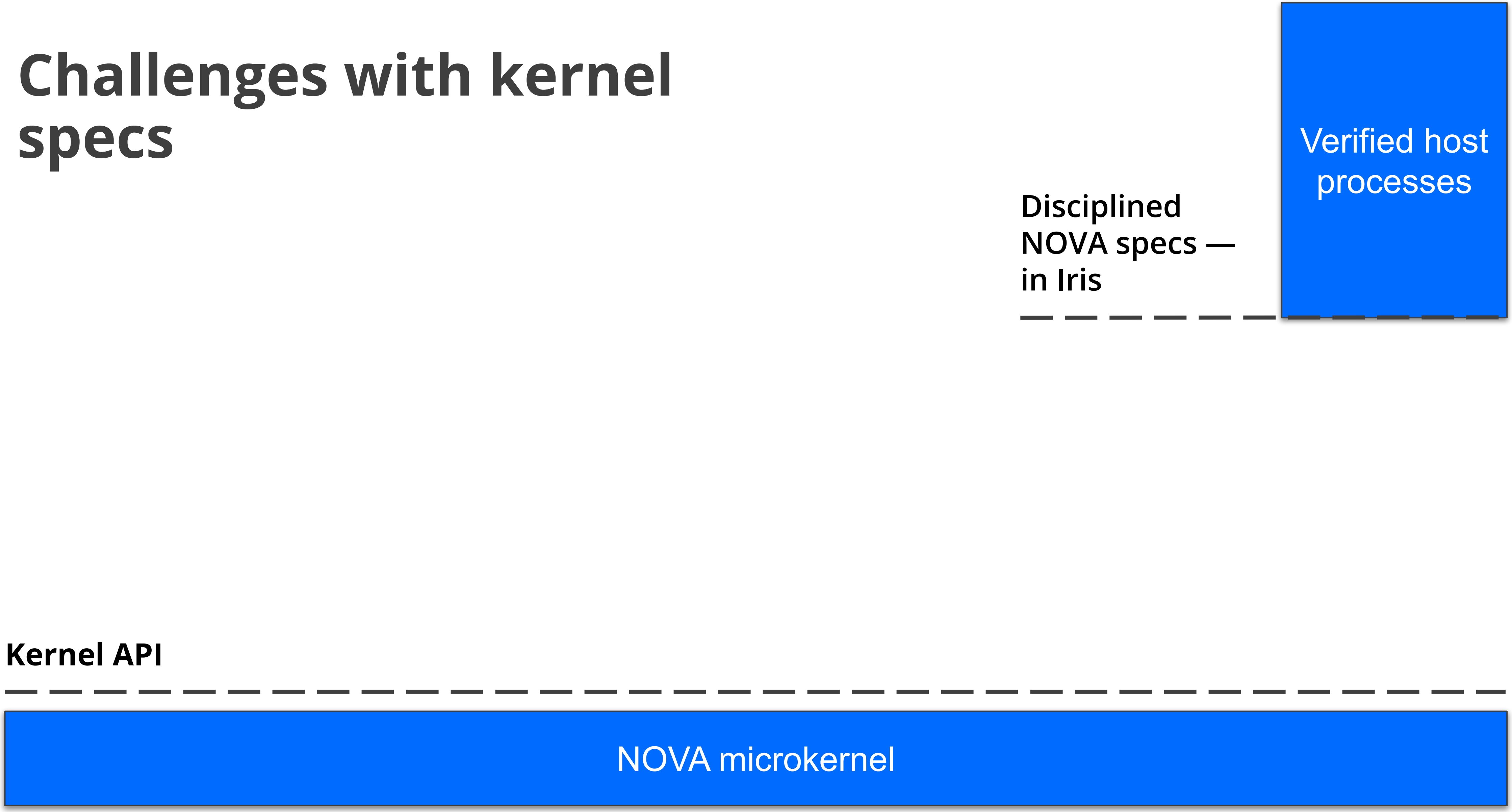


This talk

NOVA API (**outside of bare-metal property statement)**

Highly concurrent L4-family capability-based microkernel

Challenges with kernel specs



Challenges with kernel specs

Verified host processes

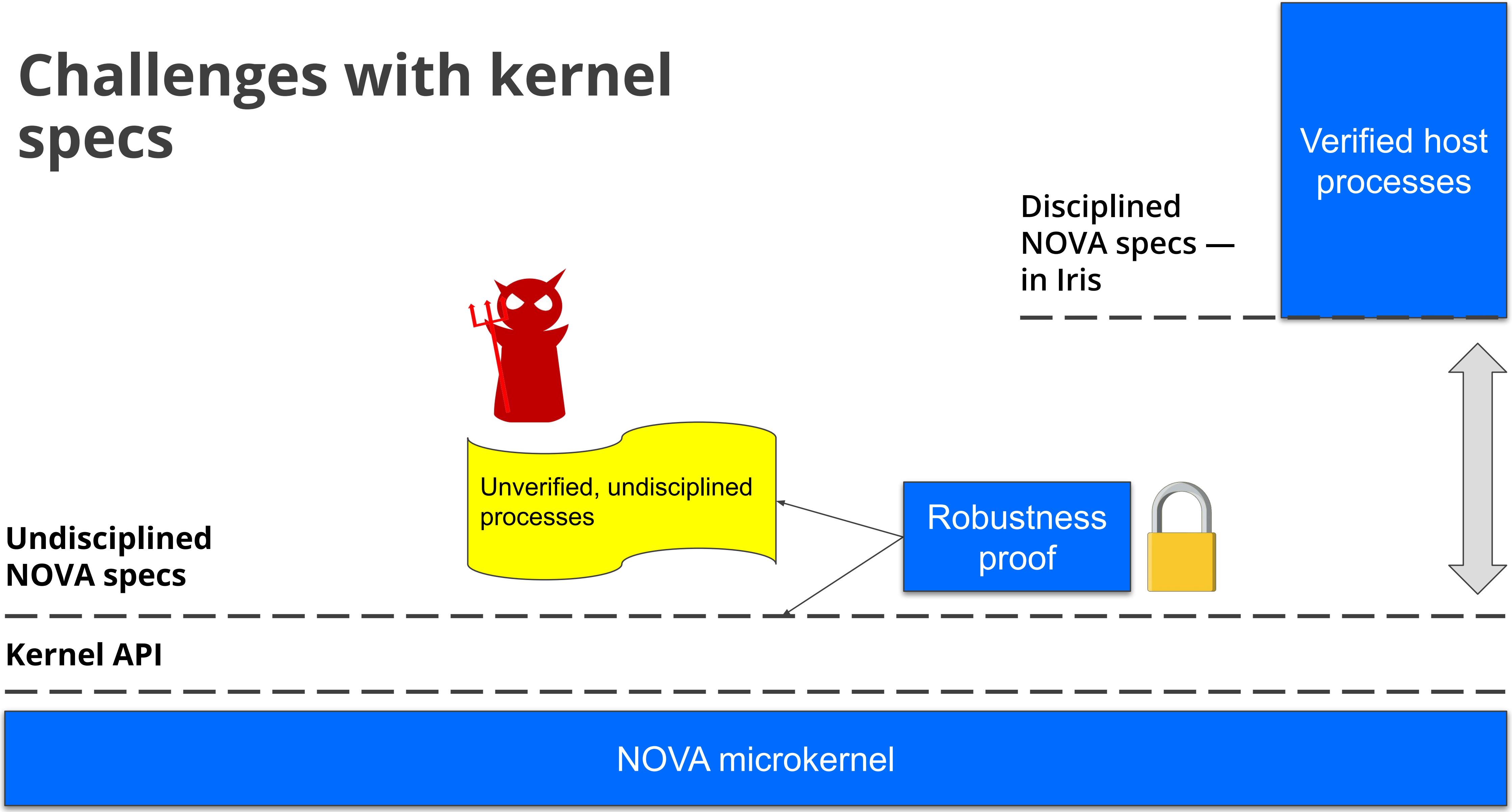
Disciplined NOVA specs — in Iris

Unverified, undisciplined processes

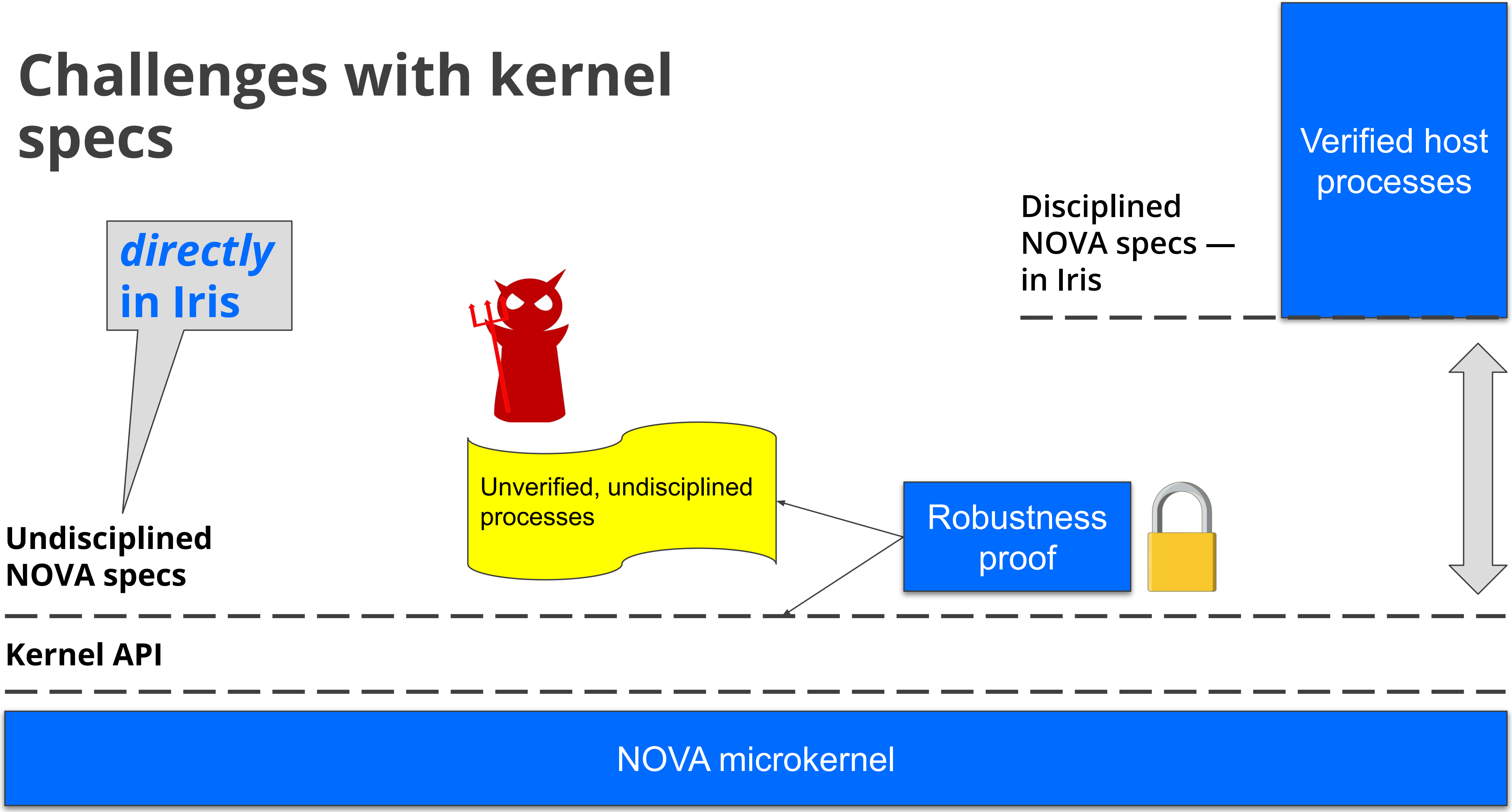
Kernel API

NOVA microkernel

Challenges with kernel specs



Challenges with kernel specs



Undisciplined specs in Iris: Advantages

- Single proof for NOVA (NOVA's pretty complex)
- Small footprint without detours through big footprint and associated overhead
- We lose adequacy for NOVA in isolation; but appropriate for us since NOVA's internal

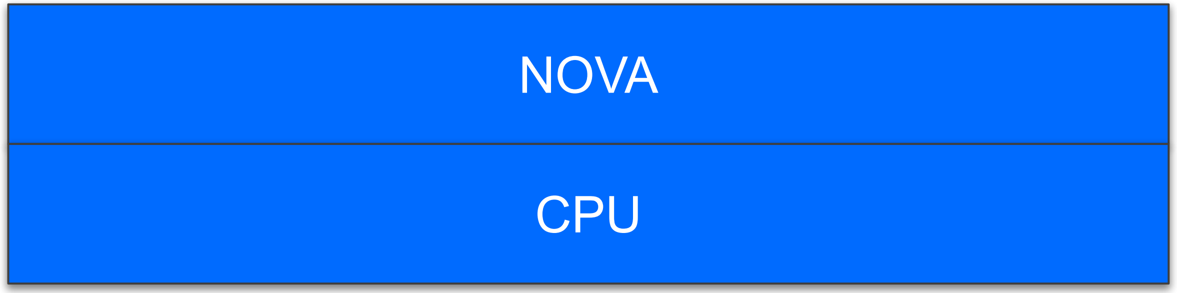
Subjectively:

- Easy to evolve
- Two specs, but little duplication (undisciplined specs are mostly about error handling and atomicity)

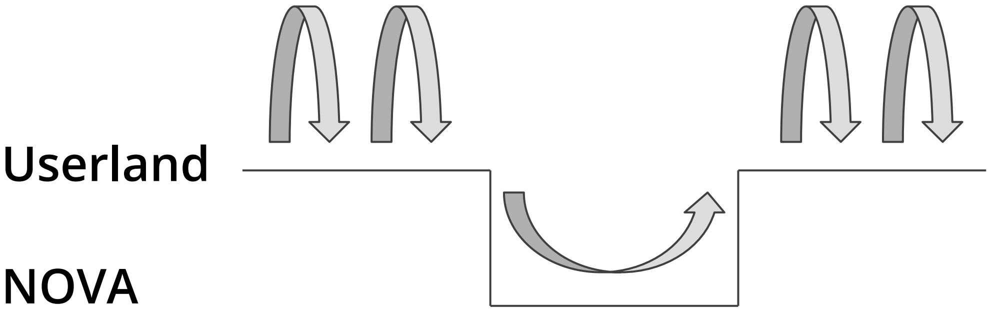
Undisciplined NOVA specs as axiomatic semantics

An undisciplined WP for the NOVA machine

NOVA machine = NOVA + CPU:



Execution alternates normal steps and NOVA steps:



Predicates:

`nova.wp : ∀ (ec : ec_nameT), mpred`

`ec.regs : ec_nameT -> Qp -> regsT -> mpred`

Types:

`Val := False. Expr := Unit.`

`ec_nameT` : an identifier for a "thread" (Execution Context)

`regsT` : the type of the "register file" (CPU internal state)

```
regular_machine_step :
   ∀ (old new : regsT), Prop
```

atomic CPU steps (no assumptions on guest discipline)

HW, caches, memory modeled as external components

An undisciplined WP for the NOVA machine

```
nova.wp_step_intro:
|={⊤,↑nova_ns}=> ▷ (∃ regs, ec.regs ec 1 regs ∗
    if syscall_trap regs then wp_hypercall ec regs else
      (∀ regs', [| regular_machine_step regs regs' |] -∗
        ec.regs ec 1 regs' ={↑nova_ns,⊤}=∗ wp ec)
      ∧ wp_traps ec regs)
⊢ nova.wp ec.
```

Elimination rule: syscall for spawning threads

An undisciplined WP for the NOVA machine

```
wp_hypercall ec regs :=
  match decode_syscall regs with
  | ipc_call => wp_ipc_call ec regs
  | ipc_reply => wp_ipc_reply ec regs
  | …
  end.
```

Robustness

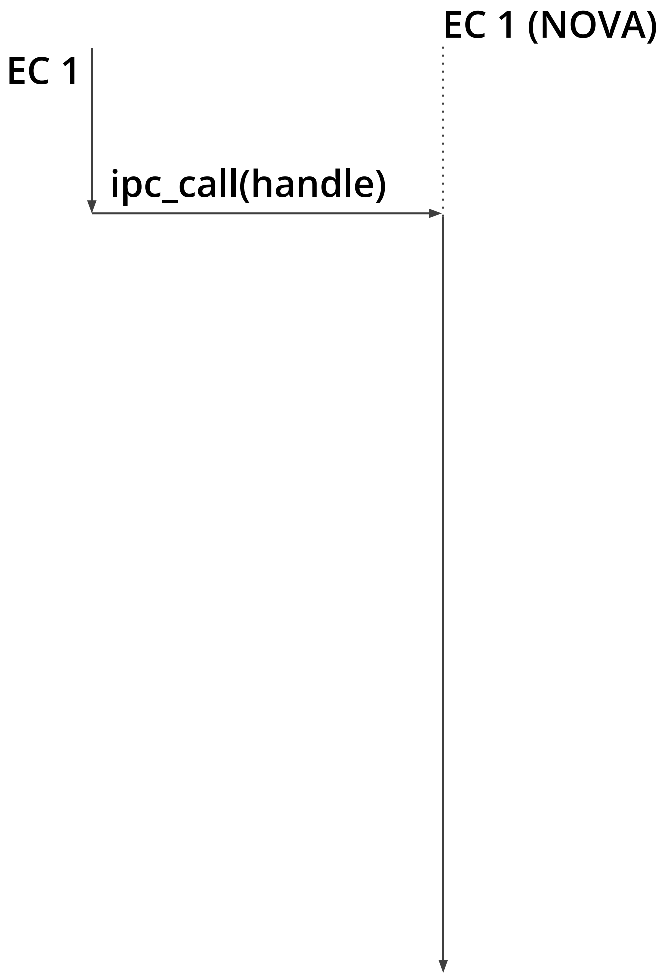
Robustness statement:

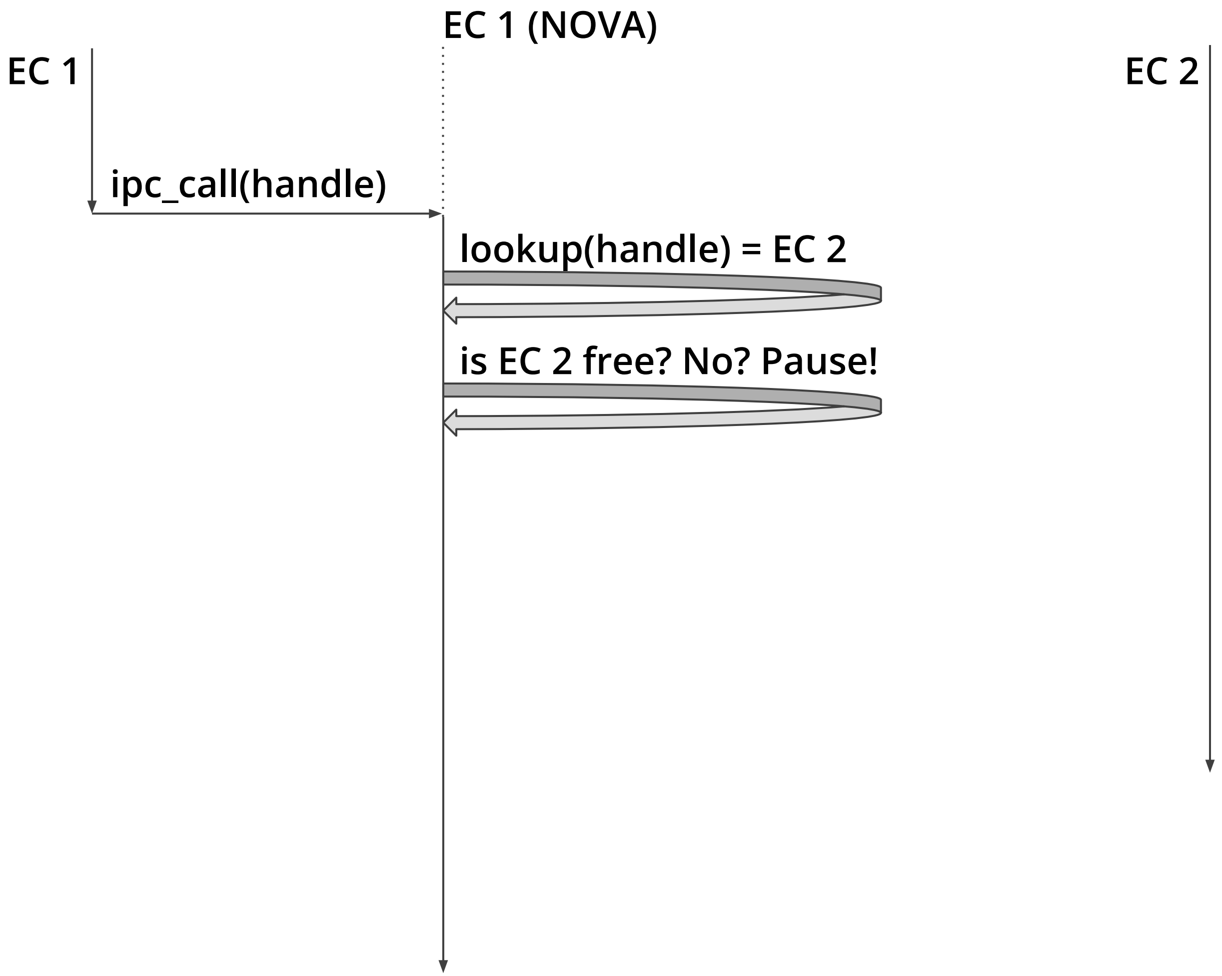
```
inv invName process_resources * persistent_process_props ⊢
nova.wp ec
```

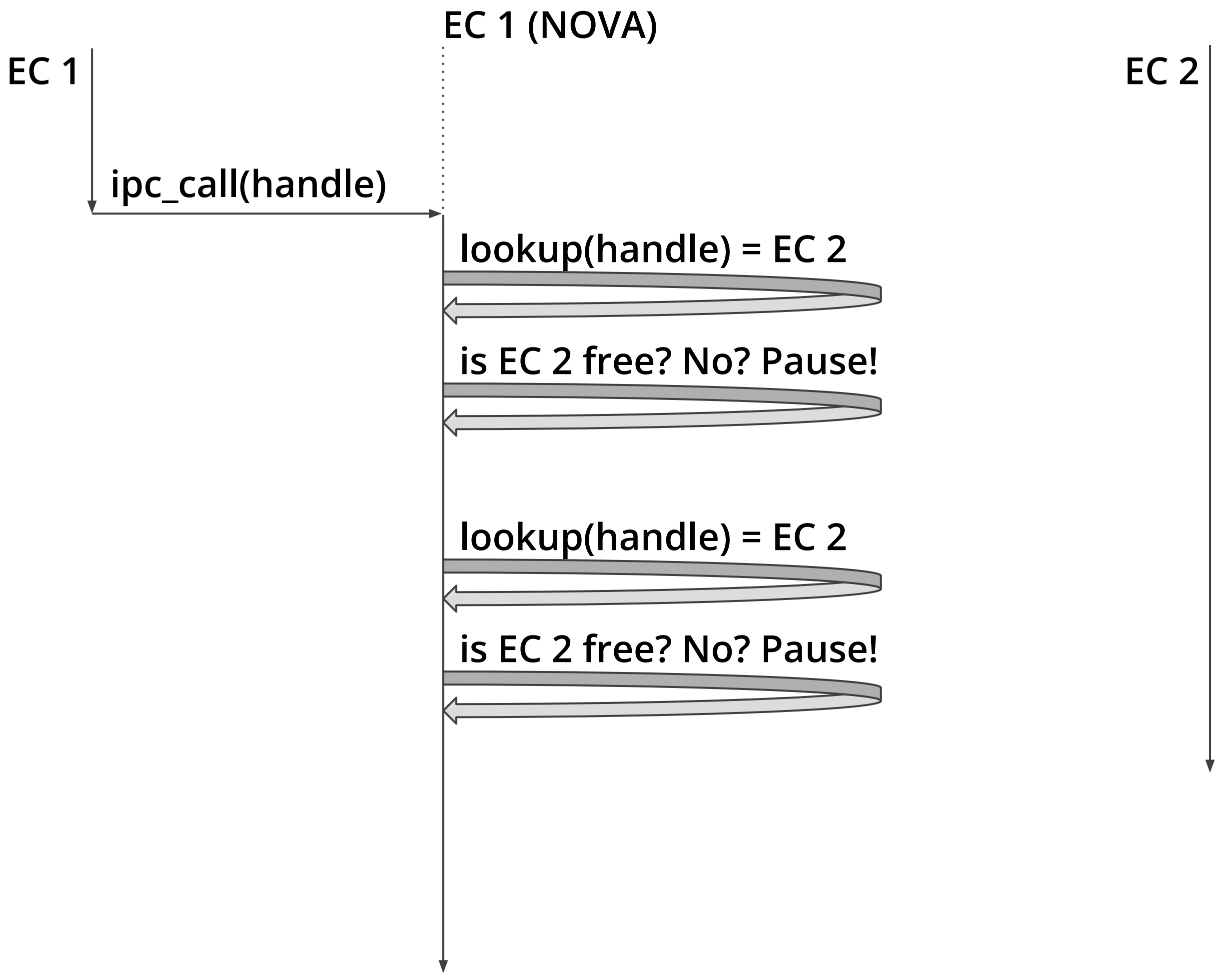
Proof sketch: by Löb induction and case analysis on the step; each obligation must be satisfied via the invariant.

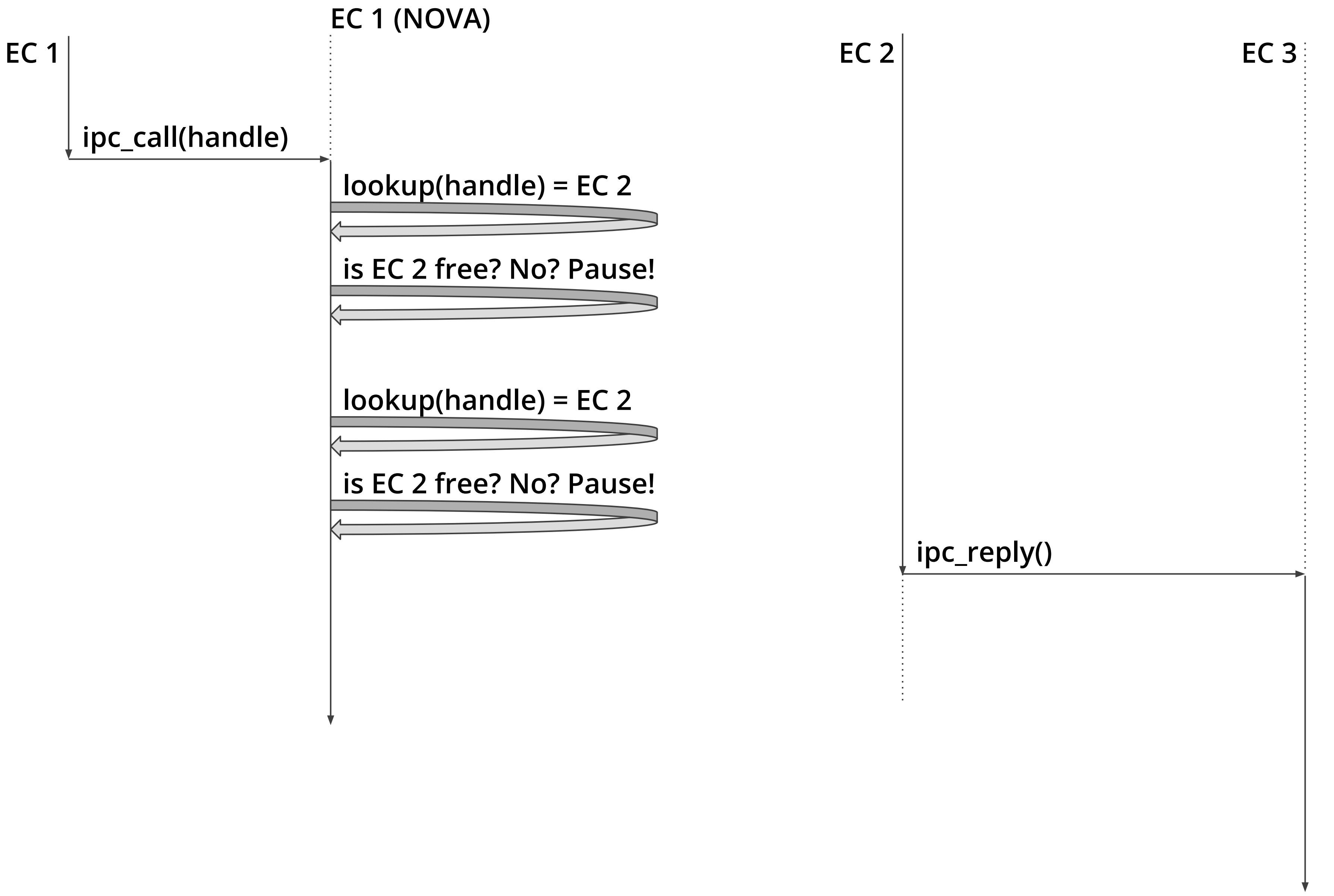
- For memory, for each physically accessible page (via page tables) we need ownership in invariants.
- For syscalls, we must satisfy all syscall preconditions from invariants.

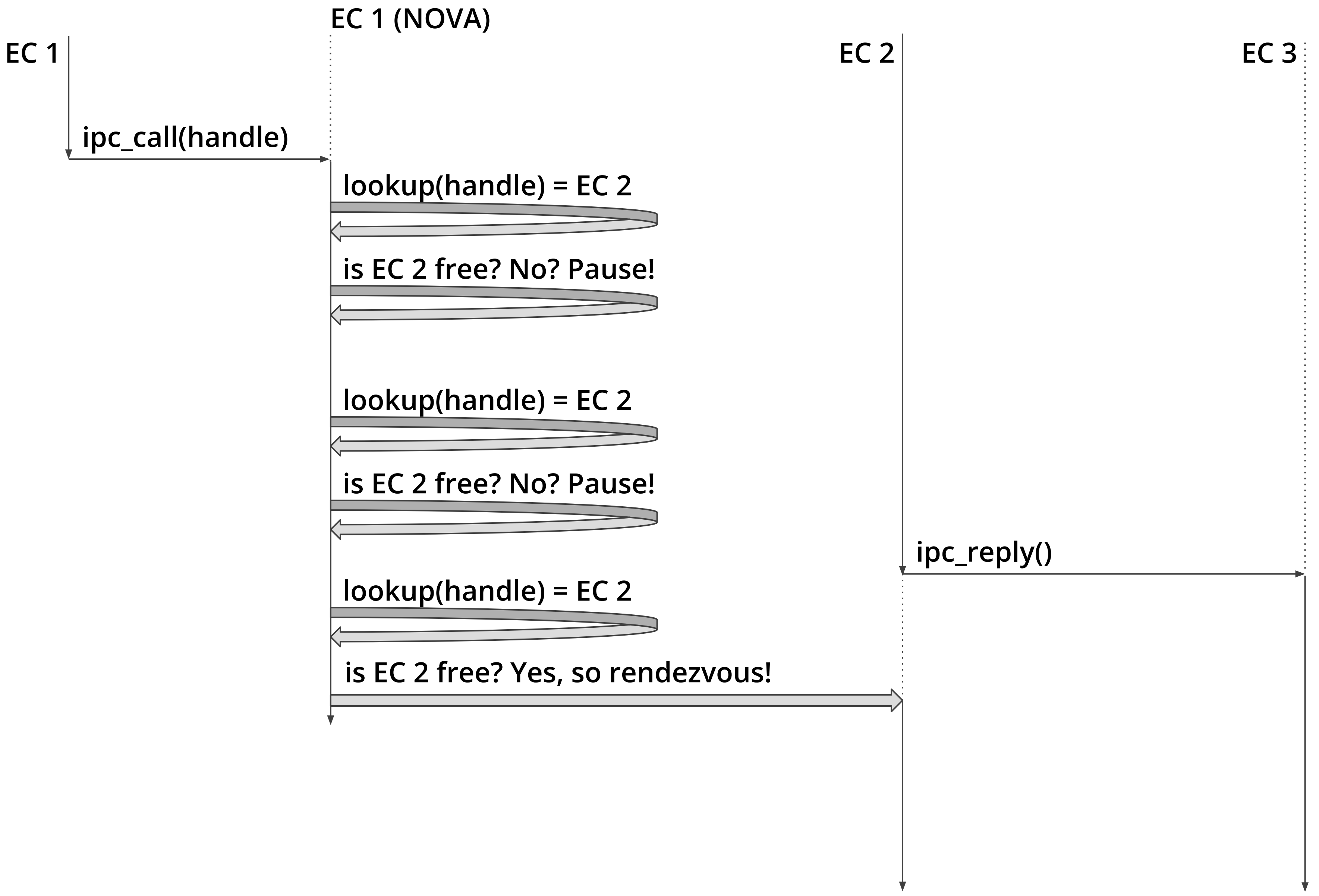
An example syscall: IPC call


EC 1 (NOVA)
EC 1
ipc_call(handle)


EC 1 (NOVA)
EC 1
EC 2
ipc_call(handle)
lookup(handle) = EC 2
is EC 2 free? No? Pause!


EC 1 (NOVA)
EC 1
EC 2
ipc_call(handle)
lookup(handle) = EC 2
is EC 2 free? No? Pause!
lookup(handle) = EC 2
is EC 2 free? No? Pause!


EC 1 (NOVA)
EC 1
EC 2
EC 3
ipc_call(handle)
lookup(handle) = EC 2
is EC 2 free? No? Pause!
lookup(handle) = EC 2
is EC 2 free? No? Pause!
ipc_reply()


EC 1 (NOVA)
EC 1
EC 2
EC 3
ipc_call(handle)
lookup(handle) = EC 2
is EC 2 free? No? Pause!
lookup(handle) = EC 2
is EC 2 free? No? Pause!
ipc_reply()
lookup(handle) = EC 2
is EC 2 free? Yes, so rendezvous!

Rendezvous in Iris

```
Definition resolve_handle_chan_rendezvous
    (caller_ec : ec_nameT) handle Q :=

  AU << ∀ chan rights q callee_state,
        cap_at caller_ec handle q (channel, rights) ∗

        □ channel_ec channel callee_ec ∗

        ec.kstate callee_ec callee_state >> @ novaM , ∅

        << ∃ result, cap_at handle q (chan, rights) ∗
           if insufficient rights then [| result = EPERM |] ∗

             ec.kstate callee_ec callee_state

           else [| callee_state = AVAILABLE ∧ result = SUCCESS |] ∗
             ec.kstate callee_ec RUNNING),
        COMM Q result callee_ec >>.
```

ipc_call combined "CPS" spec (simplified)

```
Definition ipc_spec_raw caller_ec handle :=
  resolve_handle_chan_rendezvous caller_ec handle
    (λ result callee_ec,
      ∀ src dst,
        buf_addr caller_ec src -∗ (* Persistent *)
        buf_addr callee_ec dst -∗
        do_buf_copy caller_ec callee_ec
          (do_set_regs callee_ec
            (nova.wp callee_ec)))
```

ipc_call buffer copies

Example: inter-process message send, simplified

```
{ nova_src_buf |-> msg_bytes0 * P msg_bytes0 * channel_spec channel_handle P Q }
ipc_call(channel_handle)
{ nova_src_buf |-> msg_bytes1 * Q msg_bytes1 * channel_spec channel_handle P Q }
```

Example: inter-process message send, simplified

```
{ nova_src_buf |-> msg_bytes0 * P msg_bytes0 * channel_spec channel_handle P Q }
ipc_call(channel_handle)
{ nova_src_buf |-> msg_bytes1 * Q msg_bytes1 * channel_spec channel_handle P Q }
```

- Sufficient for undisciplined clients: no ❌, assumes ***sequential*** ownership (not satisfiable from invariants)!
- Other threads can write to the buffer during the call

Buffer copy with atomic triples

```
{ nova_src_buf |-> msg_bytes ∗ (∃ xs, nova_dst_buf |-> xs) }
ipc_call_copy()
{ nova_src_buf |-> msg_bytes ∗ nova_dst_buf |-> msg_bytes }
```

```
<<< ∀ msg_bytes, nova_src_buf |-> msg_bytes ∗ (∃ xs, nova_dst_buf |-> xs) >>>
ipc_call_copy()
<<< nova_src_buf |-> msg_bytes ∗ nova_dst_buf |-> msg_bytes >>>
```

- Sufficient for unverified clients: ✅ — Sequential ownership not required!
- Implies disciplined spec: ✅ (atomic triples imply sequential triples)
- Implementable (efficiently): ❌
  - ❌ normal buffer read is not atomic
  - ❌ a big kernel lock would not suffice; only stopping all other threads
  - ❌ performance requires unsynchronized reads
  - 💡 **multiple** atomic steps!

Byte copy via sequential composition

```
<<< ∀ x, P >>> e <<< ∃ y, Q RET f x y >>> :=
  ∀ R, AU << ∀ x, P x >> << ∃ y, Q x y, COMM R (f x y) >> -∗ WP e {{ R }}

do_byte_read src Q := AR << ∀ v, src |-> v >> << Q v >>
AR << ∀ x, P x >> << R x >> :=
  AU << ∀ x, P x >> << P x, COMM R x >>
do_byte_write dst v Q := AC << ∀ w, dst |-> w >> << dst |-> v, COMM Q v >>
do_byte_copy src dst Q :=
  do_byte_read src (λ v, do_byte_write dst v Q)
```

- Sufficient for unverified clients: ✅
- Implies disciplined spec: ✅ (sequential ownership suffices to prove AUs)
- Implementable (efficiently): ~✅ (atomics suffice)

Non-deterministic parallel composition

For performance, NOVA does not order reads/writes to different bytes. So our final spec is:

```
do_buf_copy src dst Q :=
  ∃ (Qcopy : N -> mpred),
    (∗_i ∈ [0, 512[ do_byte_copy (src + i) (dst + i) (Qcopy i)) *
    ((∗_i ∈ [0, 512[ Qcopy i) -* Q)
```

Final spec: `do_buf_copy src dst R -* WP ipc_call_copy() {{ R }}`

Sufficient for unverified clients: ✅

Implementable (efficiently): ✅ (relaxed atomics suffice!)

Some metrics: Approximate spec size

Specs for 12 syscalls (out of ~15): 39 commits

- ipc_call requires 7 steps + UTCB copy
- ctrl_sm: 6 steps
- ctrl_pd (selector manipulation): 2 + 2 for each selector
- 24 steps across the other 10 syscalls

We derived sequential specs for most of those.

Conclusions

Undisciplined specs simplify maintenance of kernel specs:

- Single verification of NOVA against undisciplined spec
- Derive disciplined spec
- Conjectured: robustness (robust safety?)
- Less overhead than operational semantics
- Enable end-to-end verification