BEDROCK Systems Inc

Towards Modular Specification and Verification of Concurrent Hypervisor-based Isolation

Hoang-Hai Dang, David Swasey, **Gregory Malecha**

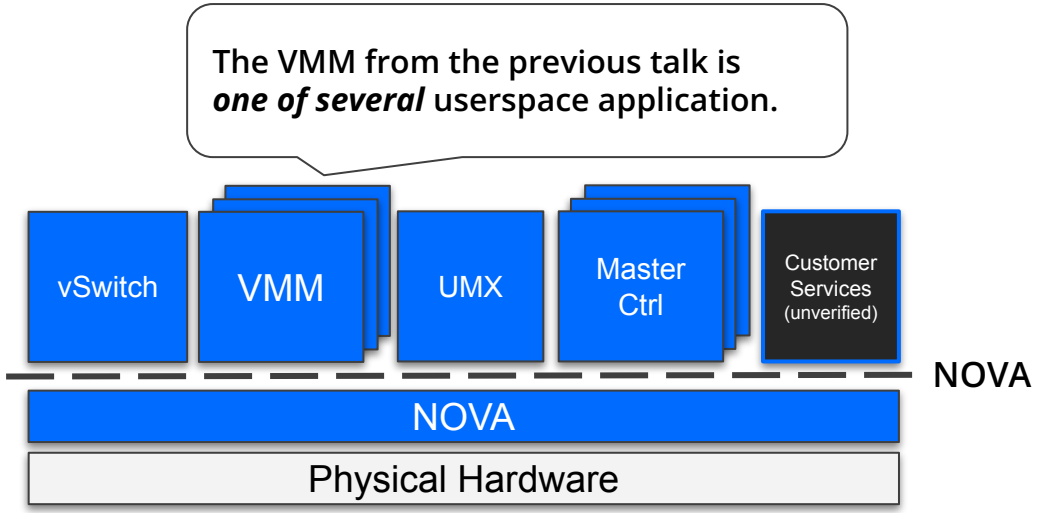
BedRock Systems Inc.

The BedRock Stack

- Microkernel-based systems stack.
- Provides virtualization functionality.
- Kernel security monitors can introspect running guests.
  - Monitoring
  - Protection
  - Remediation

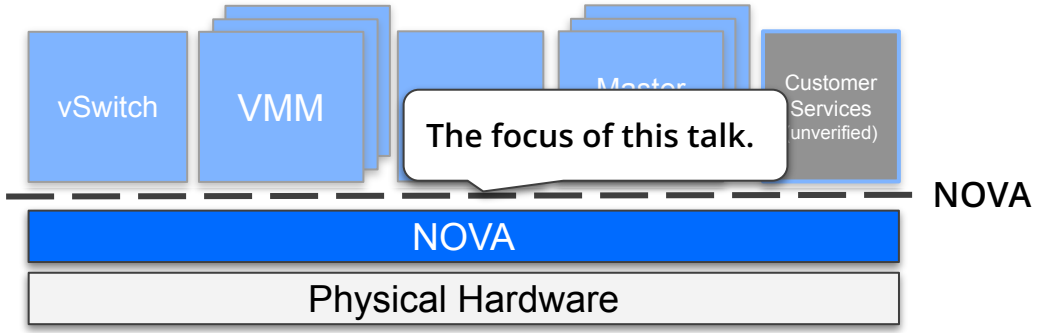
Big Ambitious Goal

Verify the stack against a strong user-space specification. E.g. "bare metal property".

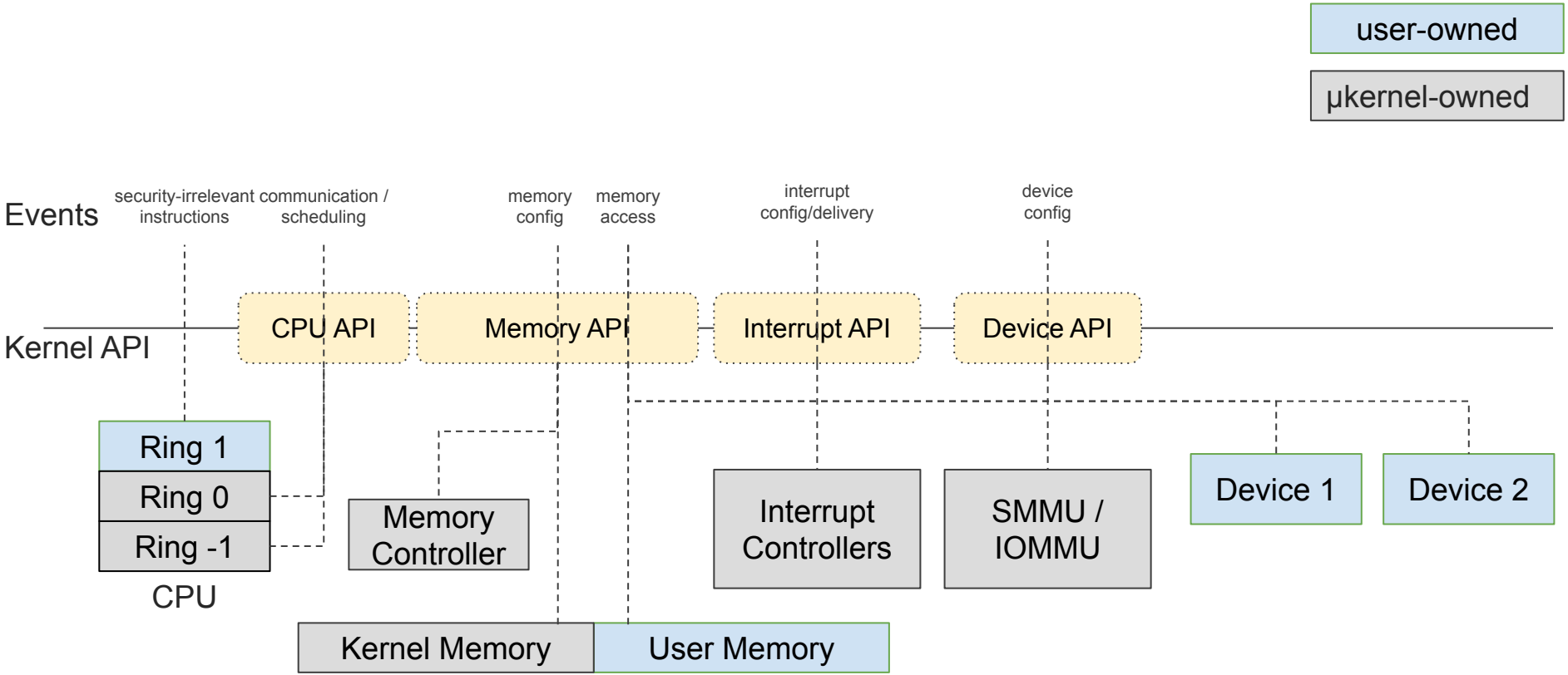


The NOVA Microkernel

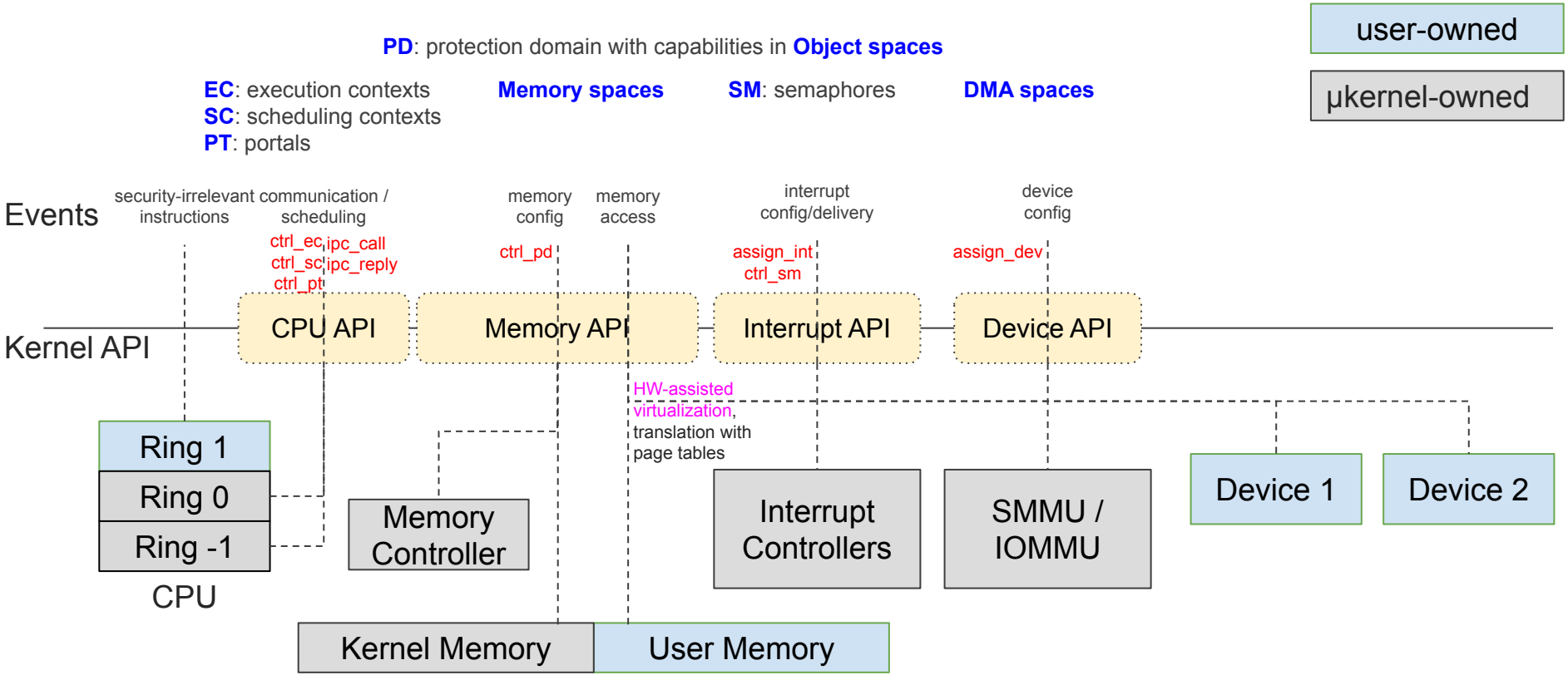
- Capability-based API supporting dynamic object creation.
- Provides low-level mechanisms for working with system resources, e.g. virtual memory, interrupts, etc.
- Communication using semaphores, IPC, and shared memory.
- Exposes both host threads and "virtual CPUs" (for hardware virtualization)
- User threads are *completely untrusted*.



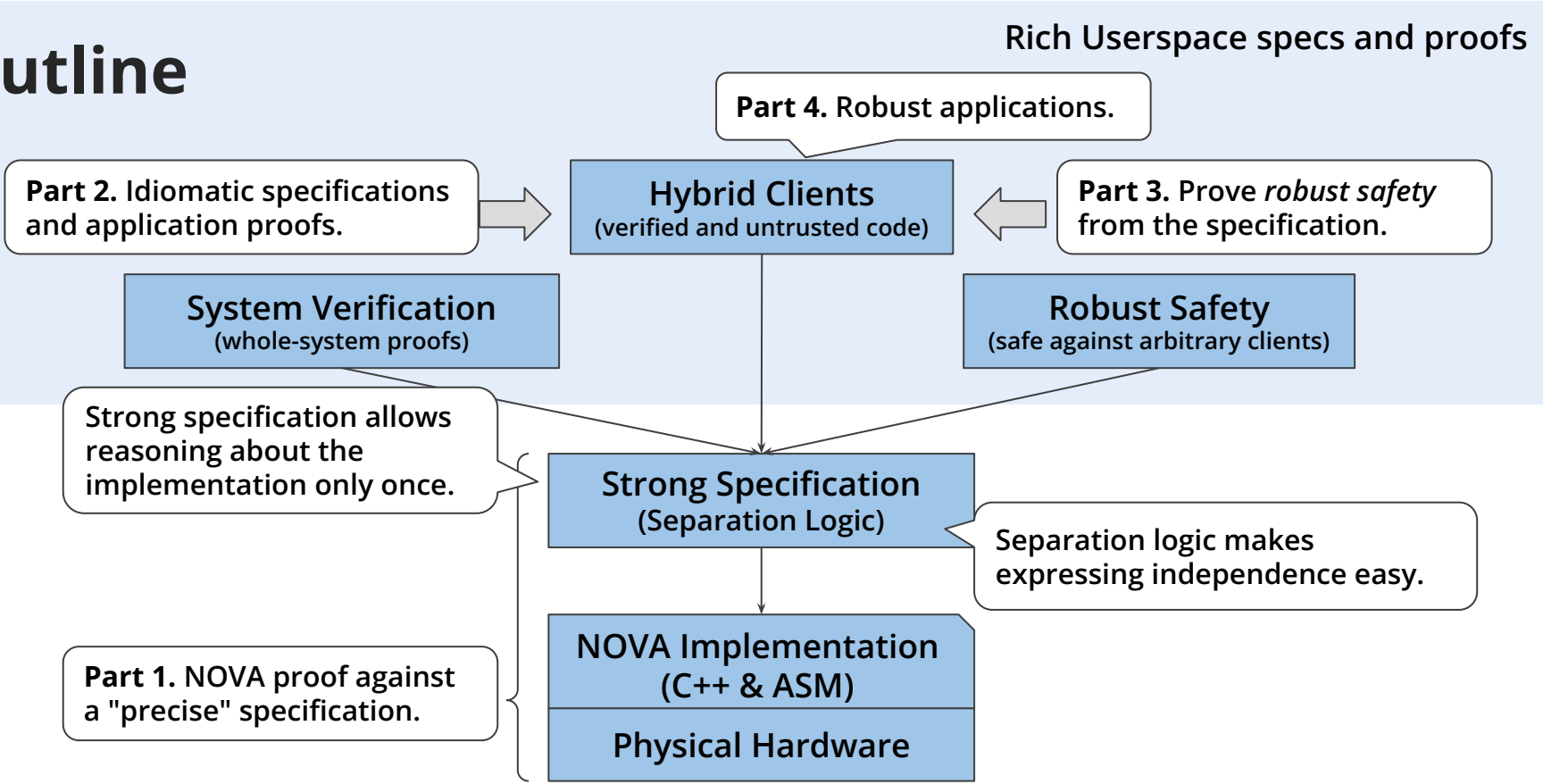
Microkernel owns minimum, security-relevant resources



NOVA exposes kernel objects and hypercalls and exposes HW-assisted virtualization



Outline


Rich Userspace specs and proofs
Part 4. Robust applications.
Part 2. Idiomatic specifications and application proofs.
Hybrid Clients
(verified and untrusted code)
Part 3. Prove robust safety from the specification.
System Verification
(whole-system proofs)
Robust Safety
(safe against arbitrary clients)
Strong specification allows reasoning about the implementation only once.
Strong Specification
(Separation Logic)
Separation logic makes expressing independence easy.
Part 1. NOVA proof against a "precise" specification.
NOVA Implementation
(C++ & ASM)
Physical Hardware

Fair Semaphores w/ Timeouts

State – reflected as separation logic resources

- A value (N)

  sm.value g n

- A queue of blocked ECs

  sm.queue g ls

Implement a FIFO discipline to get fairness

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)**: 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.

tus:

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`.

Specifying Hypercalls

Atomic steps, e.g.

```
sm.incr g Q :=
  AU << ∃ ∃ v, sm.value g v >> @
Mnova,T
     << sm.value g (v + 1) , COMM Q >>
```

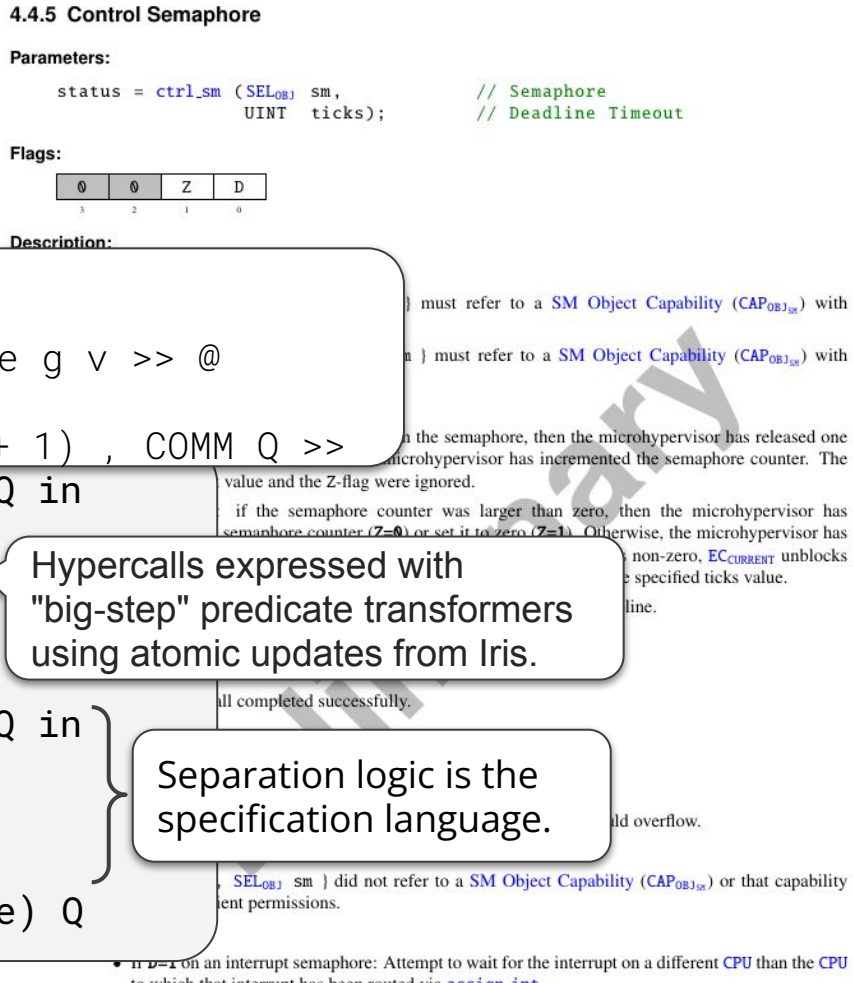
```
  letI* g := resolve_sm ec.pd {UP} sel Q in
  sm.incr g Q
⊢ wp_hypercall ec ctrl_sm(0, sel, _) Q
```

Hypercalls expressed with "big-step" predicate transformers using atomic updates from Iris.

```
  letI* g := resolve_sm ec.pd {DN} sel Q in
  letI* _ := sm_wait g ec in
     sm.acquire z g ec Q
  /\ sm.timeout time g ec Q
⊢ wp_hypercall ec ctrl_sm(0, sel, z, time) Q
```

NOVA choice.

Separation logic is the specification language.



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:

} must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with

} must refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with

the semaphore, then the microhypervisor has released one
microhypervisor has incremented the semaphore counter. The
value and the Z-flag were ignored.

if the semaphore counter was larger than zero, then the microhypervisor has
semaphore counter (Z=0) or set it to zero (Z=1). Otherwise, the microhypervisor has
non-zero, $EC_{CURRENT}$ unblocks
specified ticks value.
line.

ll completed successfully.

ld overflow.

SEL_{OBJ} sm } did not refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) or that capability
ent permissions.

- 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.

Specifying Hypercalls

Atomic steps, e.g.
sm.incr g Q
AU << …
Mnova …

Proved

```
letI* g := resol…
sm.incr g Q
⊢ wp_hypercall e…

letI* g := resolve… …el Q in
letI* _ := sm_wait g ec in
   sm.acquire z g ec Q
/\ sm.timeout time g ec Q
⊢ wp_hypercall ec ctrl_sm(0, sel, z, time) Q
```

…sed with
…dicate transformers
…omic updates from Iris.

Separation logic is the specification language.

NOVA choice.

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

…refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with

…efer to a SM Object Capability ($CAP_{OBJ_{SM}}$) with

…naphore, then the microhypervisor has released one
…ervisor has incremented the semaphore counter. The
…ored.

…nter was larger than zero, then the microhypervisor has
…=0) or set it to zero (Z=1). Otherwise, the microhypervisor has
… non-zero, $EC_{CURRENT}$ unblocks
…e specified ticks value.
…line.

…ll completed successfully.

…ld overflow.

…SEL_OBJ sm } did not refer to a SM Object Capability ($CAP_{OBJ_{SM}}$) or that capability
…ient permissions.

- 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.

The NOVA Event Handler
Simplified

```
Theorem wp_nova_ec_intro : ∀ ec regs,
  (∀ evt regs', [| cpu.step regs evt regs' |] -*
    match evt with
    | None => wp_nova_ec regs'
    | Some syscall => wp_hypercall ec syscall ..
    | Some (mem ..) => wp_mem ..
    | ...
    end)
 ⊢ wp_nova_ec ec regs.
```

User & kernel small step semantics (inspired by SAIL)

Address translation and memory access (2nd stage page tables managed by NOVA, reflected in separation logic state)

Starting NOVA (Simplified)
Input

Parametric over *any* startup program.

```
∀ startup_image,
 (∀ root_pd root_ec root_sc,
    pd.mem root_pd startup_image -*
    initial_pd state root_pd root_ec root_sc -*
    memory -* … -*
    wp_nova_ec root_ec (startup_regs …))
⊢ NOVA_loaded -*
   elf startup_image -*
   memory -*
   … -*
   wp_arm_el2 boot_regs
```

Raw machine resources.
Given to NOVA by the bootloader.

Behavior of the boot CPU expressed as a weakest-precondition.

The "program" is the register state.

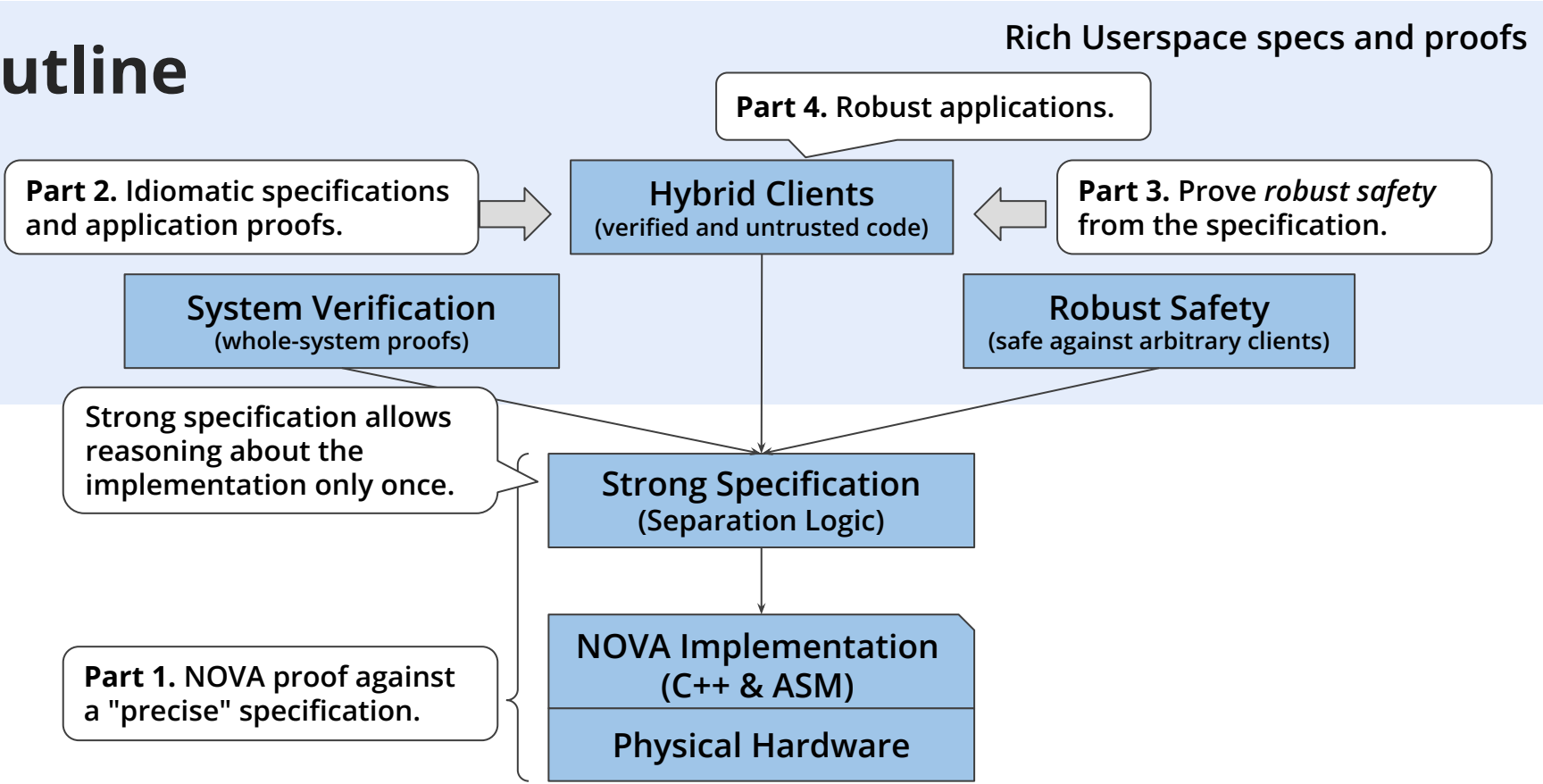
Starting NOVA (Simplified)
Userspace

```
∀ startup_image,
 (∀ root_pd root_ec root_sc,
    pd.mem root_pd startup_image -*
    initial_pd state root_pd root_ec root_sc -*
    memory -* … -*
    wp_nova_ec root_ec (startup_regs …))
⊢ NOVA_loaded -*
   elf startup_image -*
   memory -*
   … -*
   wp_arm_el2 boot_regs
```

NOVA resources.
Given to userspace by NOVA.

Weakest precondition for the root EC.

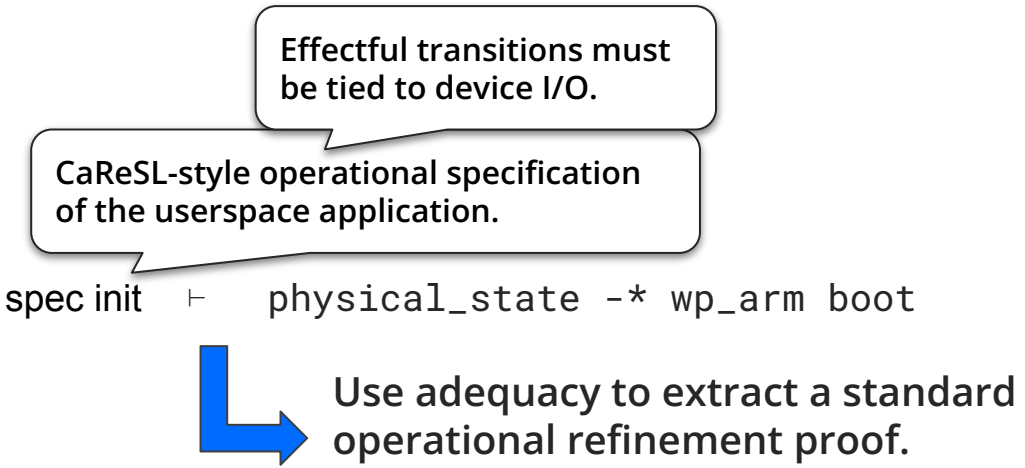
Outline


Rich Userspace specs and proofs
Part 4. Robust applications.
Part 2. Idiomatic specifications and application proofs.
Hybrid Clients
(verified and untrusted code)
Part 3. Prove robust safety from the specification.
System Verification
(whole-system proofs)
Robust Safety
(safe against arbitrary clients)
Strong specification allows reasoning about the implementation only once.
Strong Specification
(Separation Logic)
NOVA Implementation
(C++ & ASM)
Part 1. NOVA proof against a "precise" specification.
Physical Hardware

System Refinement

Establish properties for the applications that run on top of NOVA.

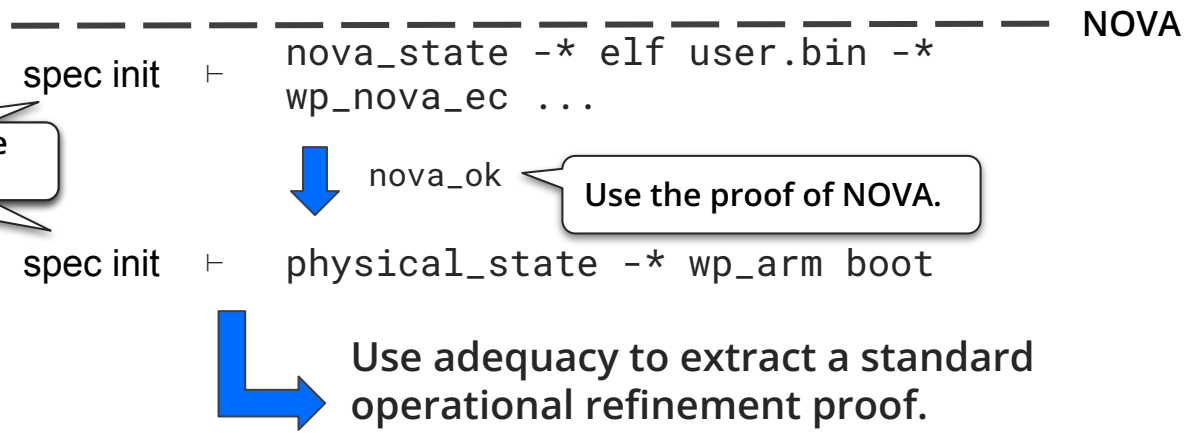
- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independent of SL) using Iris adequacy.



System Refinement

Establish properties for the applications that run on top of NOVA.

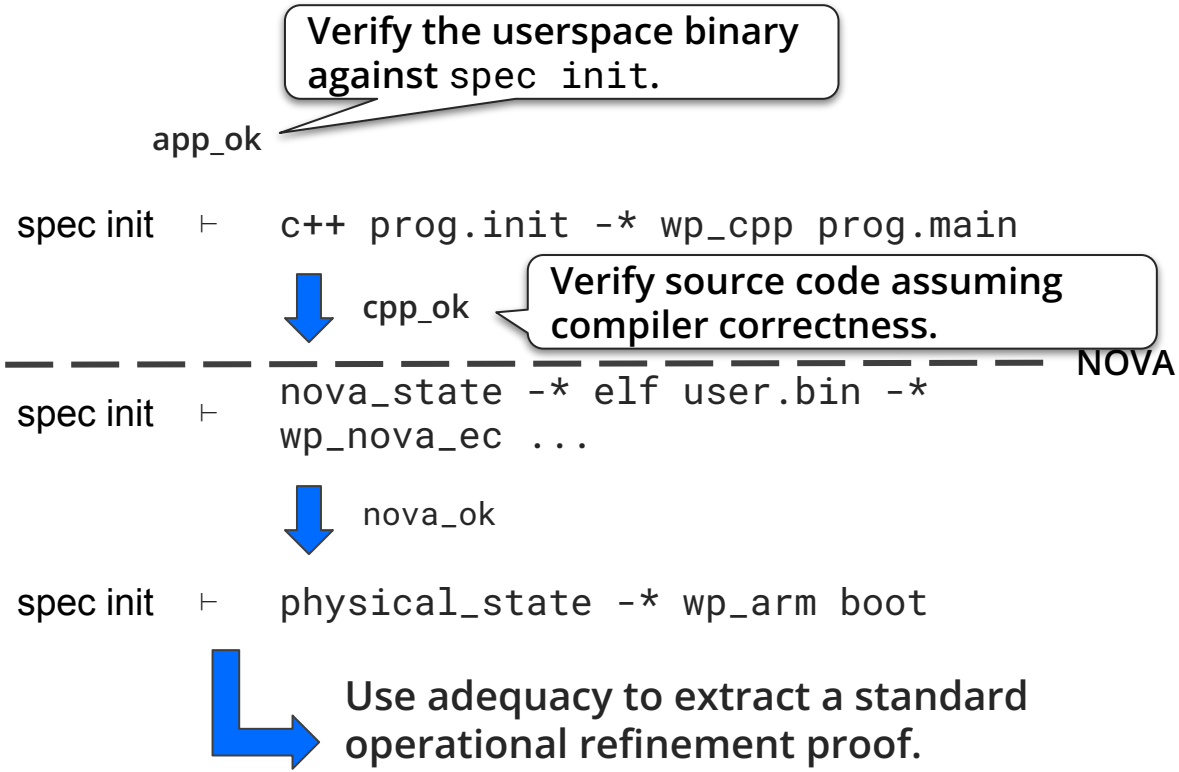
- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independe… adequacy.



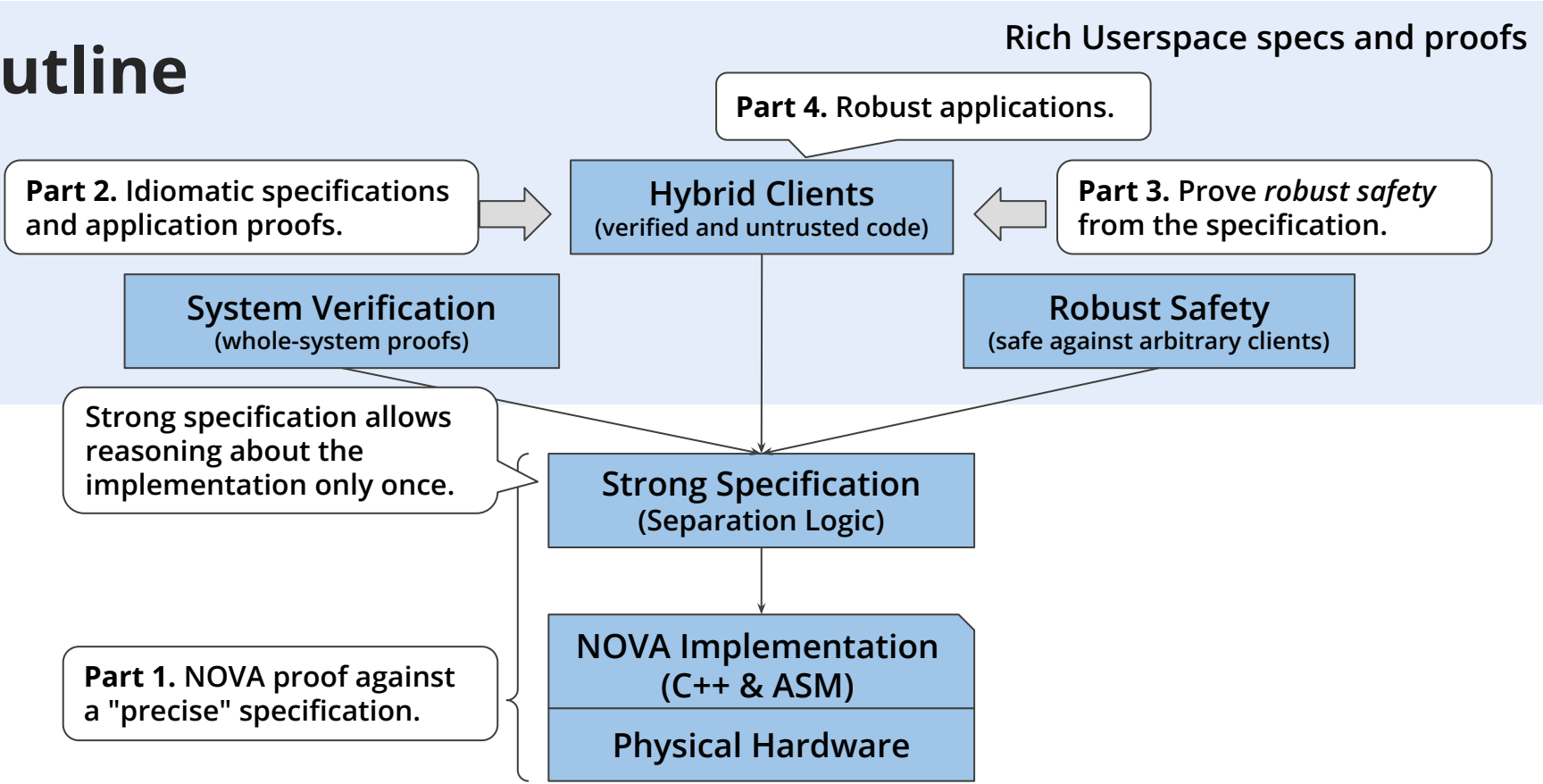
System Refinement

Establish properties for the applications that run on top of NOVA.

- Use CaReSL-style techniques to prove refinement using ghost state and invariants.
- Extract the end-to-end proof (independent of SL) using Iris adequacy.



Outline


Rich Userspace specs and proofs
Part 4. Robust applications.
Part 2. Idiomatic specifications and application proofs.
Hybrid Clients
(verified and untrusted code)
Part 3. Prove robust safety from the specification.
System Verification
(whole-system proofs)
Robust Safety
(safe against arbitrary clients)
Strong specification allows reasoning about the implementation only once.
Strong Specification
(Separation Logic)
NOVA Implementation
(C++ & ASM)
Part 1. NOVA proof against a "precise" specification.
Physical Hardware

Proving Robust Safety

Want to show that NOVA is safe for **any** userspace code.

Need to show that this assumption is trivial.

```
∀ startup_image,
 (∀ root_pd root_ec root_sc,
    pd.mem root_pd startup_image -*
    initial_pd state root_pd root_ec root_sc -*
    memory -* … -*
    wp_nova_ec root_ec (startup_regs …))
⊢ NOVA_loaded -*
    elf startup_image -*
    memory -*
    … -*
    wp_arm_el2 boot_regs
```

Proving Robust Safety from the Spec

Express the high-low state distinction within separation logic.

- Invariants allow flexible, concurrent sharing.
- Existential quantifiers express low-integrity

rs-inv

∃ objs, let valid o := o \in objs in
[*list] o \in objs,

∃ v es, sm.value ↦ v
** sm.queue ↦ ecs ** [| Forall **valid** es |]

"Low integrity" invariant for semaphores.
Ownership exists, values are minimally constrained.

Properties for other kernel object types.

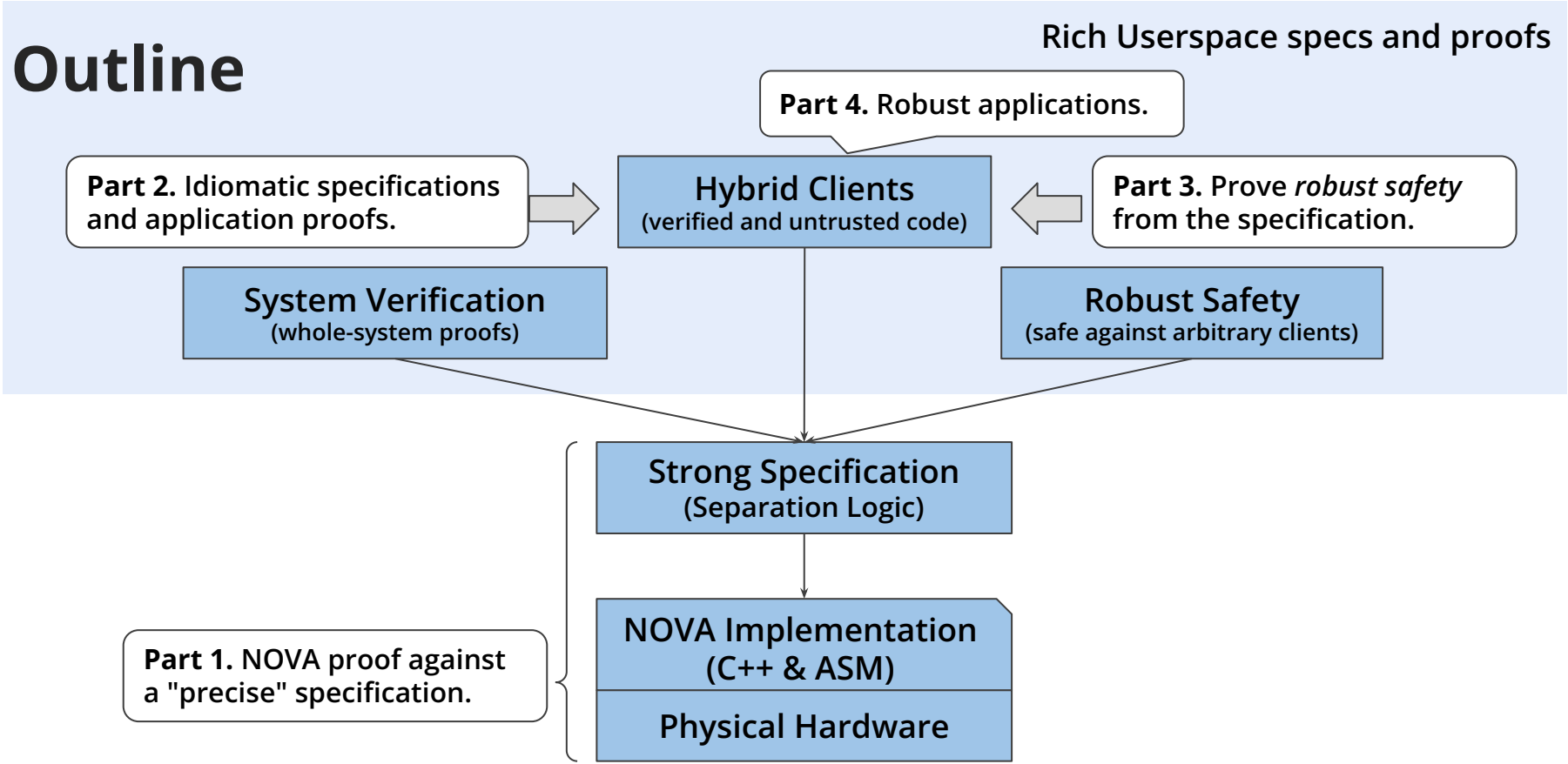
** all schedulable ECs are **valid**
** all interrupts bound to **valid** interrupt SMs

Over arbitrary user binaries.

Need to Show

```
Userspace Resources |-- |={T}=> inv rs-inv
inv rs-inv |-- wp_nova_ec boot_ec boot_regs
```

Outline


Rich Userspace specs and proofs
Part 4. Robust applications.
Part 2. Idiomatic specifications and application proofs.
Hybrid Clients
(verified and untrusted code)
Part 3. Prove robust safety from the specification.
System Verification
(whole-system proofs)
Robust Safety
(safe against arbitrary clients)
Strong Specification
(Separation Logic)
NOVA Implementation
(C++ & ASM)
Physical Hardware
Part 1. NOVA proof against a "precise" specification.

Userspace Robust Safety

State may change integrity levels throughout the execution of the program.

- High integrity state shared with untrusted code.
- Low integrity state revoked from untrusted code.

Revocation is generally difficult and requires tight reasoning about confinement and visibility.

Precise specifications support endorsement (low -> high) because they decouple state from policy.

Provide mechanisms (assertions), not policies (invariants, quantifiers).

Support a "data life cycle".

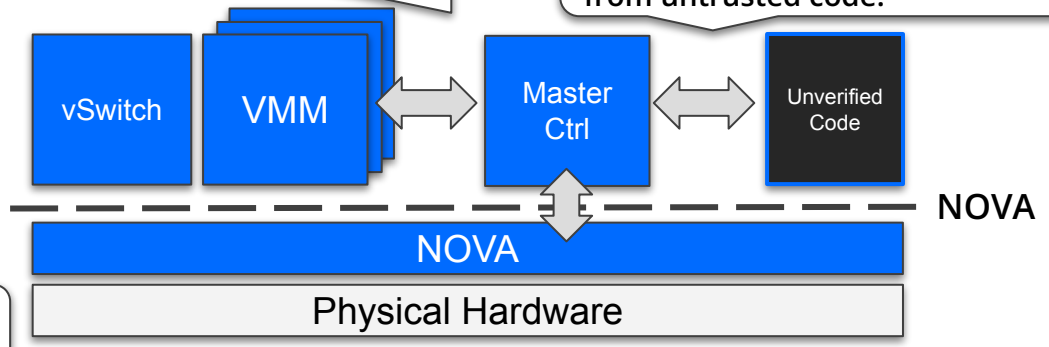
1. Create state. (high)
2. Configure state. (high)
3. Share permissions with untrusted code? (low)
4. Revoke state. (high)
5. Destroy state. (high)

Entire data lifecycle is provable using the strong specification.

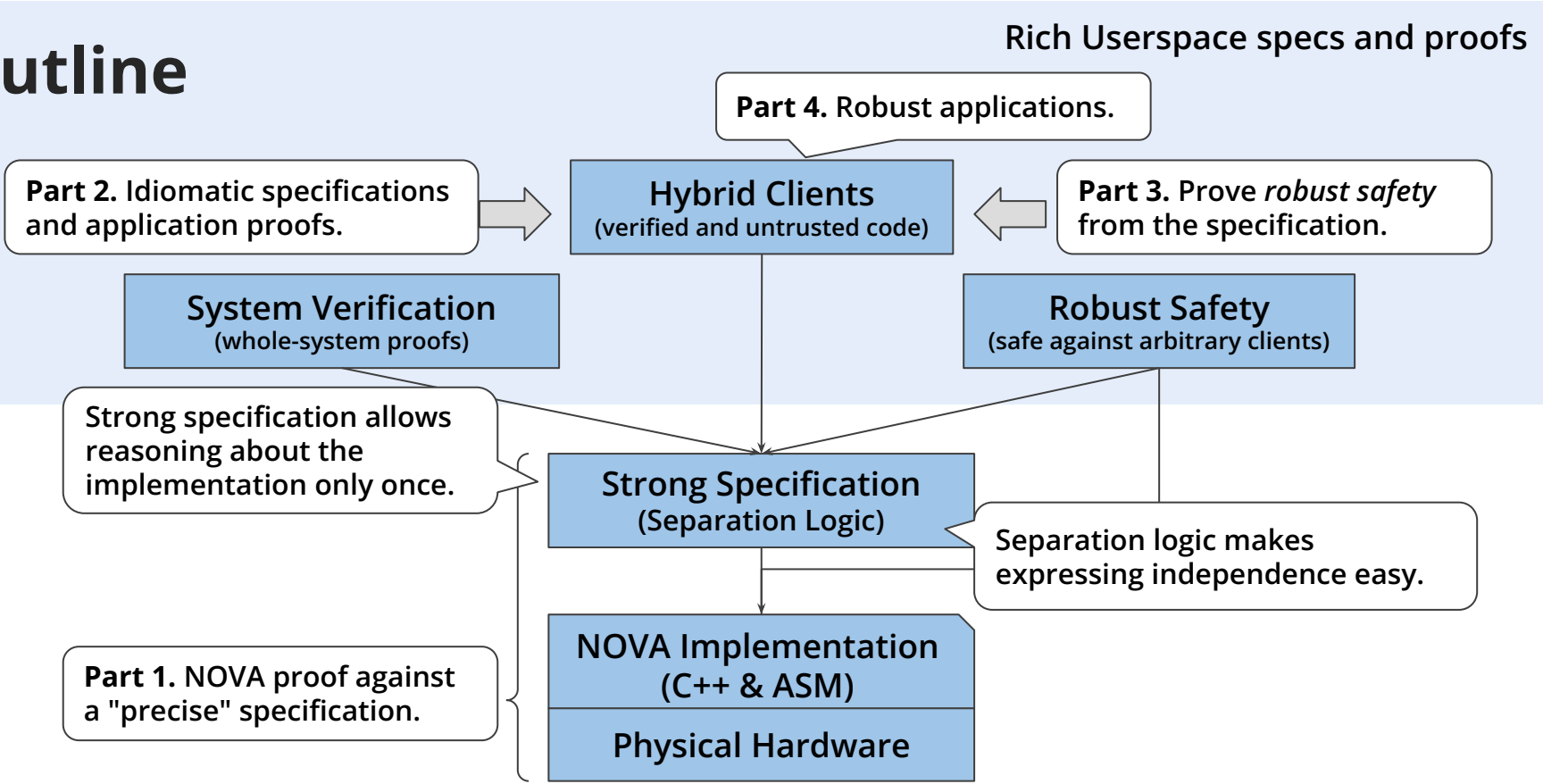
Can share limited permissions, e.g. only up a semaphore, only call a portal, only read a page, etc.

Share resources with verified code. Requires strong specification.

Create resources, pass to untrusted code, revoke resources from untrusted code.



Outline


Rich Userspace specs and proofs
Part 4. Robust applications.
Part 2. Idiomatic specifications and application proofs.
Hybrid Clients
(verified and untrusted code)
Part 3. Prove robust safety from the specification.
System Verification
(whole-system proofs)
Robust Safety
(safe against arbitrary clients)
Strong specification allows reasoning about the implementation only once.
Strong Specification
(Separation Logic)
Separation logic makes expressing independence easy.
Part 1. NOVA proof against a "precise" specification.
NOVA Implementation
(C++ & ASM)
Physical Hardware