Modular, Full-Stack Verification

Gregory Malecha, Hoang-Hai Dang, Paolo G. Giarrusso,
Simon Hudon, Jan-Oliver Kaiser (BlueRock Security)
David Swasey (Riverside Research)

gregory@bluerock.io

We can ***specify real, concurrent systems*** in ways that ***naturally align with systems design principles***.

Full verification of real systems code is tractable.

Modular, Full-Stack Verification

Gregory Malecha, Hoang-Hai Dang, Paolo G. Giarrusso,
Simon Hudon, Jan-Oliver Kaiser (BlueRock Security)
David Swasey (Riverside Research)

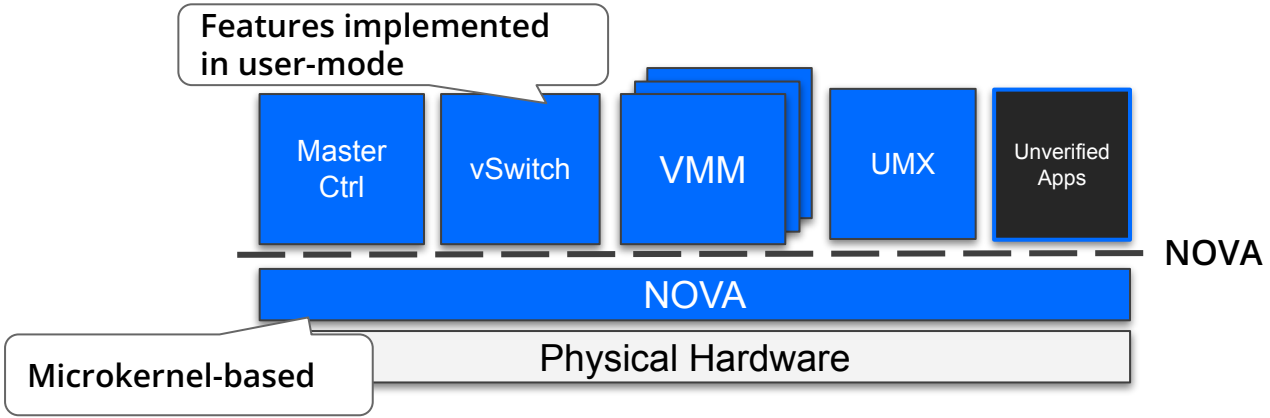
gregory@bluerock.io

The Challenge

A Virtualization Platform

©BlueRock Inc. 2025

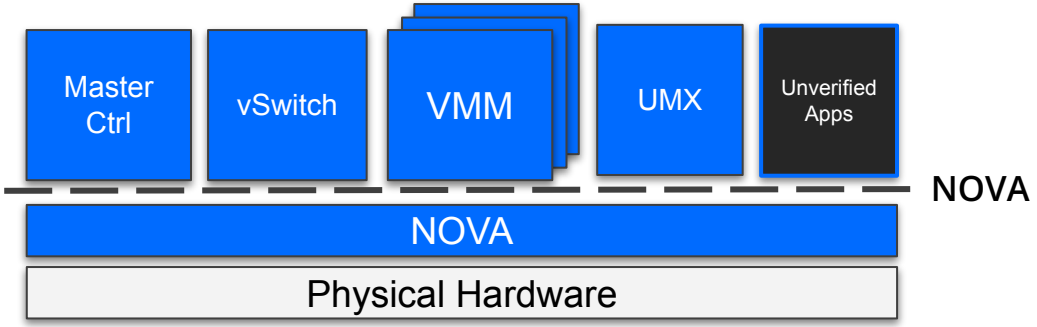
The Challenge
A Virtualization Platform



©BlueRock Inc. 2025

The Challenge
A Virtualization Platform

1. **Real System** – concurrency, dynamic resource sharing, real systems engineers (+mainstream PL, e.g. C++).
2. **Two-sided Specification** – single specification used by both the implementation and the clients.
3. **Modular Verification** – aligned with the architecture of the actual systems engineers.
4. **Robust** – support inter-operation with unverified code.



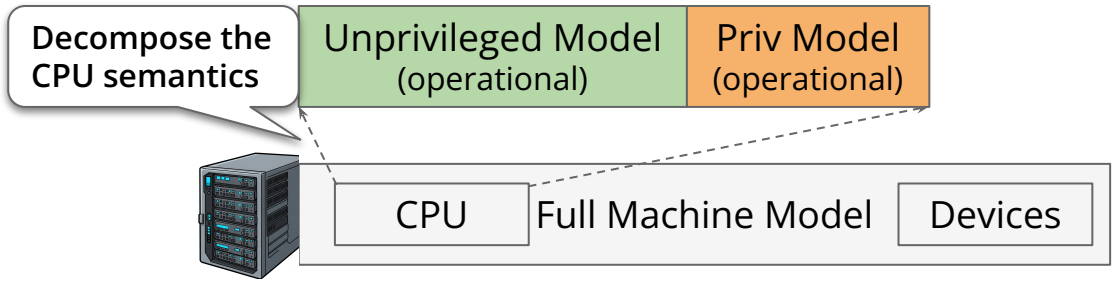
©BlueRock Inc. 2025

The Full-Stack Proof

CPU | Full Machine Model | Devices

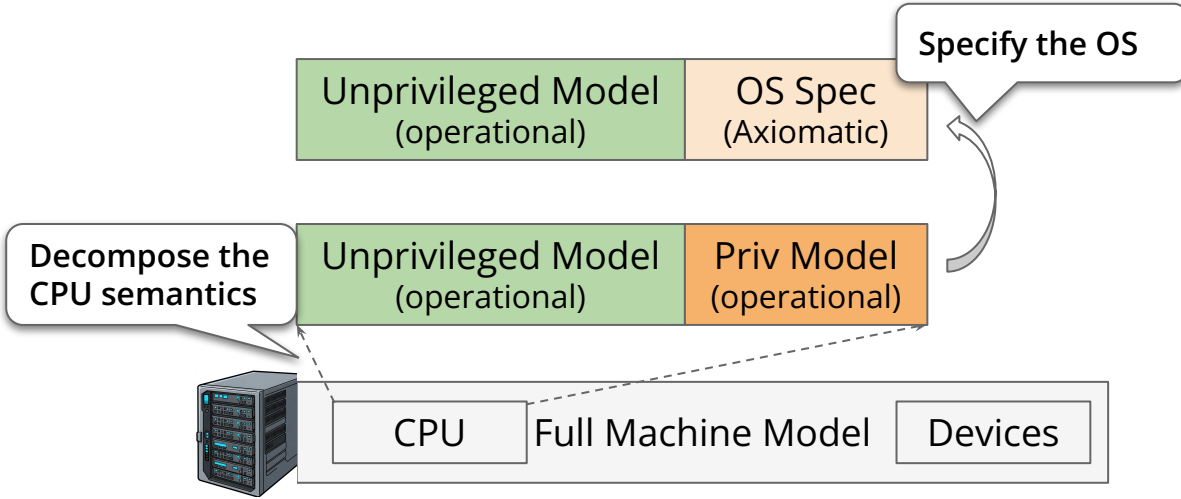
©BlueRock Inc. 2025

The Full-Stack Proof



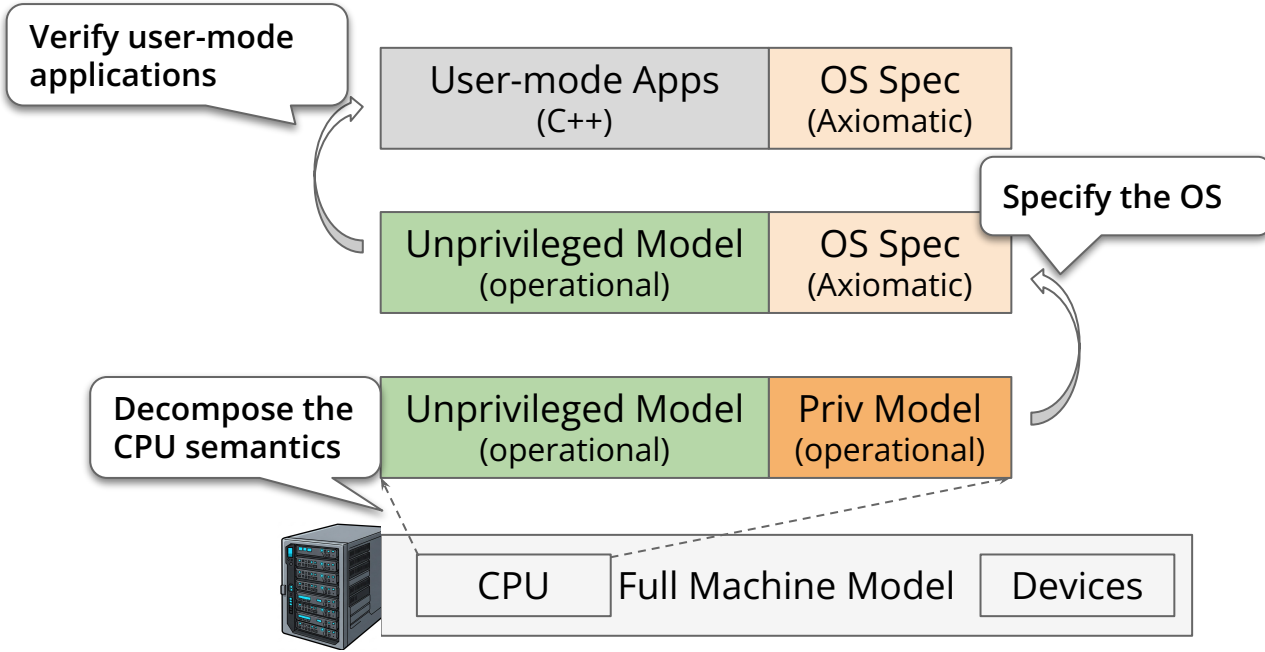
©BlueRock Inc. 2025

The Full-Stack Proof



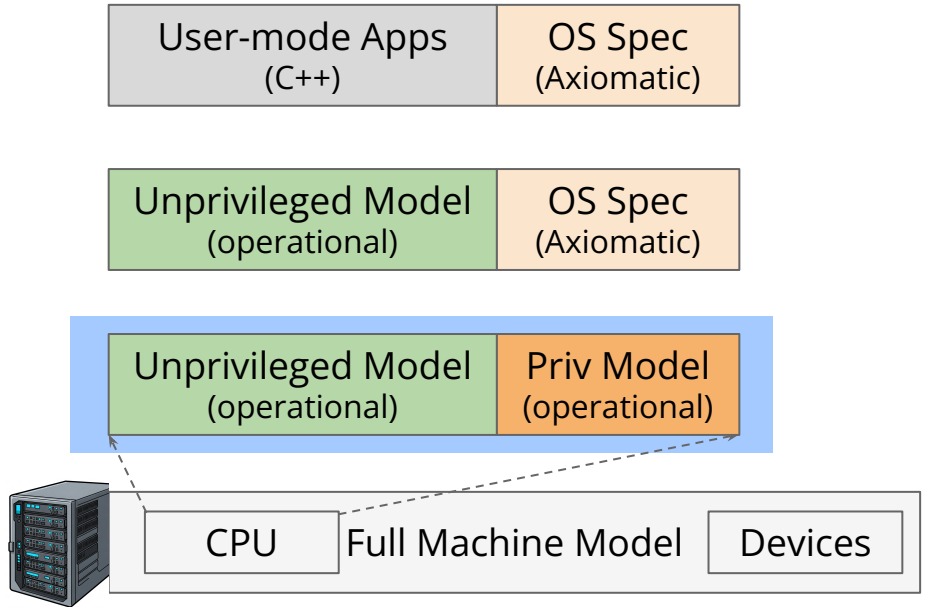
©BlueRock Inc. 2025

The Full-Stack Proof



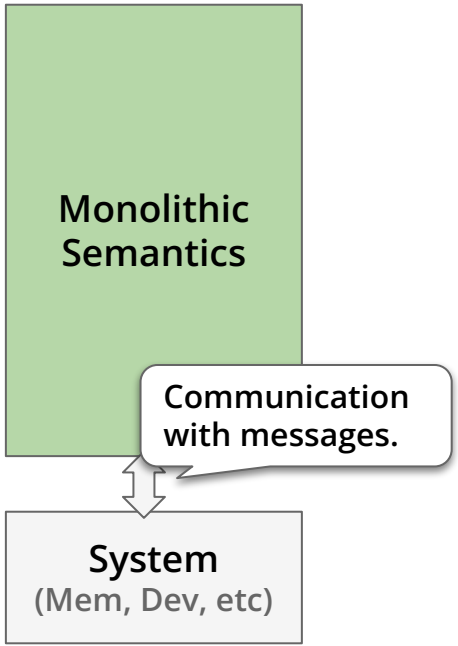
©BlueRock Inc. 2025

The Full-Stack Proof



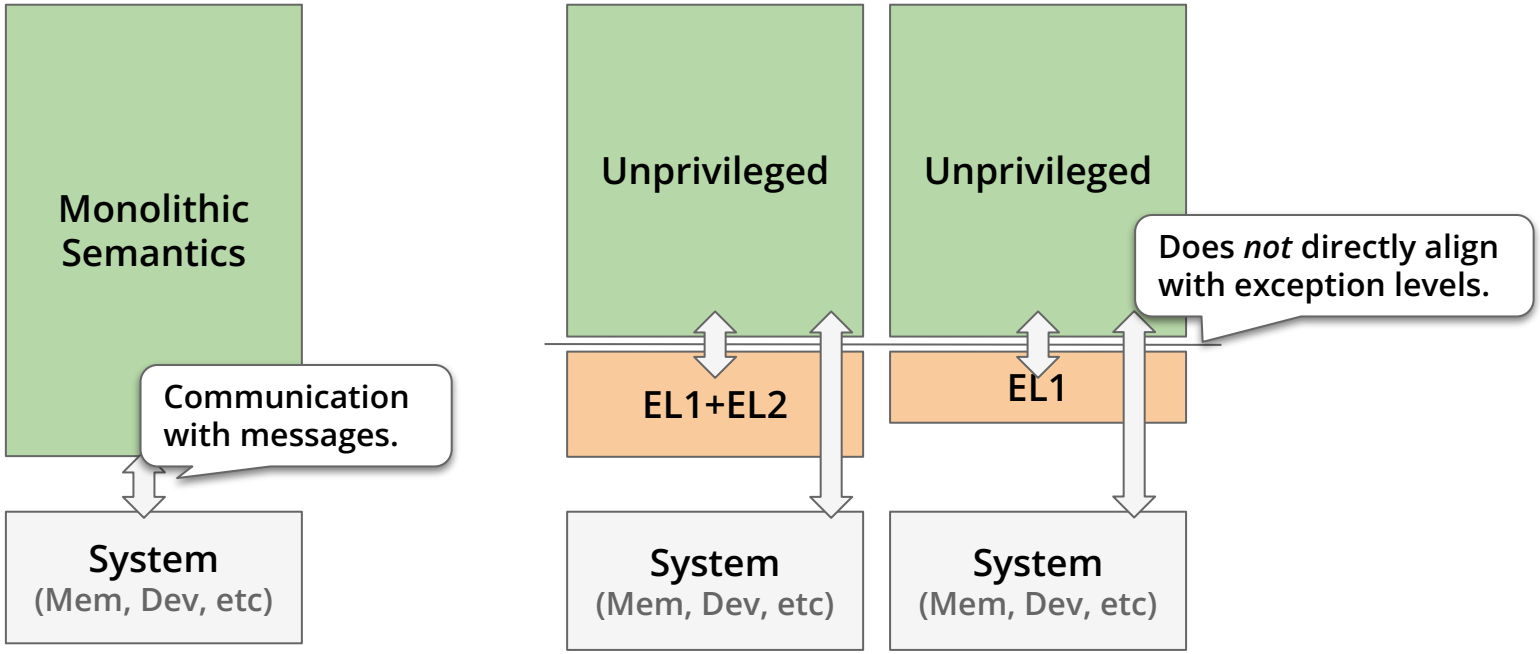
©BlueRock Inc. 2025

Decomposing CPU Semantics



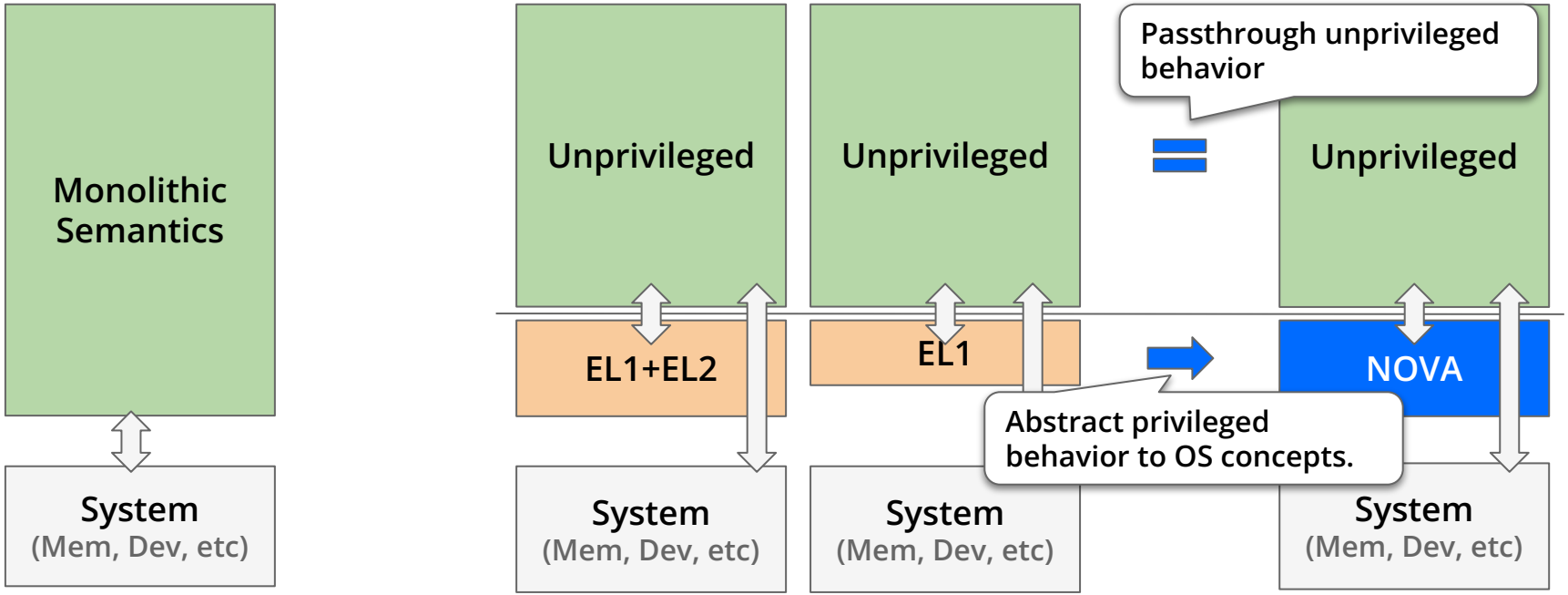
©BlueRock Inc. 2025

Decomposing CPU Semantics



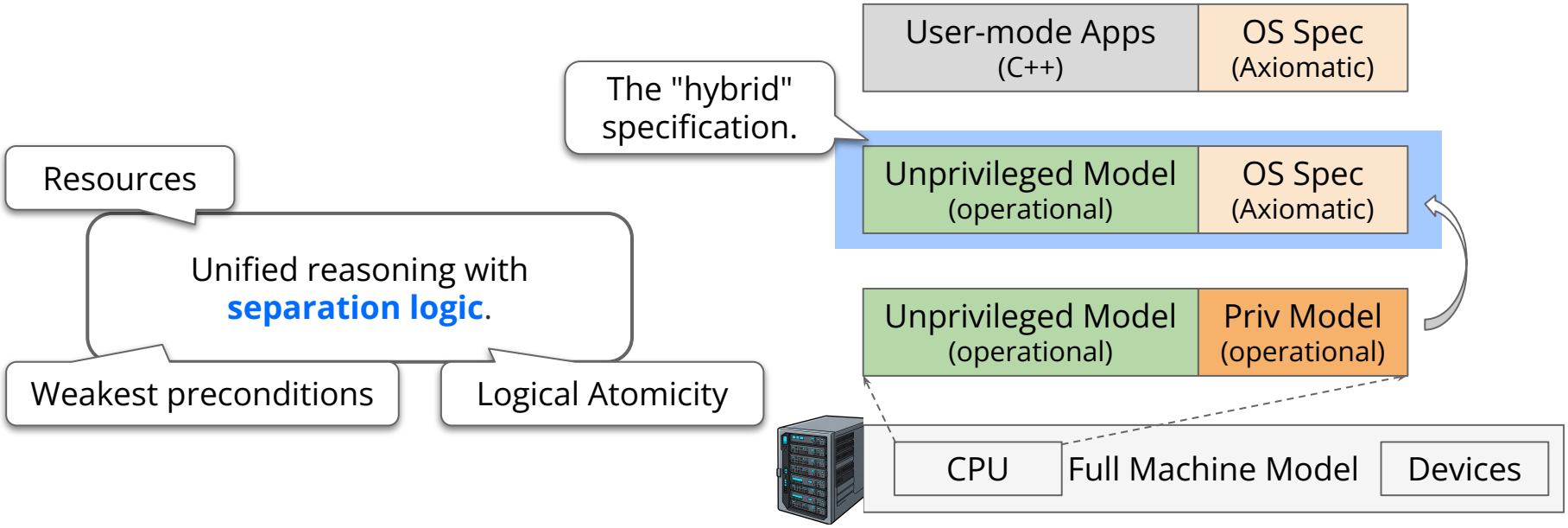
©BlueRock Inc. 2025

Decomposing CPU Semantics



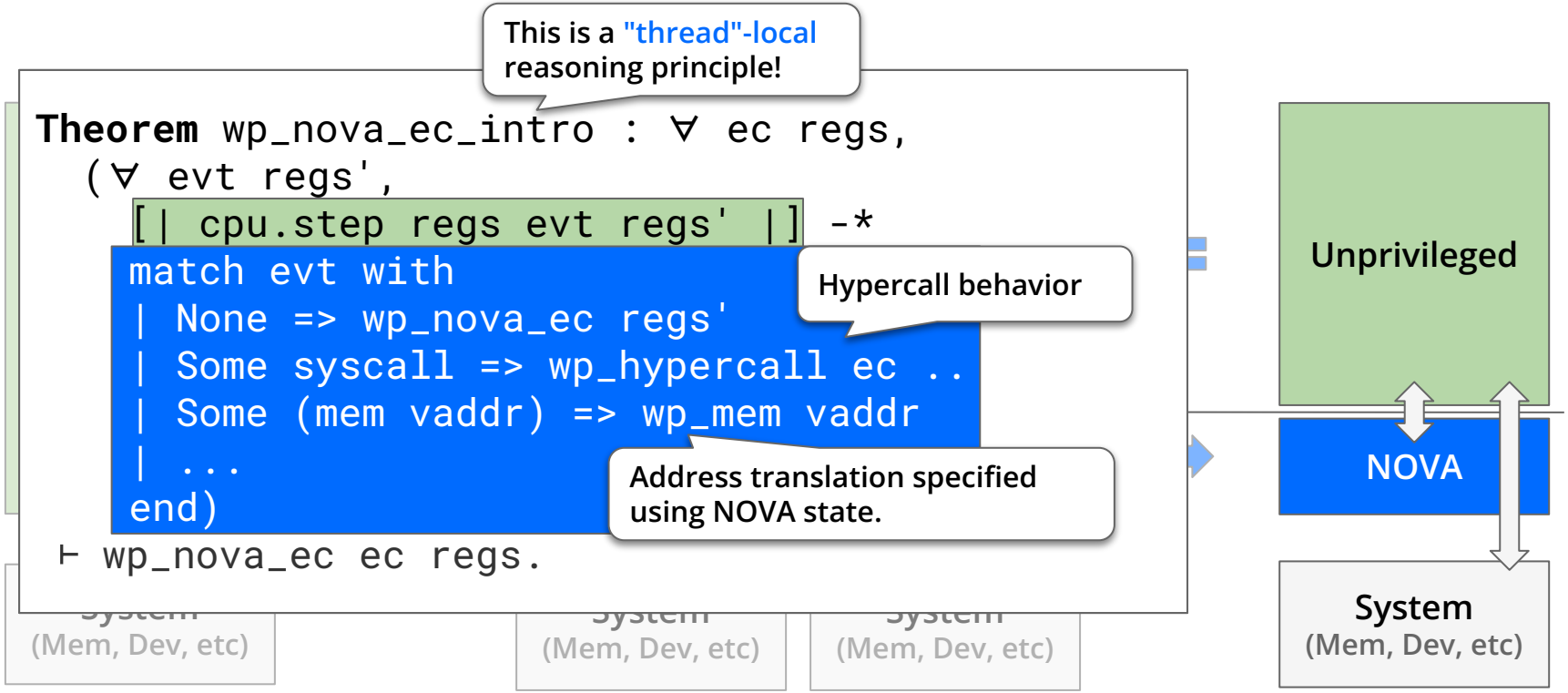
©BlueRock Inc. 2025

The Full-Stack Proof
Specify the OS



©BlueRock Inc. 2025

The NOVA Machine



©BlueRock Inc. 2025

NOVA's Capability API:
kernel objects, hypercalls, and HW-assisted virtualization

PD: protection domain with capabilities in **Object spaces**

EC: execution contexts
SC: scheduling contexts
PT: portals

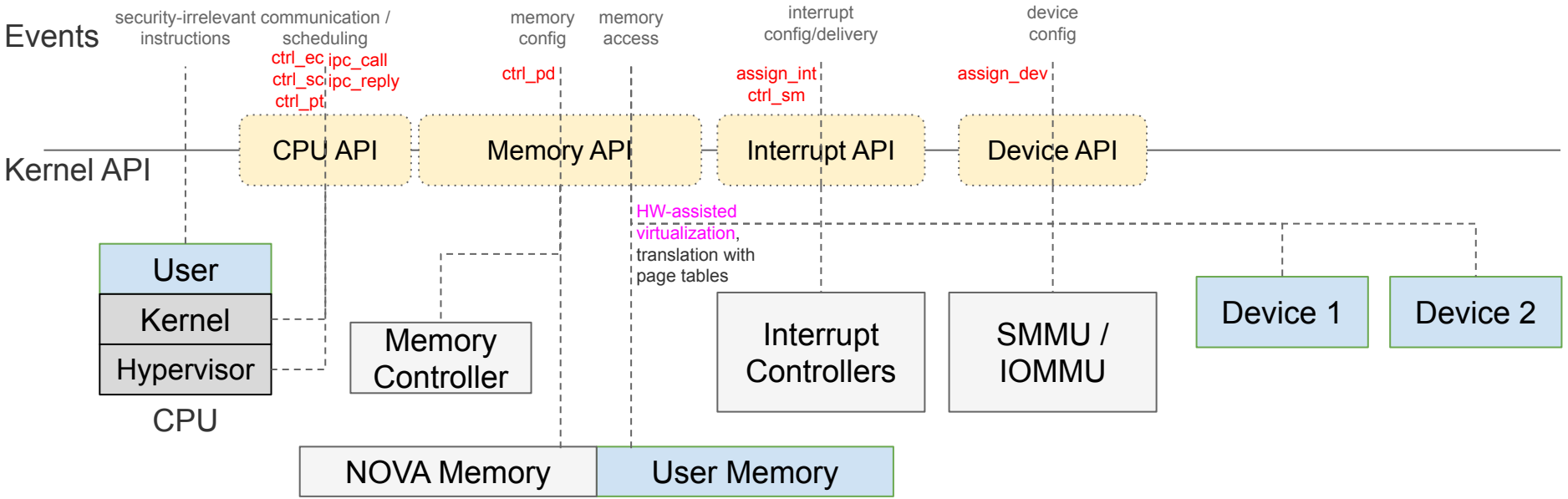
Memory spaces

SM: semaphores

DMA spaces

user-owned

µkernel-owned



©BlueRock Inc. 2025

NOVA's Capability API:
kernel objects, hypercalls, and HW-assisted virtualization

PD: protection domain with capabilities in **Object spaces**

EC: execution contexts
SC: scheduling contexts
PT: portals

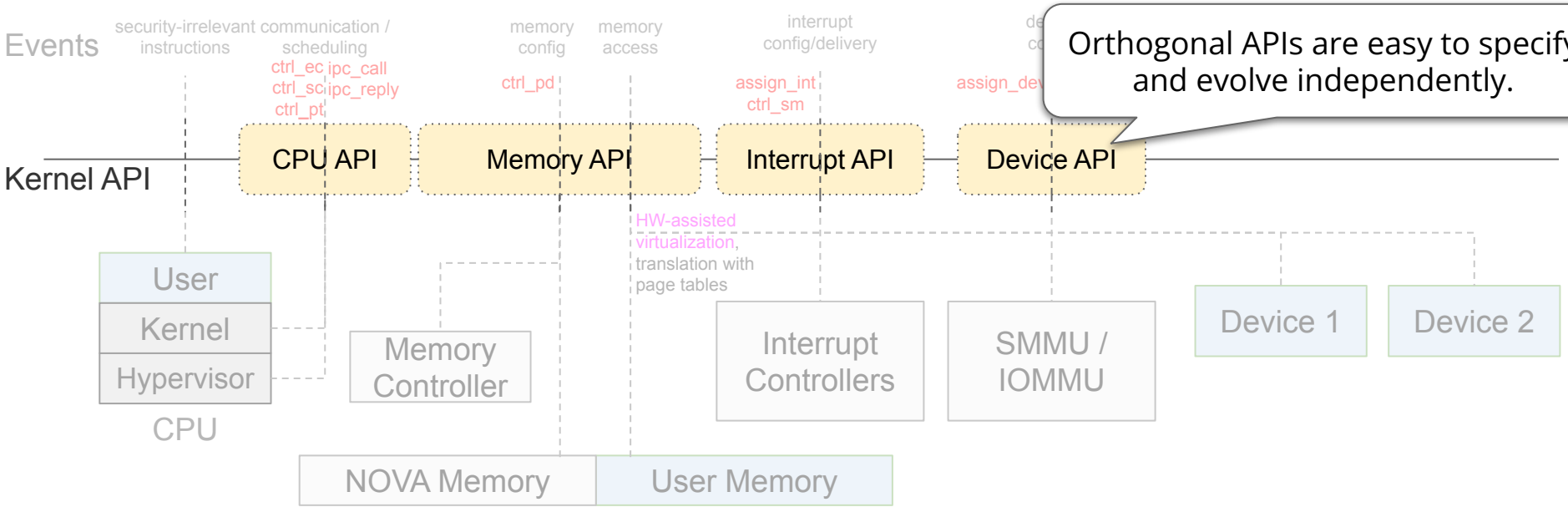
Memory spaces

SM: semaphores

DMA spaces

user-owned

µkernel-owned



©BlueRock Inc. 2025

NOVA's Capability API:
kernel objects, hypercalls, and HW-assisted virtualization

user-owned

µkernel-owned

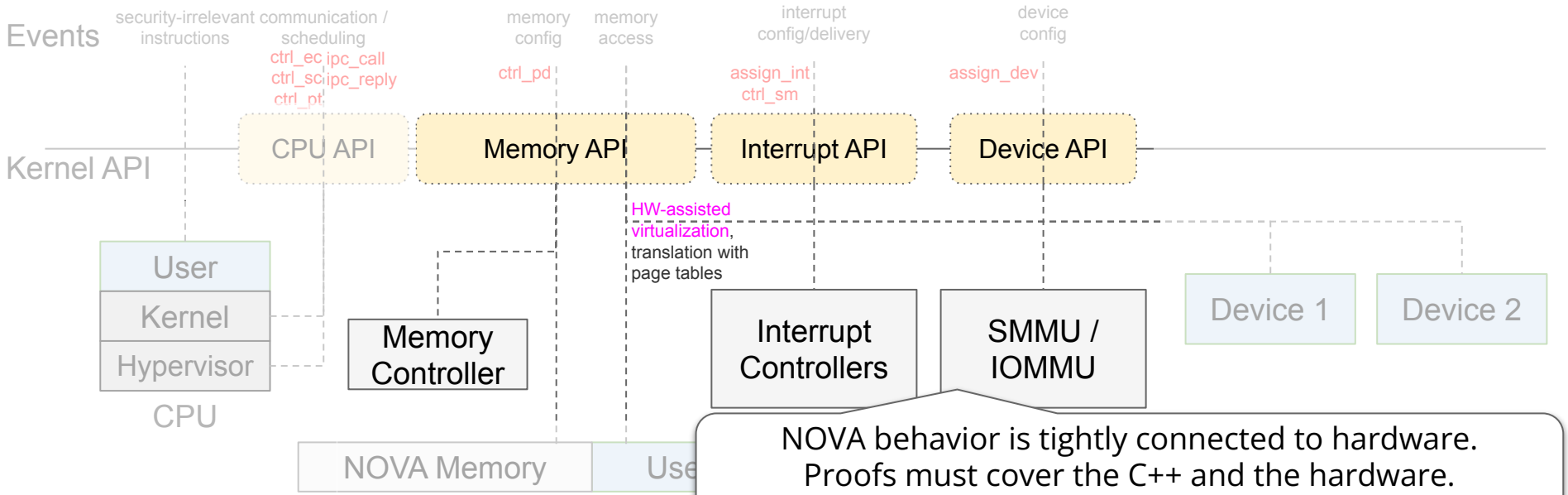
PD: protection domain with capabilities in **Object spaces**

EC: execution contexts
SC: scheduling contexts
PT: portals

Memory spaces

SM: semaphores

DMA spaces



NOVA behavior is tightly connected to hardware.
Proofs must cover the C++ and the hardware.
Concurrency is unavoidable.

©BlueRock Inc. 2025

Fair Semaphores

4.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ  sm,            // Semaphore
                  UINT     ticks);        // Deadline Timeout
```

Flags:

Description:

Prior to the hypercall:

- If **D=0 (Up)**: { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{UP}$.
- If **D=1 (Down)**: { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{DN}$.

If the hypercall completed successfully:

- If **D=0 (Up)**: if there were ECs blocked on the semaphore, then the microhypervisor has released one of those blocked ECs. Otherwise, the microhypervisor has incremented the semaphore counter. The deadline timeout value and the Z-flag were ignored.
- If **D=1 (Down)**: if the semaphore counter was larger than zero, then the microhypervisor has decremented the semaphore counter (**Z=0**) or set it to zero (**Z=1**). Otherwise, the microhypervisor has blocked $EC_{CURRENT}$ on the semaphore. If the deadline timeout value was non-zero, $EC_{CURRENT}$ unblocks with a timeout status when the architectural timer reaches or exceeds the specified ticks value.

Blocking and releasing of ECs on a semaphore uses the FIFO queueing discipline.

Status:

SUCCESS

- The hypercall completed successfully.

TIMEOUT

- If **D=1**: Down operation aborted when the timeout triggered.

OVRFLOW

- If **D=0**: Up operation aborted because the semaphore counter would overflow.

BAD_CAP

- { $PD_{CURRENT}$, SEL_{OBJ} sm } did not refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) or that capability had insufficient permissions.

BAD_CPU

- If **D=1** on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via `assign_int`.

Preliminary

©BlueRock Inc. 2025

Fair Semaphores
State

A minimal piece of independent state and an ownership discipline/"protocol"

Reflected as **separation logic resources**

- Selectors map to capabilities

  pd;sel ↦ (obj, perm)

- The counter of a semaphore

  sm.counter obj n

- The queue of blocked threads

  sm.queue obj ls

4.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ  sm,          // Semaphore
                  UINT     ticks);      // Deadline Timeout
```

Prior to the hypercall:

- If **D=0 (Up)**: { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{UP}$.
- If **D=1 (Down)**: { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{DN}$.

If the hypercall completed successfully:

- If **D=0 (Up)**: if there were ECs blocked on the semaphore, then the microhypervisor has released one of those blocked ECs. Otherwise, the microhypervisor has incremented the semaphore counter. The deadline timeout value and the Z-flag were ignored.
- If **D=1 (Down)**: if the semaphore counter was larger than zero, then the microhypervisor has decremented the semaphore counter (**Z=0**) or set it to zero (**Z=1**). Otherwise, the microhypervisor has blocked $EC_{CURRENT}$ on the semaphore. If the deadline timeout value was non-zero, $EC_{CURRENT}$ unblocks with a timeout status when the architectural timer reaches or exceeds the specified ticks value.

Blocking and releasing of ECs on a semaphore uses the FIFO queueing discipline.

Status:

SUCCESS

- The hypercall completed successfully.

TIMEOUT

- If **D=1**: Down operation aborted when the timeout triggered.

OVRFLOW

- If **D=0**: Up operation aborted because the semaphore counter would overflow.

BAD_CAP

- { $PD_{CURRENT}$, SEL_{OBJ} sm } did not refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) or that capability had insufficient permissions.

BAD_CPU

- If **D=1** on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via `assign_int`.

Preliminary

©BlueRock Inc. 2025

Fair Semaphores
Behavior

Reflected as **separation logic resources**

- Selectors map to capabilities

  pd;sel ↦ (obj, perm)

- The counter of a semaphore

  sm.counter obj n

- The queue of blocked threads

  sm.queue obj ls

Sequencing

```
  let (Some obj) :=
    resolve_sm ec.pd {UP} sel else K BAD_CAP;
  sm.incr obj K
⊢ wp_hypercall ec ctrl_sm(0, sel, _) K
```

Continuation.

4.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ sm,          // Semaphore
                  UINT    ticks);      // Deadline Timeout
```

Flags:

0	0	Z	D
3	2	1	0

Description:

Prior to the hypercall:

- If D=0 (Up): { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{UP}$.
- If D=1 (Down): { $PD_{CURRENT}$, SEL_{OBJ} sm } must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with permission $CTRL_{DN}$.

If the hypercall completed successfully:

- If D=0 (Up): if there were ECs blocked on the semaphore, then the microhypervisor has released one of those blocked ECs. Otherwise, the microhypervisor has incremented the semaphore counter. The deadline timeout value and the Z-flag were ignored.

- If D=1: Down operation aborted when the time

OVRFLOW

- If D=0: Up operation aborted because the semaphore counter would overflow.

BAD_CAP

- { $PD_{CURRENT}$, SEL_{OBJ} sm } did not refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) or that capability had insufficient permissions.

BAD_CPU

- If D=1 on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via assign_int.

Preliminary

©BlueRock Inc. 2025

Fair Semaphores

Behavi

Reflected

- Sele

  pd

- Th

  sm.counter obj n

- The queue of blocked threads

  sm.queue obj ls

```
sm.incr g K :=
  AU<< ∃ n. sm.counter g n >> @N
    << sm.counter g (cap (n+1)),
       COMM K (if overflows (n+1)
               then OVERFLOW else SUCCESS) >>
```

Exchange this...

...for this

Afterwards, continue

```
    resolve_sm ec.pd {UP} sel else K BAD_CAP;
  sm.incr obj K
⊢ wp_hypercall ec ctrl_sm(0, sel, _) K
```

4.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ sm,          // Semaphore
                  UINT    ticks);      // Deadline Timeout
```

- If D=1: Down operation aborted when the timeout triggered.

OVRFLOW

- If D=0: Up operation aborted because the semaphore counter would overflow.

BAD_CAP

- { PD_CURRENT, SEL_OBJ sm } did not refer to a SM Object Capability (CAP_OBJ_SM) or that capability had insufficient permissions.

BAD_CPU

- If D=1 on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via assign_int.

©BlueRock Inc. 2025

Fair Semaphores
Behavior

Reflected as **separation logic resource**

- Selectors map to capabilities

  pd;sel ↦ (obj, perm)

- The counter of a semaphore

  sm.counter obj n

- The queue of blocked threads

  sm.queue obj ls

```
resolve_sm pd sel K :=
  AU<<∃ obj,perm. pd;sel↦(obj,perm)>> @NOVA
    <<ec.pd;sel↦(obj,perm),
      COMM K (if {UP} ⊆ perm
              then Some obj else None)>>
```

Two atomic steps.

```
let (Some obj) :=
  resolve_sm ec.pd {UP} sel else K BAD_CAP;
sm.incr obj K
```

```
sm.incr g K :=
  AU<<∃ n. sm.counter g n>> @NOVA
    <<sm.counter g (cap (n+1)),
      COMM K (if overflows (n+1)
              then OVERFLOW else SUCCESS)>>
```

4.4.5 Control Semaphore

Parameters:

```
status = ctrl_sm (SEL_OBJ sm,          // Semaphore
                  UINT    ticks);      // Deadline Timeout
```

Flags:

0	0	Z	D
3	2	1	0

Description:

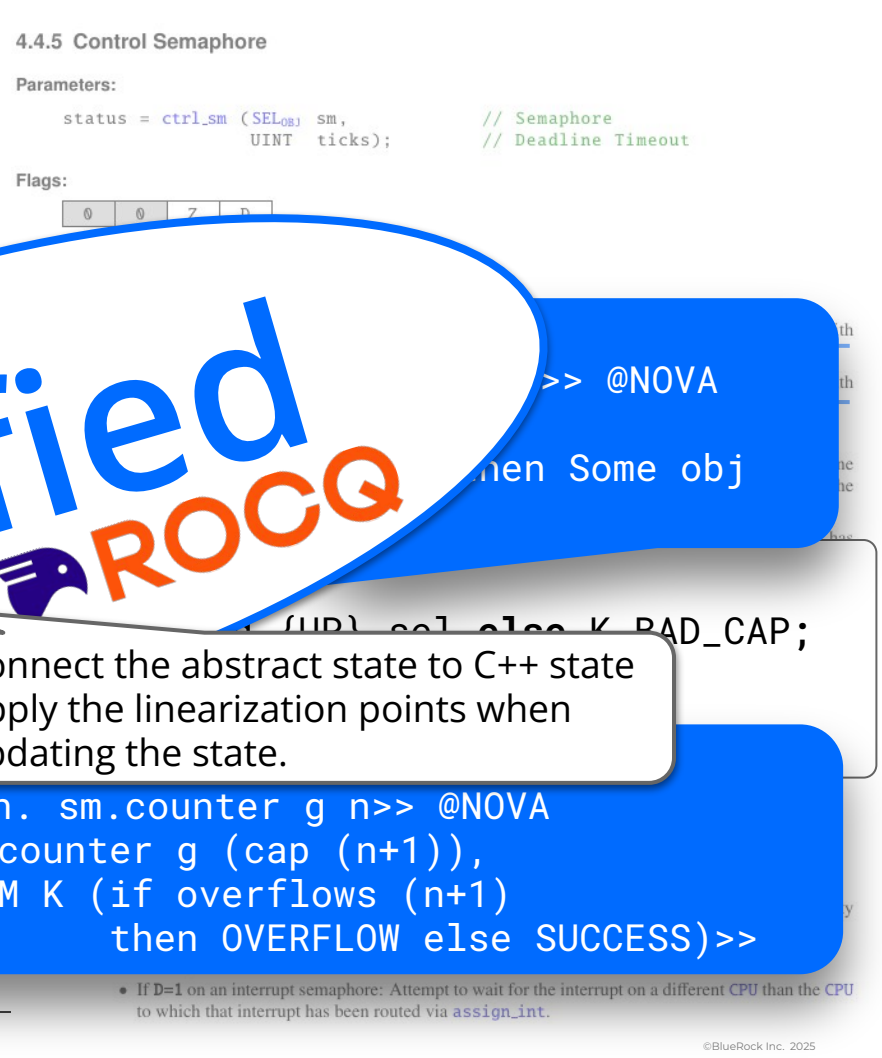
- If D=1 on an interrupt semaphore: Attempt to wait for the interrupt on a different CPU than the CPU to which that interrupt has been routed via assign_int.

©BlueRock Inc. 2025

Fair Semaphores
Behavior

Reflected as **separation logic resource**

- Selectors map to capa
  - pd;sel ↦ (obj, pe
- The counter o
  - sm.counter
- The queue of blocked
  - sm.queue obj ls



Verified ROCQ

1. Connect the abstract state to C++ state
2. Apply the linearization points when updating the state.

©BlueRock Inc. 2025

Spawning Threads
Concurrent Behavior

Thread creation occurs when scheduling contexts are bound to global execution contexts.

The proof obligation for a thread creation is expressed as a WP for the new thread.

5.3.3 Create Scheduling Context

Parameters:

```
status = create_sc (SEL_OBJ sel,        // Created SC
                    SEL_OBJ pd,         // Owner PD
                    SEL_OBJ ec,         // Bound EC
                    SCD     scd)        // Scheduling Context Descriptor
```

Flags:

0	0	0	0
3	2	1	0

Description:

Creates a new Scheduling Context (SC).

Prior to the hypercall:

- $SPC_{OBJ_{CURRENT}}$[sel] must refer to a Null Capability (CAP_0).
- $SPC_{OBJ_{CURRENT}}$[pd] must refer to a PD Capability ($CAP_{OBJ_{PD}}$) with permission SC.
- $SPC_{OBJ_{CURRENT}}$[ec] must refer to an EC Capability ($CAP_{OBJ_{EC}}$) with permission $BIND_{SC}$.

If the hypercall completed successfully:

- A new Scheduling Context (SC) has been created.
- The created SC is bound to the EC referred to by $SPC_{OBJ_{CURRENT}}$[ec] on the CPU of that EC, with its scheduling parameters set according to scd.
- The resources for the created SC were accounted to the PD referred to by $SPC_{OBJ_{CURRENT}}$[pd].

```
  let g_pd := resolve_pd ec.pd sel_pd {SC} Q in
  let g_ec := resolve_ec ec.pd sel_ec {EC_BIND_SC} Q in
  let g_sc := sc.create ec.pd sel_sc Q in
  let run := ec.bind g_ec g_sc in

  (if run then wp_nova_ec g_ec else emp)* K SUCCESS
⊢ wp_hypercall ec create_sc(sel_sc, sel_pd, sel_ec, scd) K
```

BAD_PAR

- At least one SCD field in scd was invalid.

MEM_OBJ

- The Protection Domain referred to by $SPC_{OBJ_{CURRENT}}$[pd] had insufficient memory resources for

Spawning Threads
Concurrent Behavior

Thread creation occurs when scheduling contexts are bound to global execution contexts.

The proof obligation for a thread creation is expressed as a WP for the new thread.

5.3.3 Create Scheduling Context

Parameters:

```
status = create_sc (SEL_OBJ sel,        // Created SC
                    SEL_OBJ pd,         // Owner PD
                    SEL_OBJ ec,         // Bound EC
                    SCD     scd)        // Scheduling Context Descriptor
```

Flags:

Description:

Creates a new Scheduling Context (SC).

Prior to the hypercall:

- $SPC_{OBJ_{CURRENT}}$[sel] must refer to a Null Capability (CAP_0).
- $SPC_{OBJ_{CURRENT}}$[pd] must refer to a PD Capability ($CAP_{OBJ_{PD}}$) with permission SC.
- $SPC_{OBJ_{CURRENT}}$[ec] must refer to an EC Capability ($CAP_{OBJ_{EC}}$) with permission $BIND_{SC}$.

If the hypercall completed successfully:

- A new Scheduling Context (SC) has been created.
- The created SC is bound to the EC referred to by $SPC_{OBJ_{CURRENT}}$[ec] on the CPU of that EC, with its scheduling parameters set according to scd.
- The resources for the created SC were accounted to the PD referred to by $SPC_{OBJ_{CURRENT}}$[pd].

```
let g_pd := resolve_pd ec.pd sel_pd {SC} Q in
let g_ec := resolve_ec ec.pd sel_ec {EC_BIND_SC} Q in
reate ec.pd sel_sc Q in
nd g_ec g_sc in

(if run then wp_nova_ec g_ec else emp)* K SUCCESS

⊢ wp_hypercall ec create_sc(sel_sc, sel_pd, sel_ec, scd) K
```

Prove an extra WP for the new thread.

Continue running the current thread.

BAD_PAR

- At least one SCD field in scd was invalid.

MEM_OBJ

- The Protection Domain referred to by $SPC_{OBJ_{CURRENT}}$[pd] had insufficient memory resources for

Spawning Threads
Concurrent Behavior

Thread creation occurs when scheduling contexts are bound to global execution contexts.

The proof obligation for a thread creation is expressed as a WP for the new thread.

5.3.3 Create Scheduling Context

Parameters:

```
status = create_sc (SEL_OBJ sel,     // Created SC
                    SEL_OBJ pd,      // Owner PD
                    SEL_OBJ ec,      // Bound EC
                    SCD     scd)     // Scheduling Context Descriptor
```

Flags:

0	0	0	0
3	2	1	0

Description:

Creates a new Scheduling Context (SC).

Prior to the hypercall:

- $SPC_{OBJ_{CURRENT}}$[sel] must refer to a Null Capability (CAP_0).
- $SPC_{OBJ_{CURRENT}}$[pd] must refer to a PD Capability ($CAP_{OBJ_{PD}}$) with permission SC.
- $SPC_{OBJ_{CURRENT}}$[ec] must refer to an EC Capability ($CAP_{OBJ_{EC}}$) with permission $BIND_{SC}$.

If the hypercall completed successfully:

- A new Scheduling Context (SC) has been created.
- The created SC is bound to the EC referred to by $SPC_{OBJ_{CURRENT}}$[ec] on the CPU of that EC, with its

PD referred to by $SPC_{OBJ_{CURRENT}}$[pd].

```
  let g_pd := resolve_pd                      in
  let g_ec := resolve_                 _BIND_SC) 0 in
             reat
             nd g
  (if run then wp_nova_ec g_ec else emp) * K SUCCESS
⊢ wp_hypercall ec create_sc(sel_sc, sel_pd, sel_ec, scd) K
```

Need logical atomicity to make this complete.

Prove an extra WP for the new thread.

Independence is the core of concurrency!

Continue running the current thread.

BAD_PAR

- At least one SCD field in scd was invalid.

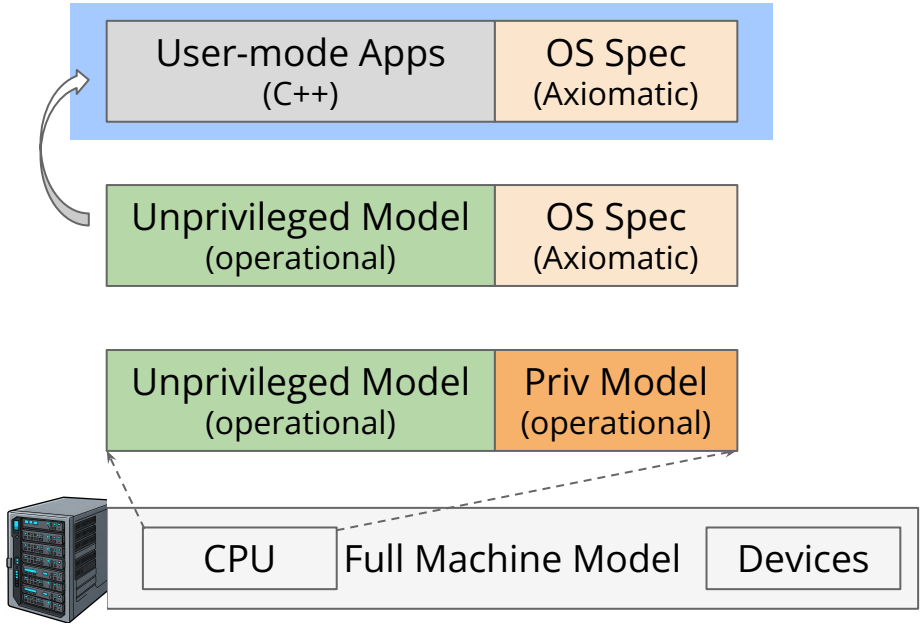
MEM_OBJ

- The Protection Domain referred to by $SPC_{OBJ_{CURRENT}}$[pd] had insufficient memory resources for

Malecha, *et. al*. **Developing with Formal Methods at BedRock Systems**. IEEE S&P
Giarrusso, *et. al*. **Verifying a Virtual Machine Monitor.** BlueRock Technical Report

The Full-Stack Proof
User-mode Verification

- Device drivers.
  - PL011
  - (Simple-mode) Zynqmp
- Terminal multiplexor
- VIRTIO-based layer 2 switch.
- Virtual Machine Monitor



Unified reasoning with separation logic.

Highly reusable proofs

©BlueRock Inc. 2025

Uniform Reasoning in Separation Logic

```
Definition portal_spec_cpu
    (exit_reason : evt.t nova.g                                    )
    (γg : GlobalRoundupInfo.ghos
    (cγ : bm.cpu.Name → Vcpu.gna
    (shareds : bm.cpu.Name → Vcpu.shared_t)
    (bm_cid : bm.cpu.Name)
    : WpSpec_cpp :=
  (** Sequential ownership of current VCPU -- C++/NOVA *)
  \let γvcpu := cγ bm_cid
  \pre{m_seq : Vcpu.seq_t} vcpu |-> Vcpu.seqR (cQp.m 1) bm_cid γvcpu m_seq

  (** Shared handle to the <<Board>> -- C++/NOVA *)
  \let pd := (Model.Cpu.pd_name (Vcpu.base_baseγ γvcpu))
  \prepost{γboard q_board m_board} Model.Board.P pd q_board γboard m_board
  \require γboard .^ Model.BoardImpl.Model._stage = Model.Board.SteadyState

  (** Shared handles to other VCPUs -- C++/NOVA *)
  \prepost{q_vcpus} _global "Model::vcpus" |-> Vcpu.vcpusR cγ shareds q_vcpus γg

  (** VCPU registers -- NOVA *)
  \prepost{vcpu_regs : arch.regs_t guest}
    Model.Cpu.ecRegs (Vcpu.base_baseγ γvcpu) vcpu_regs

  (** Specification state of the Guest -- spec  *)
  \pre{bm_state : lts_state (bm.cpu.lts _)} bm.cpu.core bm_cid bm_state
  \require{decode_assist} cpu.pendingInstruction bm_state decode_assist
  \require{fault_regs} VCPU.FaultRegs.reg_accessor_encodes_decodeassist fault_regs
                       exit_reason decode_assist

  (** Spec & Implementation state are related! *)
  \require CPU.related_full_states bm_state vcpu_regs

  (* ... other ownership ... *)
```

User-mode entry point (from hardware virtualization)

BlueRock

Uniform Reasoning in Separation Logic

```
Definition portal_spec_cpu
    (exit_reason : evt.t nova.g                                    )
    (γg : GlobalRoundupInfo.ghos
    (cγ : bm.cpu.Name → Vcpu.gna
    (shareds : bm.cpu.Name → Vcpu.shared_t)
    (bm_cid : bm.cpu.Name)
    : WpSpec_cpp :=
  (** Sequential ownership of current VCPU -- C++/NOVA *)
  \let γvcpu := cγ bm_cid
  \pre{m_seq : Vcpu.seq_t} vcpu |-> Vcpu.seqR (cQp.m 1) bm_cid γvcpu m_seq

  (* Shared handle to the <<Board>> -- C++/NOVA *)
  \let pd := (Model.Cpu.pd_name (Vcpu.base_baseγ γvcpu))
  \prepost{γboard q_board m_board} Model.Board.P pd q_board γboard m_board
  \require γboard .^ Model.BoardImpl.Model._stage = Model.Board.SteadyState

  (** Shared handles to other VCPUs -- C++/NOVA *)
  \prepost{q_vcpus} _global "Model::vcpus" |-> Vcpu.vcpusR cγ shareds q_vcpus γg

  (** VCPU registers -- NOVA *)
  \prepost{vcpu_regs : arch.regs_t guest}
    Model.Cpu.ecRegs (Vcpu.base_baseγ γvcpu) vcpu_regs

  (** Specification state of the Guest -- spec  *)
  \pre{bm_state : lts_state (bm.cpu.lts _)} bm.cpu.core bm_cid bm_state
  \require{decode_assist} cpu.pendingInstruction bm_state decode_assist
  \require{fault_regs} VCPU.FaultRegs.reg_accessor_encodes_decodeassist fault_regs
                          exit_reason decode_assist

  (** Spec & Implementation state are related! *)
  \require CPU.related_full_states bm_state vcpu_regs

  (* ... other ownership ... *)
```

User-mode entry point (from hardware virtualization)

C++ state.

and some NOVA state

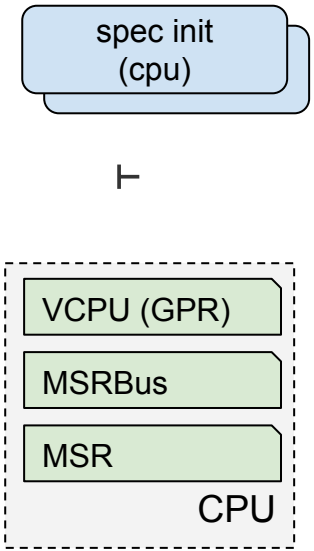
NOVA State

Specification state

Simulation

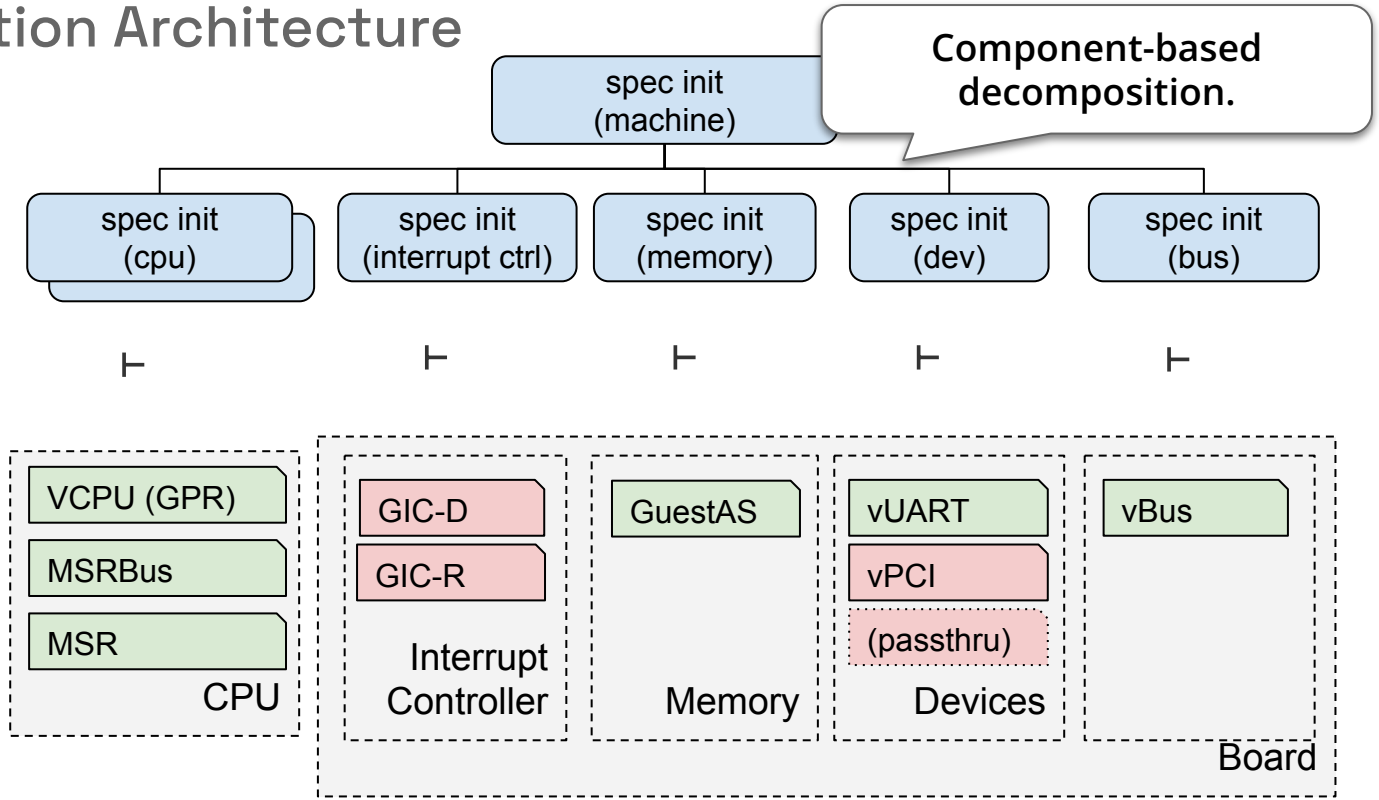
BlueRock

Virtual Machine Monitor
Verification Architecture



©BlueRock Inc. 2025

Virtual Machine Monitor
Verification Architecture



©BlueRock Inc. 2025

Takeaways

- Separation logic is a powerful formalism for reasoning about dynamic systems.
- It can be used to specify complex, concurrent behavior.
- It can be used to *modularly verify* complex, concurrent behavior.
- It works for the (good) *code that you have right now*.

The Future

- Specify your next system using SL!
- Verify the implementation!

Future Research

- Refine low-level hardware models
  - Security reasoning.
  - Weak memory.
- Liveness verification.
  - Availability.
- Hyper-properties.
  - Confidentiality.
  - Isolation.

©BlueRock Inc. 2025