TOWARD COMPOSITIONAL VERIFICATION OF INTERRUPTIBLE OS KERNELS AND DEVICE DRIVERS

Xiongnan (Newman) Wu
Joint work with Hao Chen, Zhong Shao, Joshua Lockerman, and Ronghui Gu

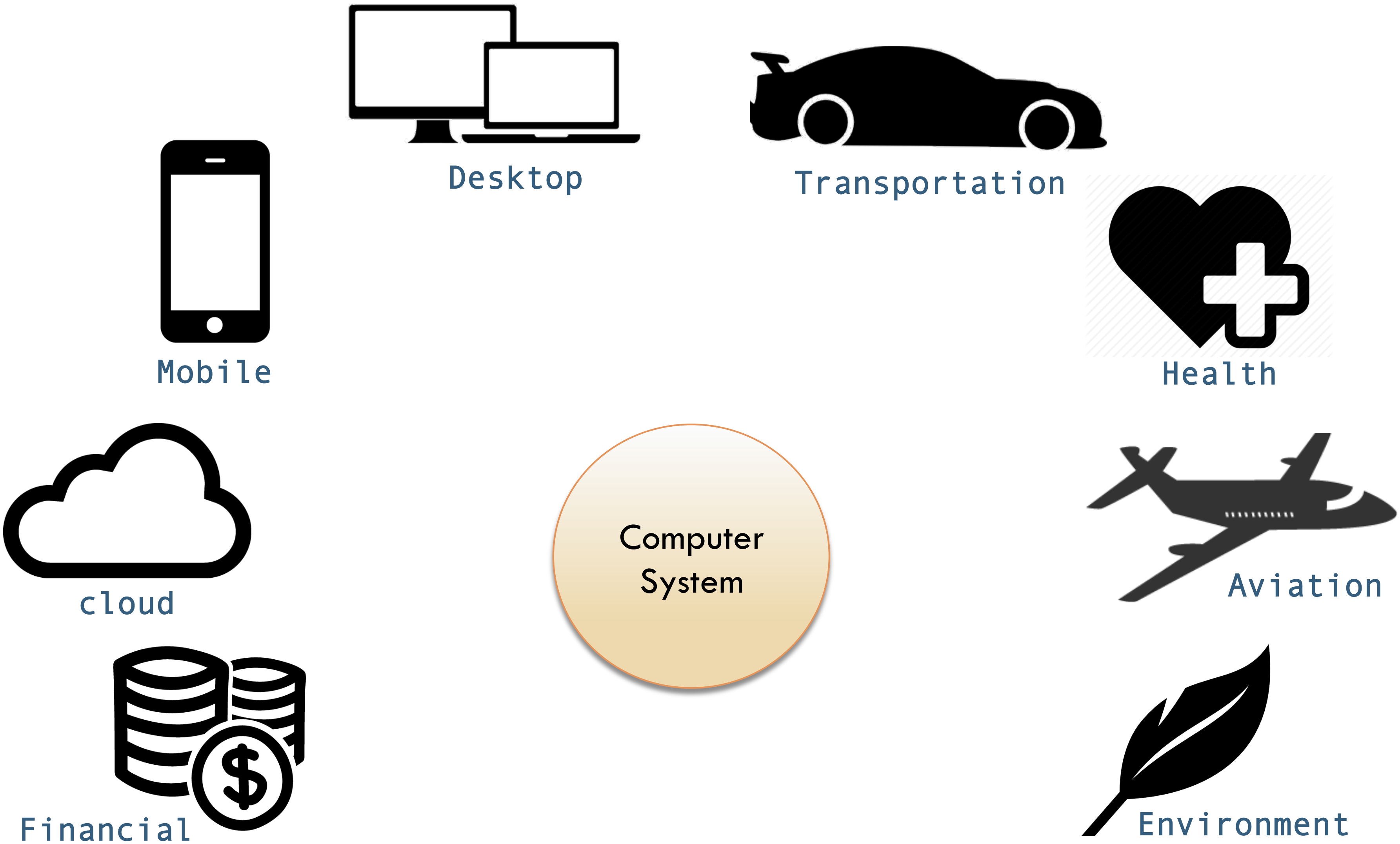
Yale University

CertiKOS

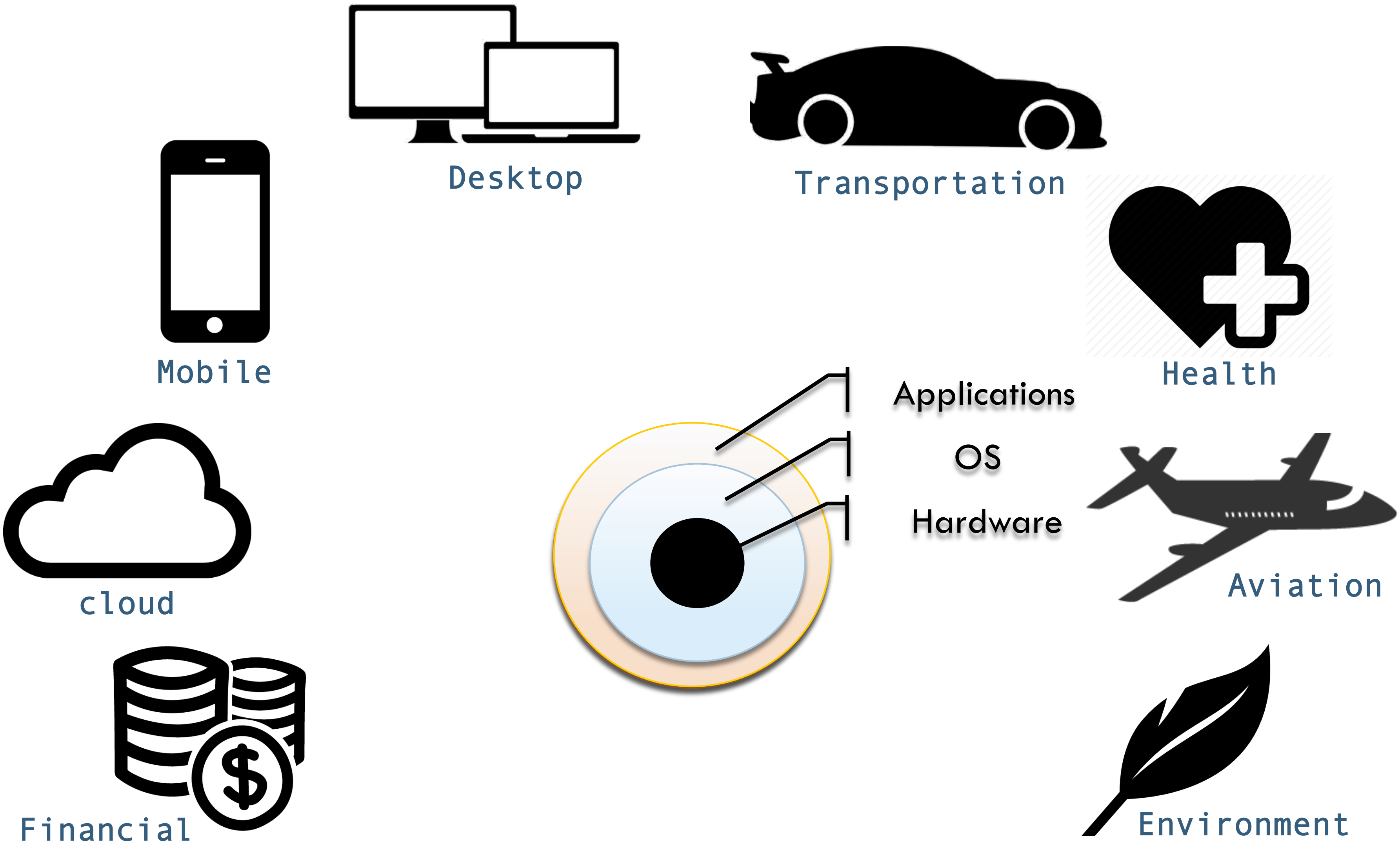
PLDI 2016

June 16, 2016

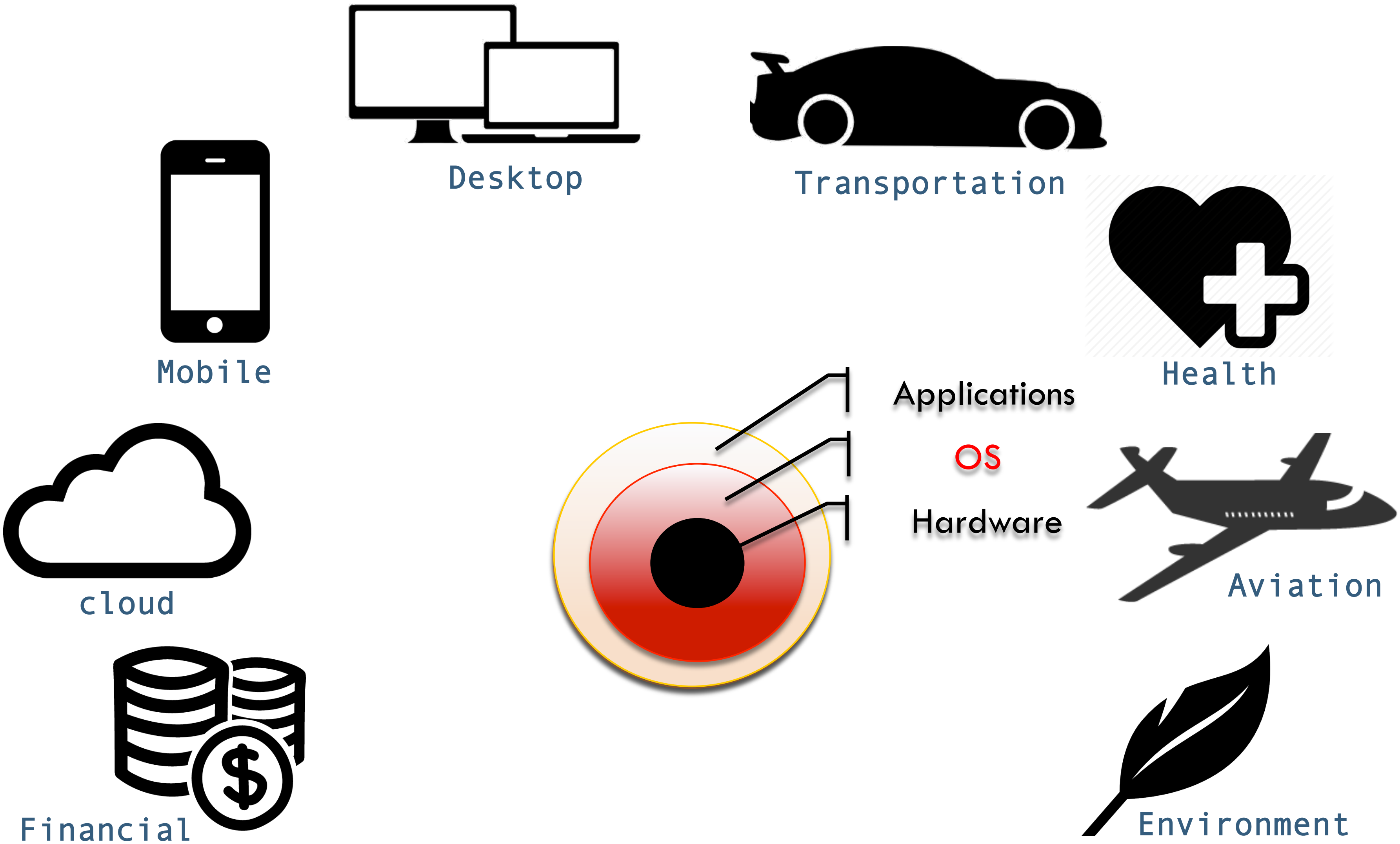
Do we really need high-assurance OS?



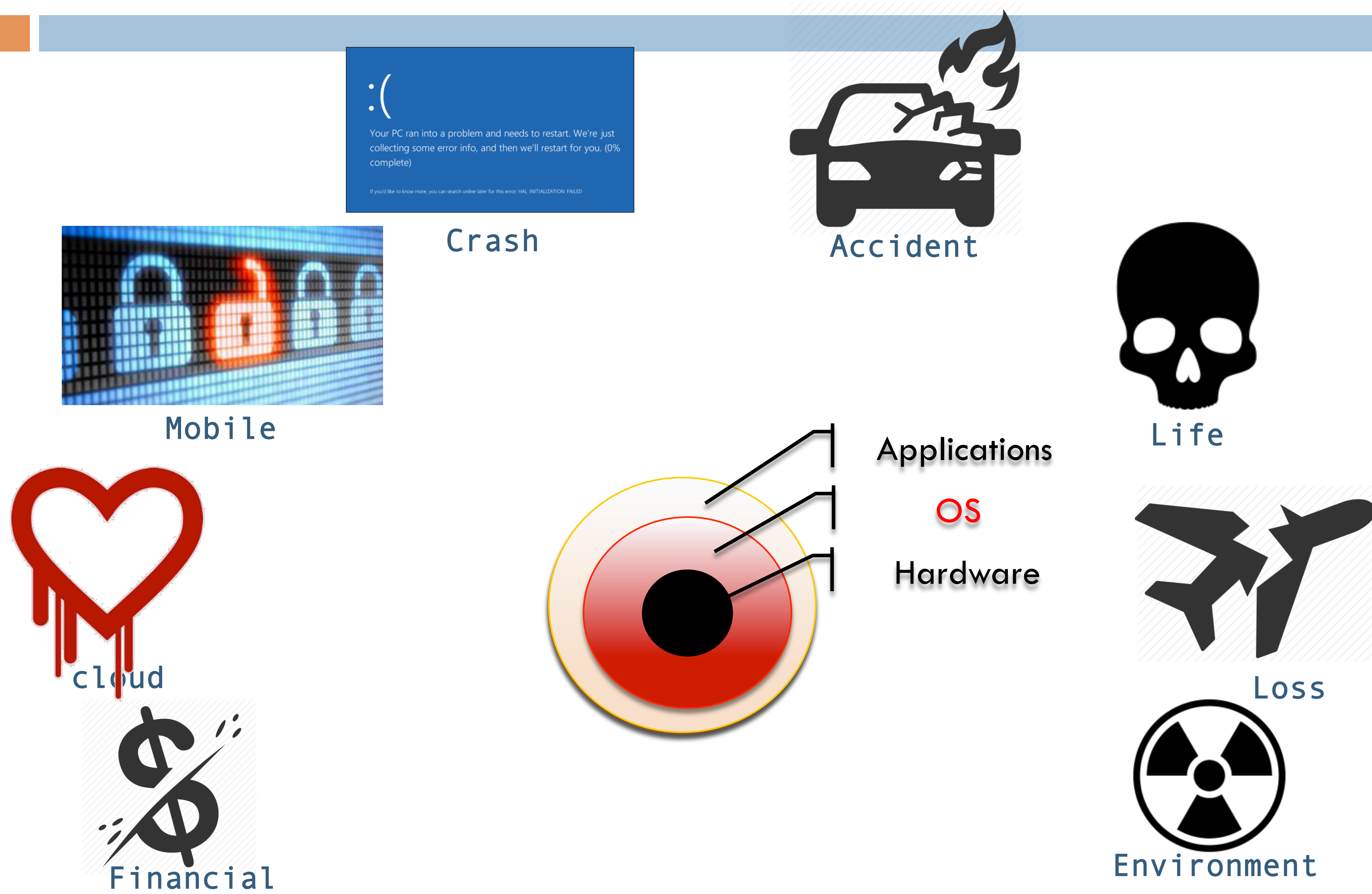
Do we really need high-assurance OS?



Do we really need high-assurance OS?

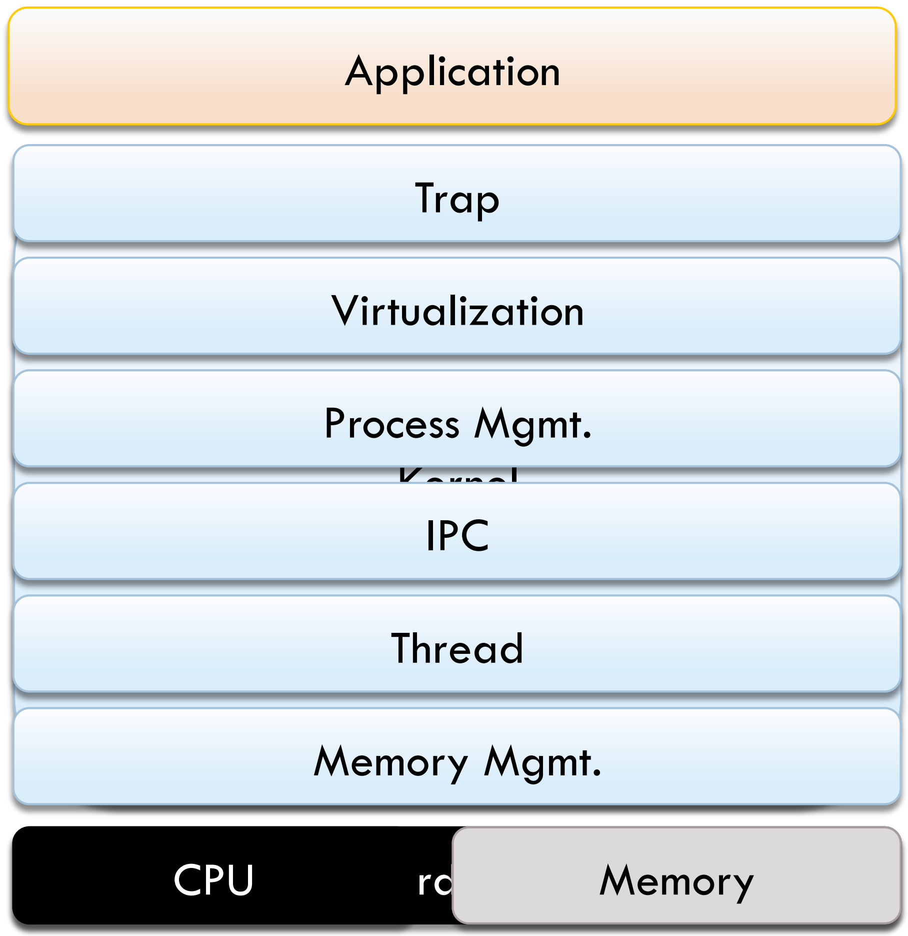


Do we really need high-assurance OS?

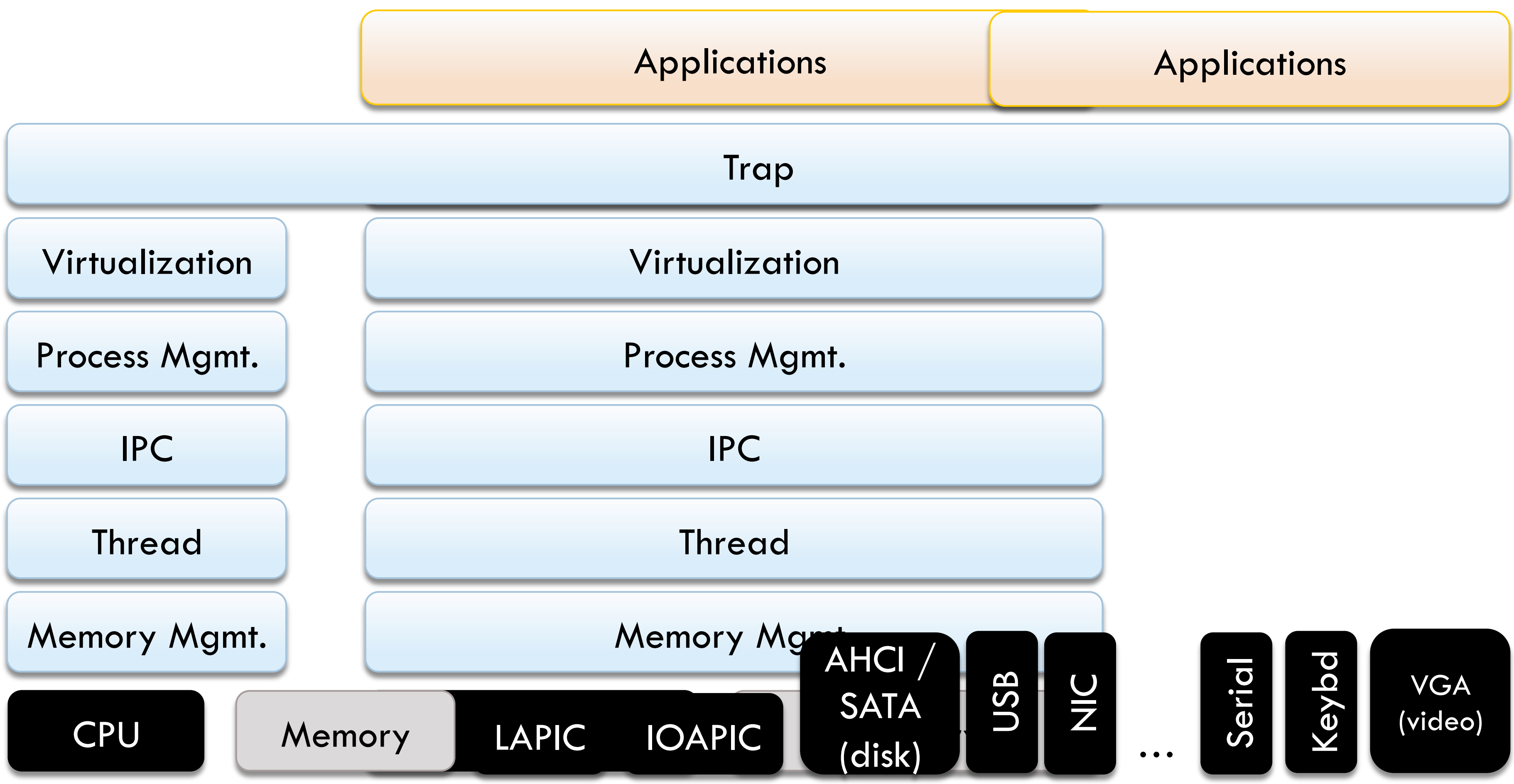


Formal Verification of OS Kernel

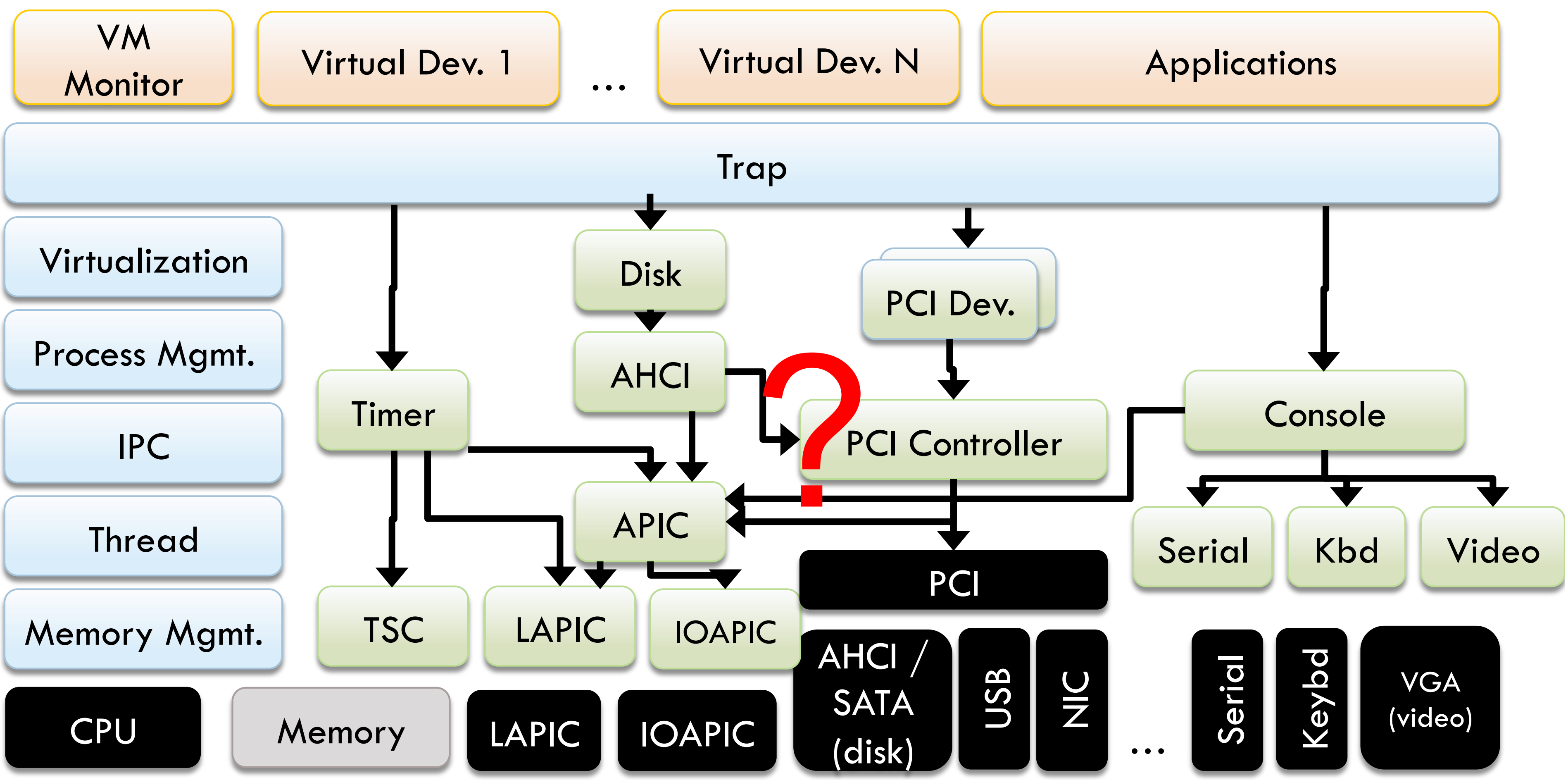
- seL4
- CertiKOS
- Verve



Formal Verification of OS Kernel



Formal Verification of OS Kernel

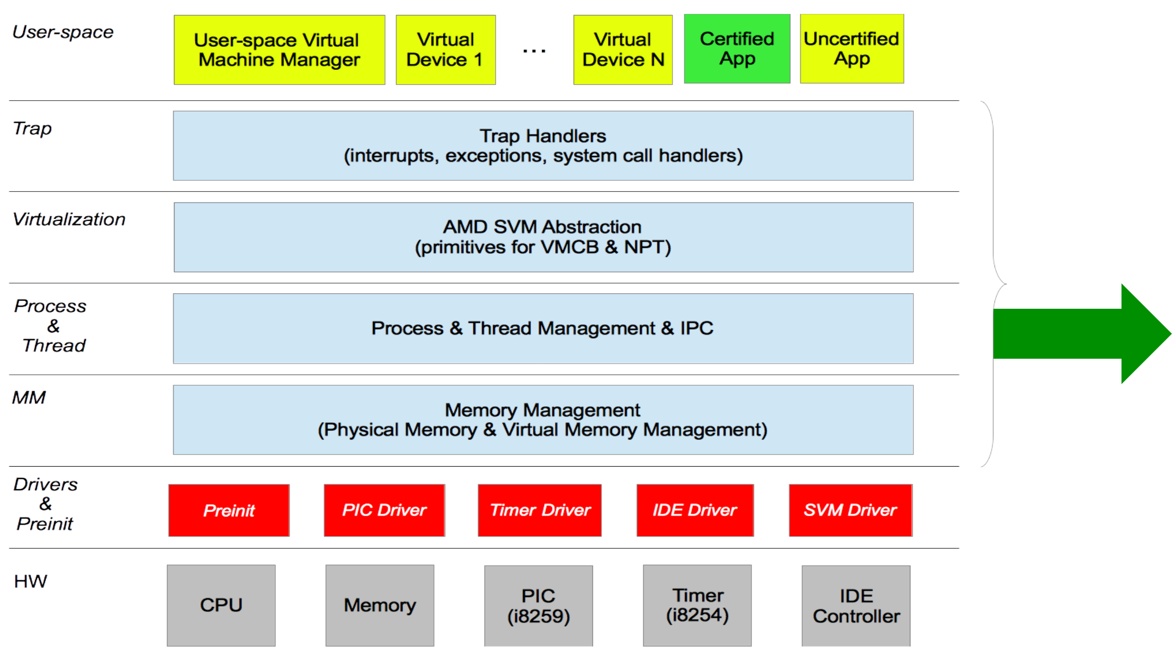


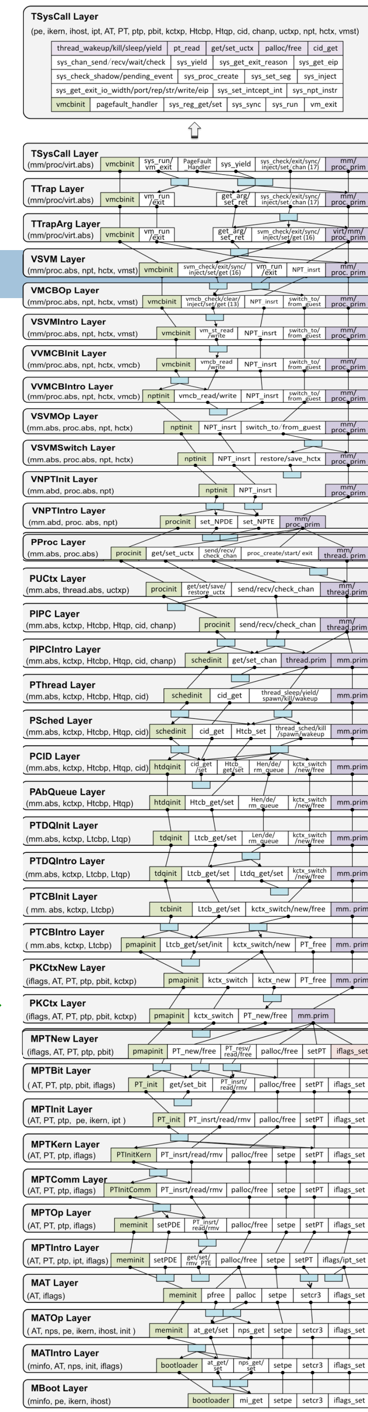
Device Drivers in Mainstream OS

- 70% of Linux 2.4.1 kernel are device drivers.
- 70% of Windows crash are caused by third-party driver code.

mCertiKOS Overview [POPL'15]

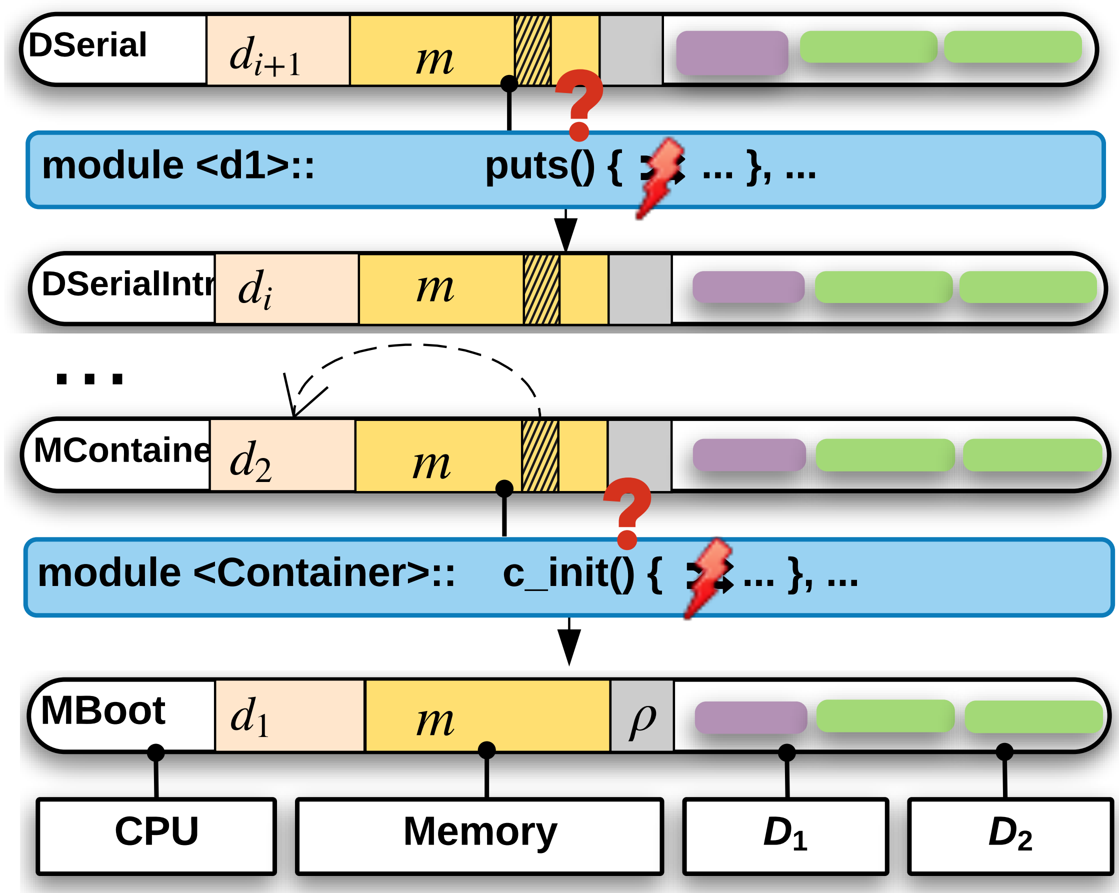
- Single-core version of CertiKOS.
- 3k LOC, can boot Linux as guest.
- Aggressive use of abstraction over deep specification (37 layers).





Main Challenge

Every fine-grained processor step could be interrupted.

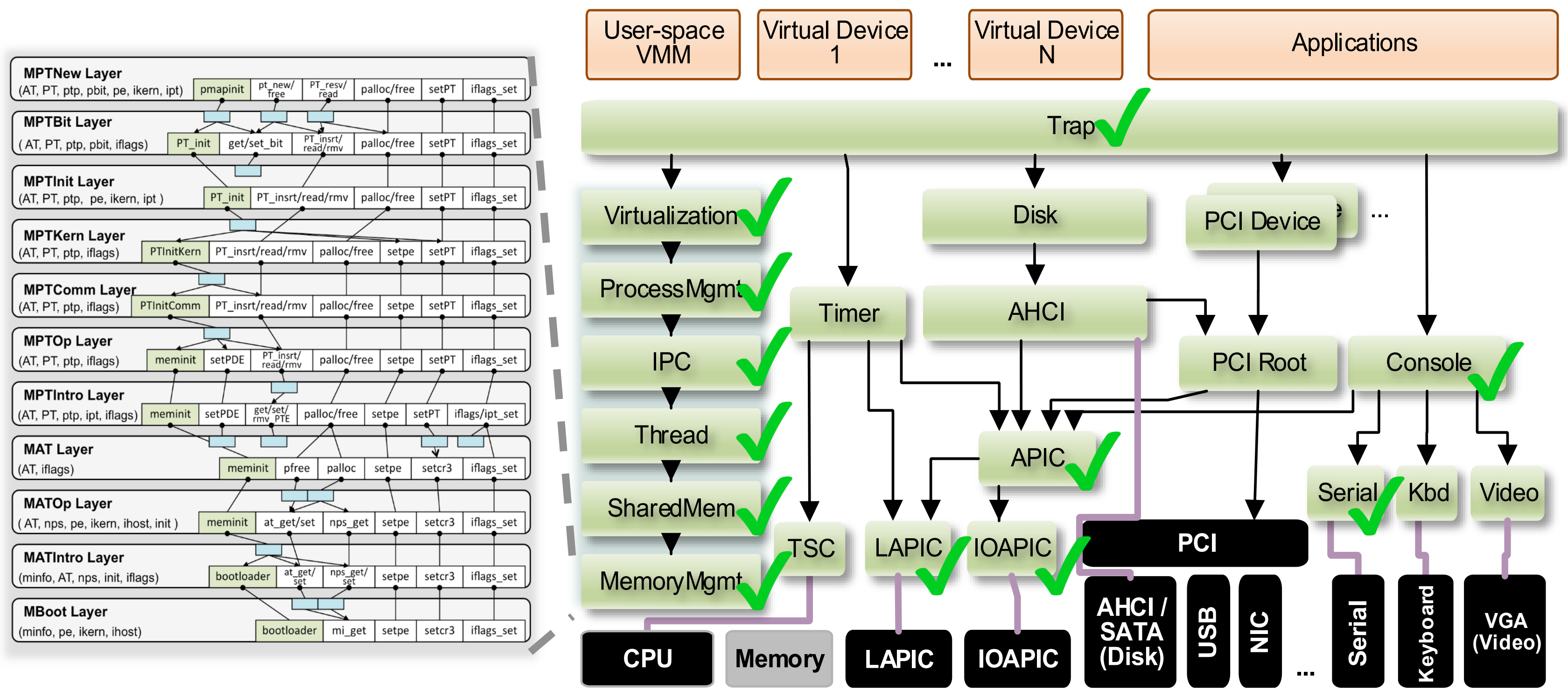


Other Challenges

- Interrupt hardware can be *dynamically* configured.
- Devices and CPU run in *parallel*.
- Device drivers are written in both C and *assembly*.
- The correctness results of different components should be *linked formally*.

Our Contributions

The *first* formally verified *interruptible* OS kernel with *device drivers*.

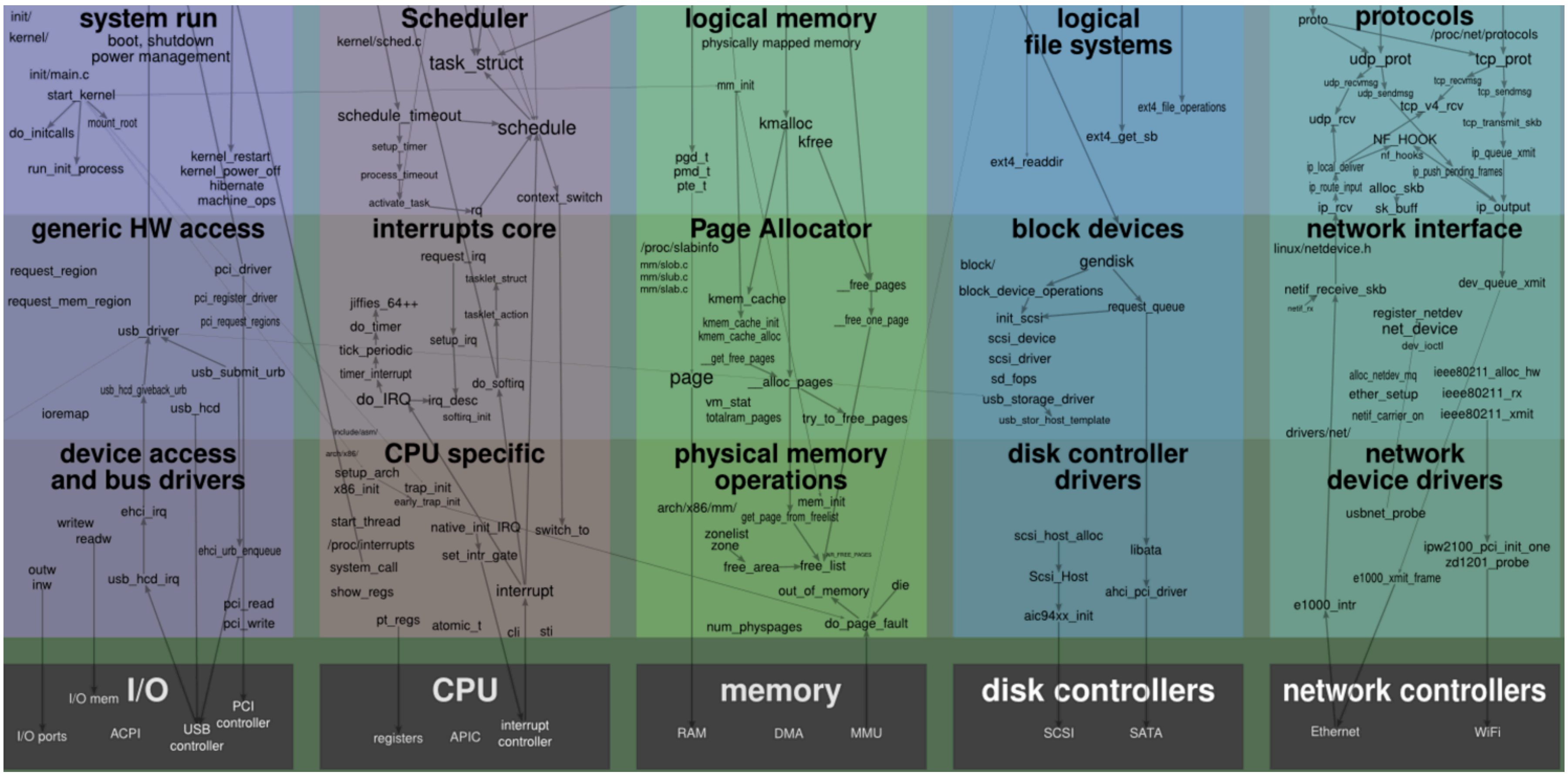


Our Contributions

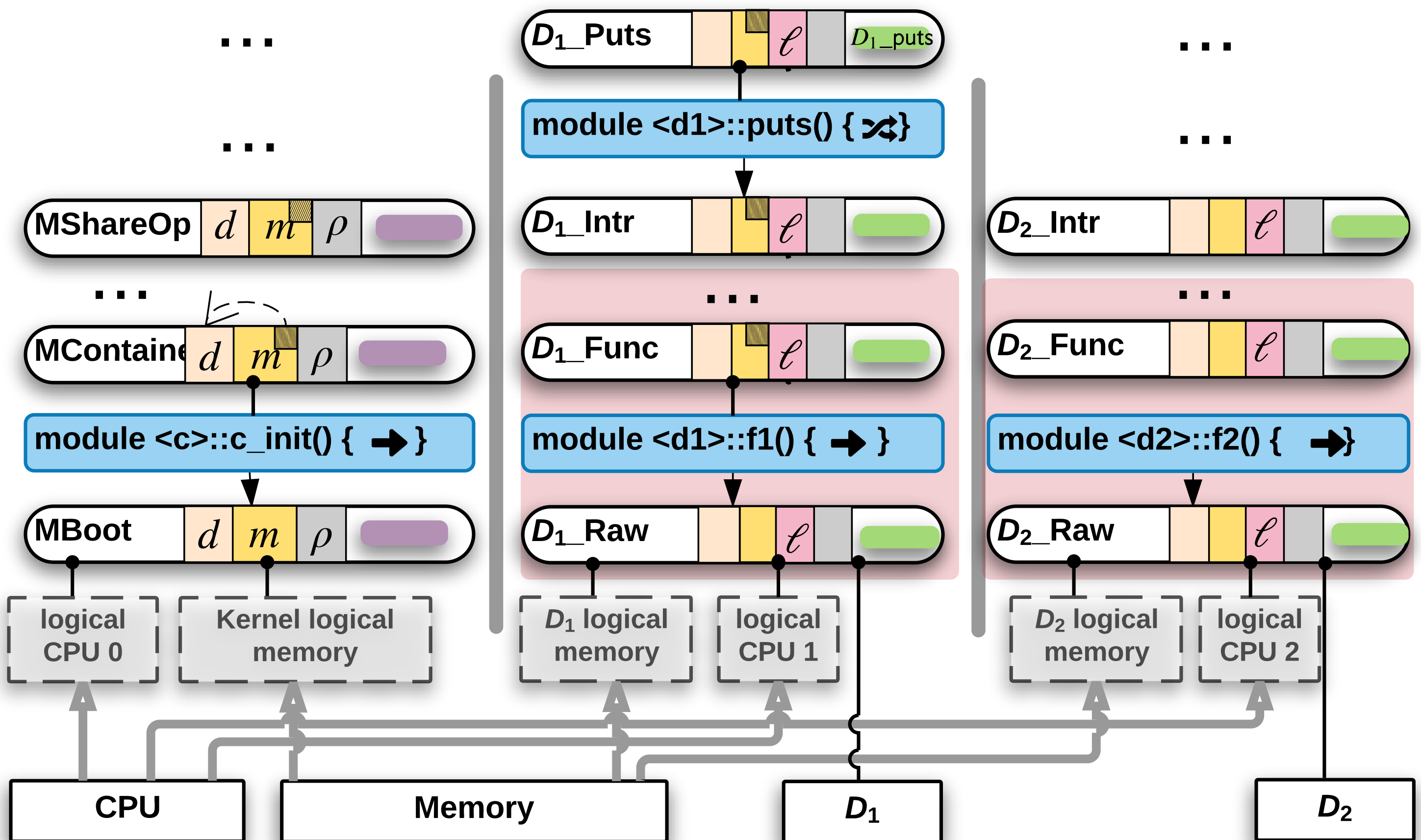
- New techniques for certifying abstraction layers with multiple ***logical CPUs*** and devices.
- New techniques for building formal ***certified device hierarchies***.
- An abstraction-layer-based approach for reasoning about ***interrupts***.
- ***Case study***: interruptible mCertiKOS with device drivers.

Linux Kernel Map

Kernel components are sorted into different stacks of abstraction layers based on their underlying hardware device.



New Machine Model



Our Contributions

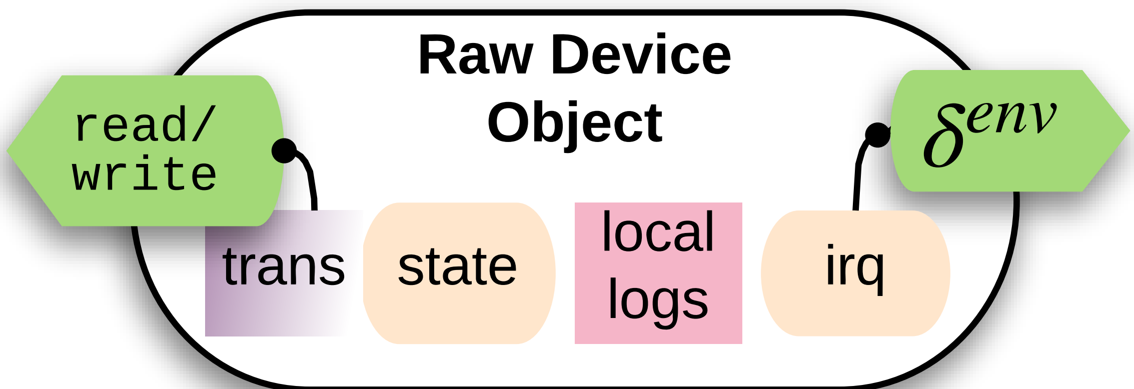
- New techniques for certifying abstraction layers with multiple ***logical CPUs*** and devices.
- New techniques for building formal ***certified device hierarchies***.
- An abstraction-layer-based approach for reasoning about ***interrupts***.
- ***Case study***: interruptible mCertiKOS with device drivers.

Hardware Device Model

- Devices are modeled as transition systems parameterized by all possible lists of external events.
- Example external events:
  - Recv (s: list char)
  - KeyPressed (c: Z)
- State: observable registers.
- Transition:
  - environmental transition: δ^{env}
  - I/O transition: δ^{CPU}

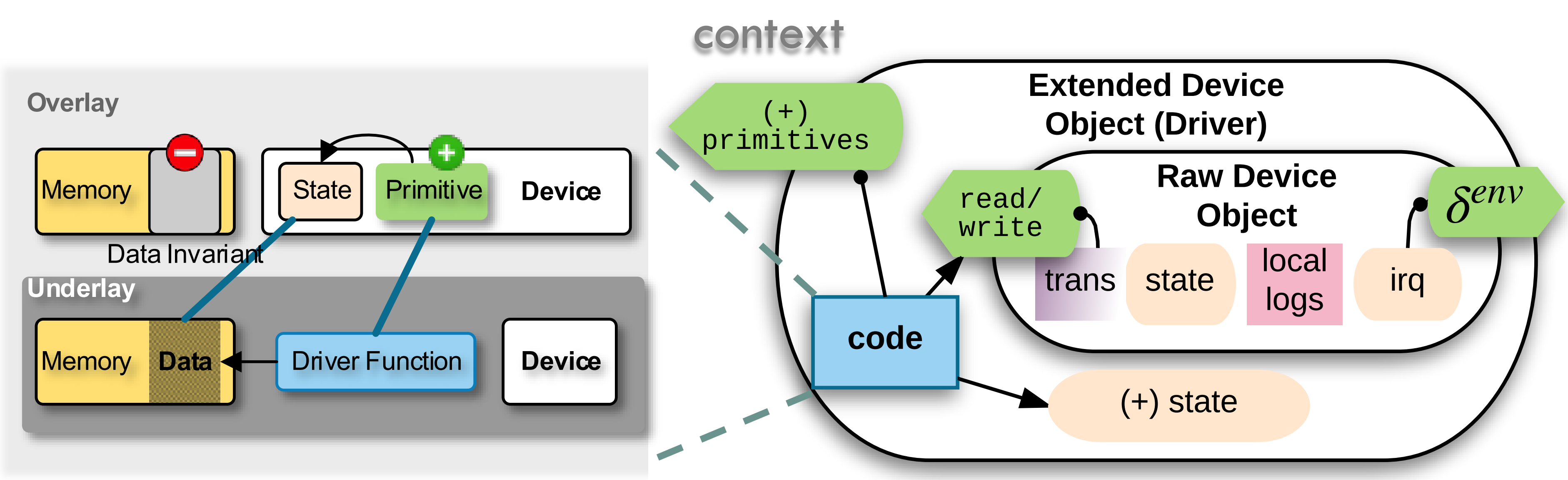
Raw Device Object

- Local log for the list of observed external events.
- Multiple local logs to handle disjoint set of external events asynchronously.
- Read/Write instructions: IN/OUT, memory mapped I/O, *etc.*

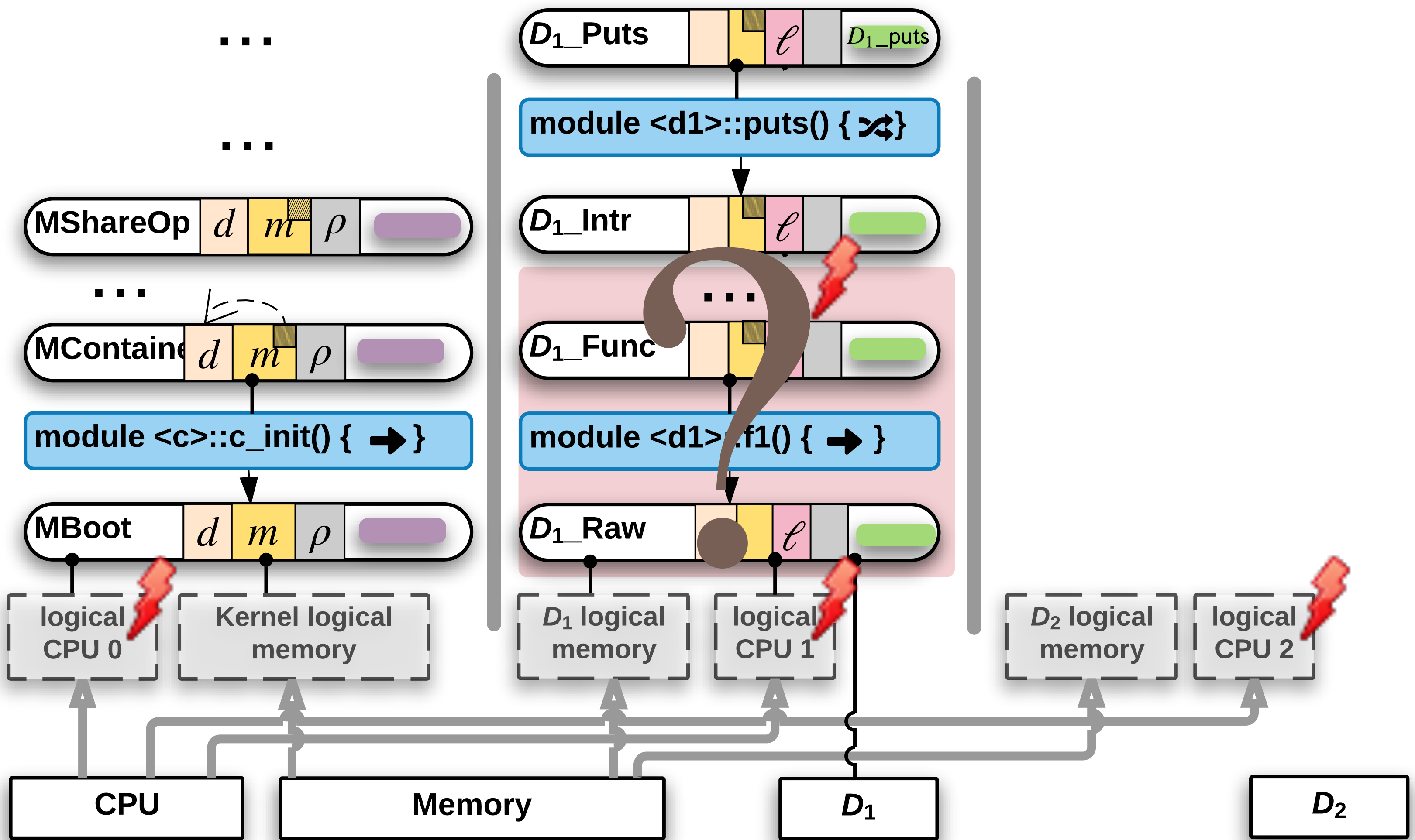


Extended Device Object

Driver as a logical device.



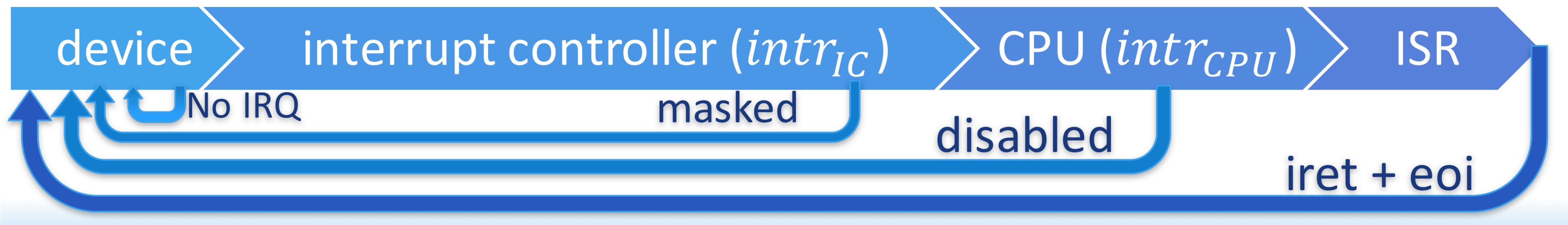
Recap: Machine Model

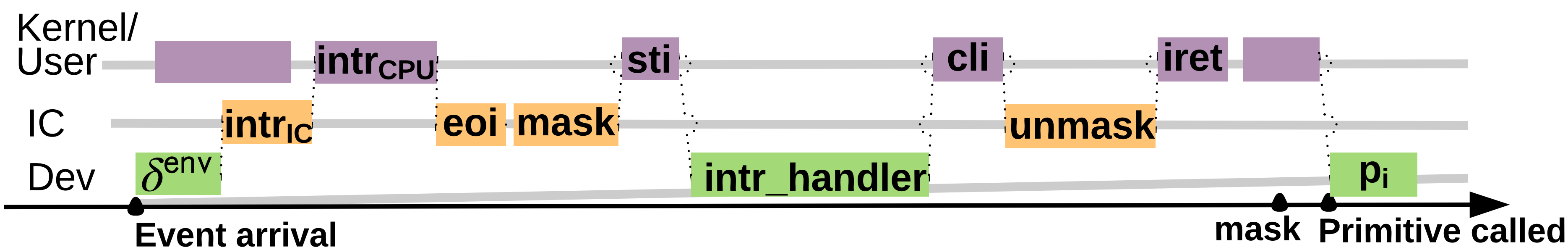


Our Contributions

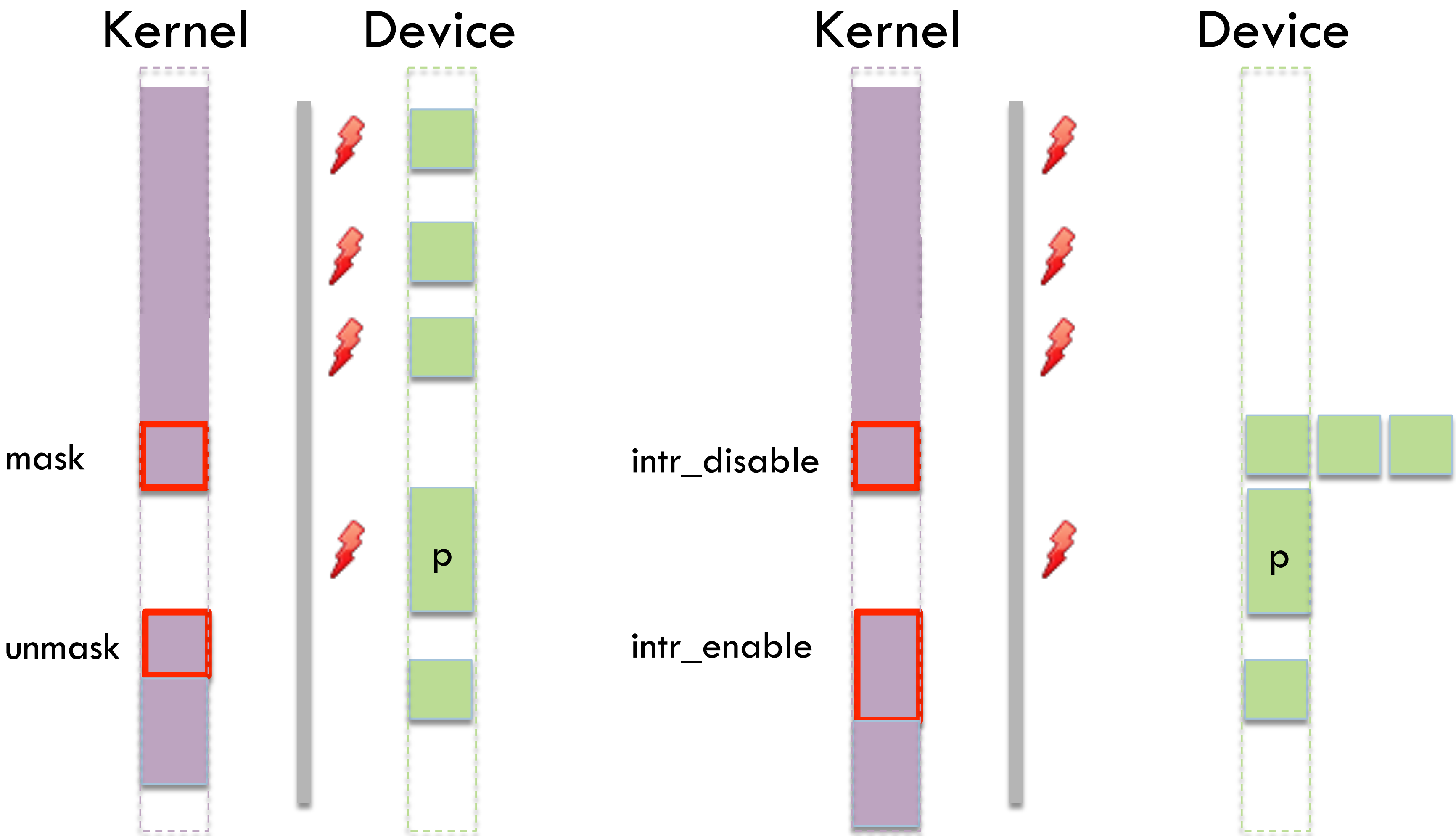
- New techniques for certifying abstraction layers with multiple ***logical CPUs*** and devices.
- New techniques for building formal ***certified device hierarchies***.
- An abstraction-layer-based approach for reasoning about ***interrupts***.
- ***Case study***: interruptible mCertiKOS with device drivers.

Interrupt Models



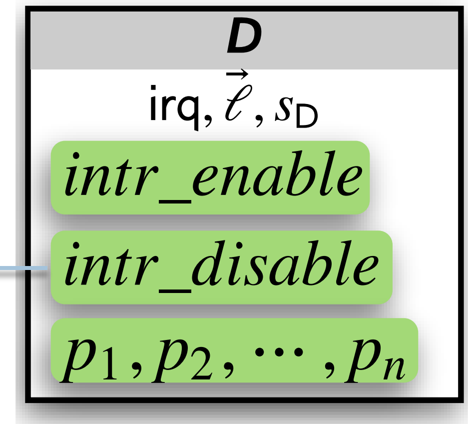


New HW Interrupt Model



Semantics of *intr_disable*

- Scans external events.
- Recursively performs the environmental transition.
- Synchronizes unhandled interrupts.



details in the paper

DISABLENOINTR: Disable with no unhandled interrupt

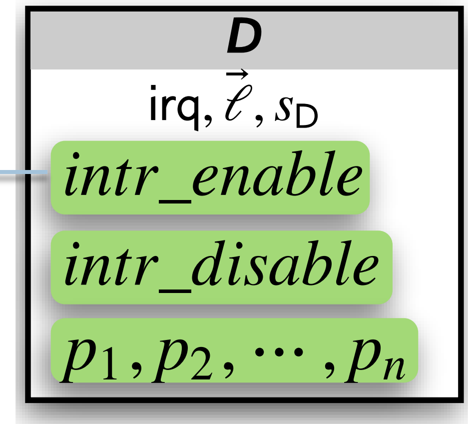
$$\frac{(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) \qquad s_{\mathsf{tmp}} = \delta^{\mathsf{env}}(s, e) \qquad s_{\mathsf{tmp}}.irq = \mathsf{false} \qquad s' = s[\mathsf{iFlag} \leftarrow 0]}{\mathtt{intr\_disable}(s, \ell_i, \ell^{env}) = (s', \ell_i)}$$

DISABLEINTR: Disable with unhandled interrupts

$$\frac{(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) \qquad s' = \delta^{\mathsf{env}}(s, e) \qquad s'.irq = \mathsf{true} \qquad (s'', \ell_i'') = \mathtt{intr\_handler}(s', \ell_i', \ell^{env}) \qquad (s''', \ell_i''') = \mathtt{intr\_disable}(s'', \ell_i'', \ell^{env})}{\mathtt{intr\_disable}(s, \ell_i, \ell^{env}) = (s''', \ell_i''')}$$

Semantics of *intr_enable*

- Recursively discharges pending interrupts.
- Delayed interrupts that occur while the interrupt is disabled.



details in the paper

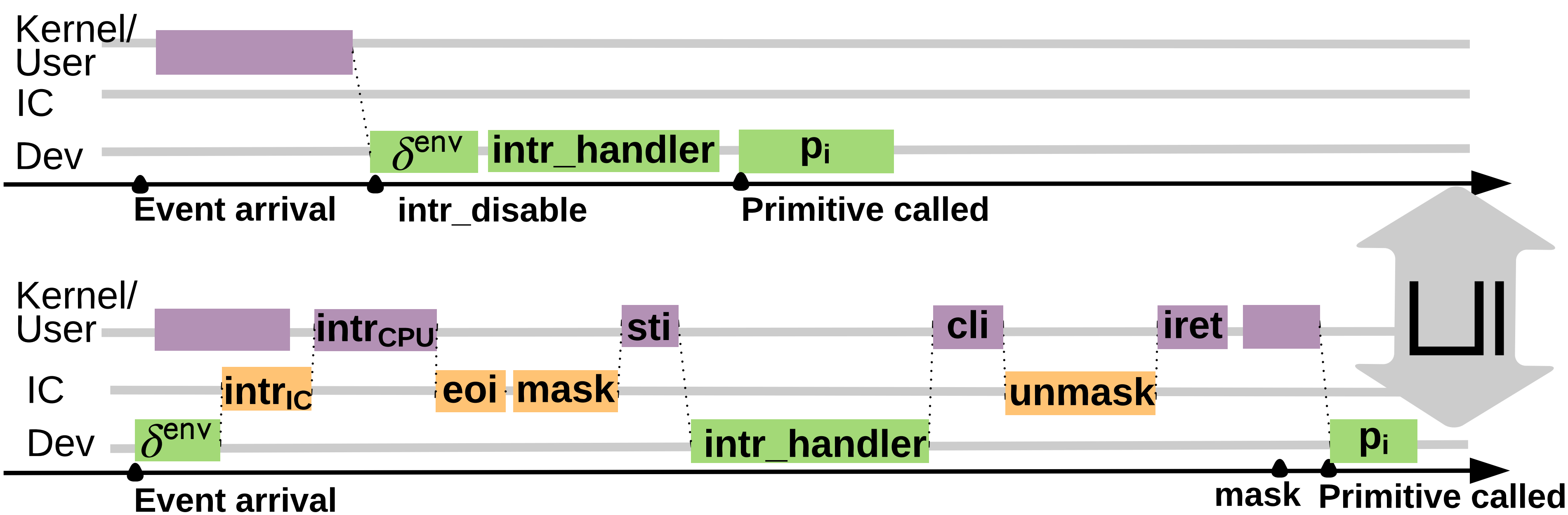
ENABLENOINTR: Enable with no pending interrupt

$$\frac{s.irq = \text{false} \qquad s' = s[\text{iFlag} \leftarrow 1]}{\texttt{intr\_enable}(s, \ell_i, \ell^{env}) = (s', \ell_i)}$$

ENABLEINTR: Enable with pending interrupts

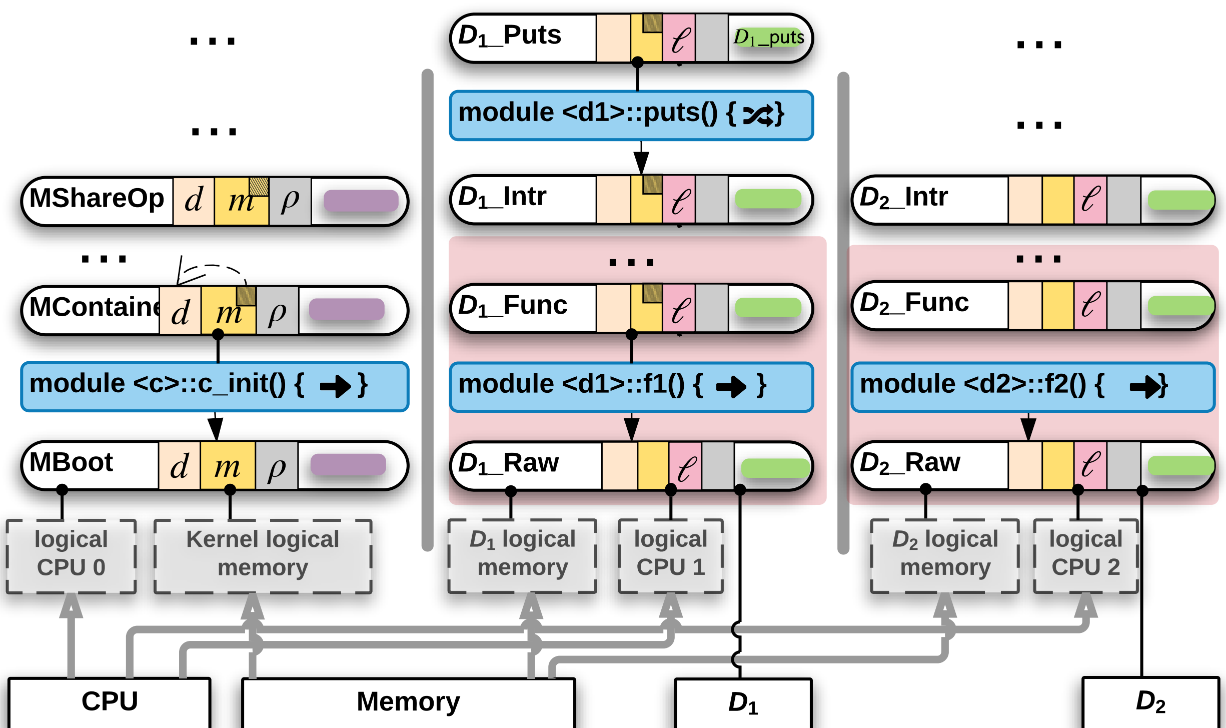
$$\frac{s.irq = \text{true} \qquad (s', \ell_i') = \texttt{intr\_handler}(s, \ell_i, \ell^{env}) \qquad (s'', \ell_i'') = \texttt{intr\_enable}(s', \ell_i', \ell^{env})}{\texttt{intr\_enable}(s, \ell_i, \ell^{env}) = (s'', \ell_i'')}$$

Refinement btw. The HW & Abstract Interrupt Model



Our Approach

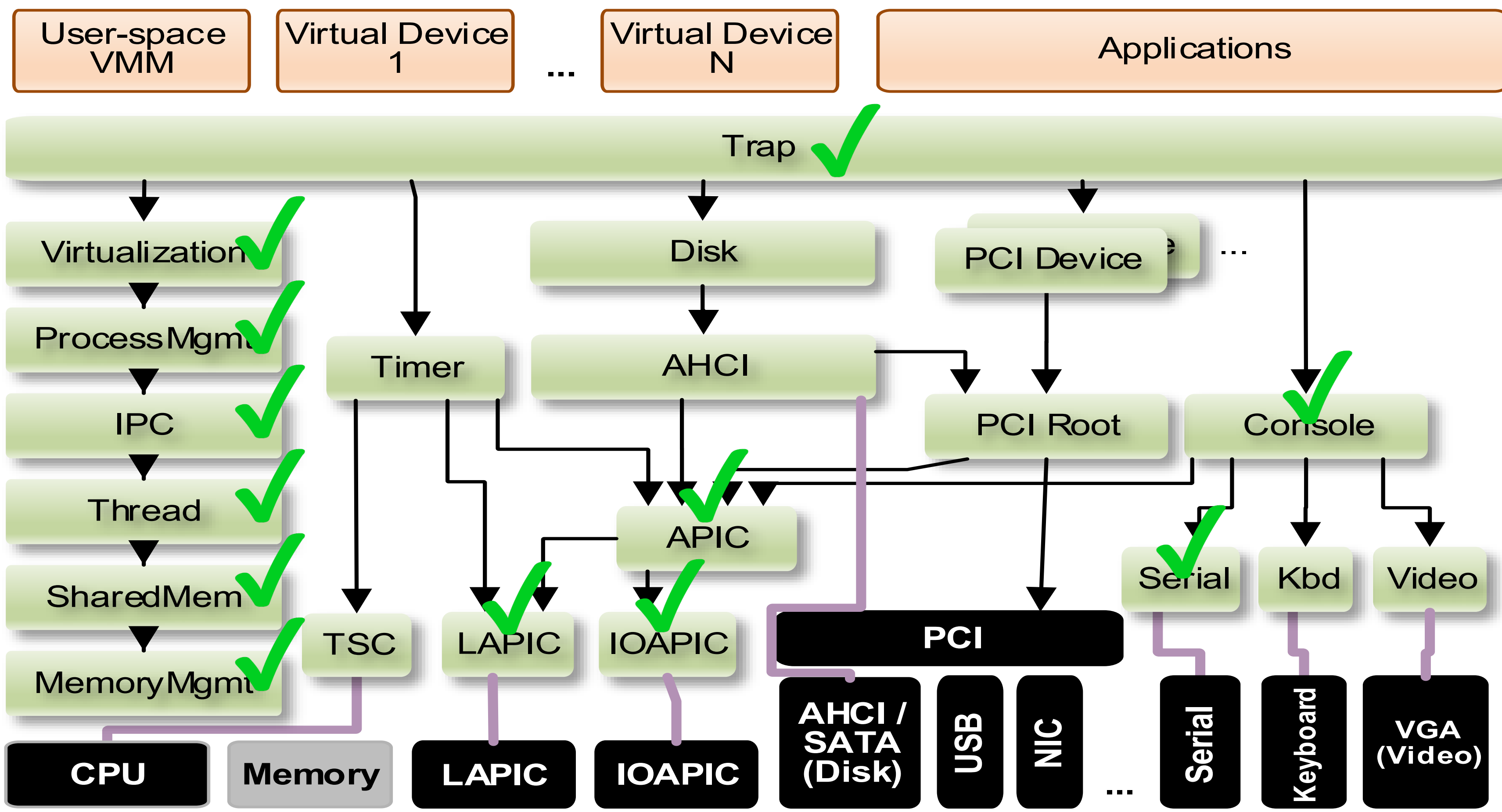
- The driver code of each device runs on its own “logical CPU”, operates its own internal states.
- Interruptible code can be naturally reasoned on top of the abstract interrupt model.



Our Contributions

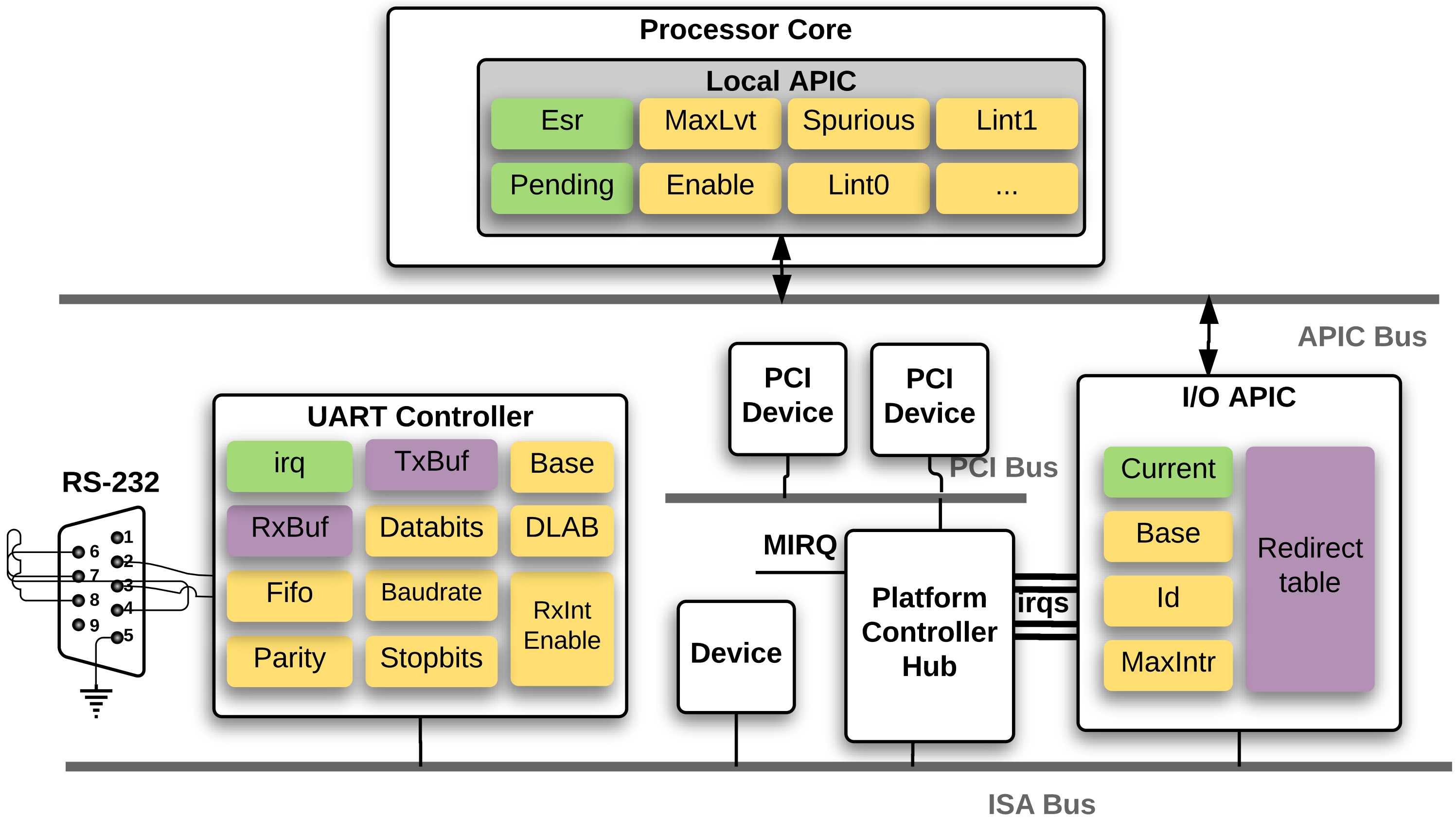
- New techniques for certifying abstraction layers with multiple ***logical CPUs*** and devices.
- New techniques for building formal ***certified device hierarchies***.
- An abstraction-layer-based approach for reasoning about ***interrupts***.
- ***Case study***: interruptible mCertiKOS with device drivers.

Interruptible mCertiKOS with Drivers



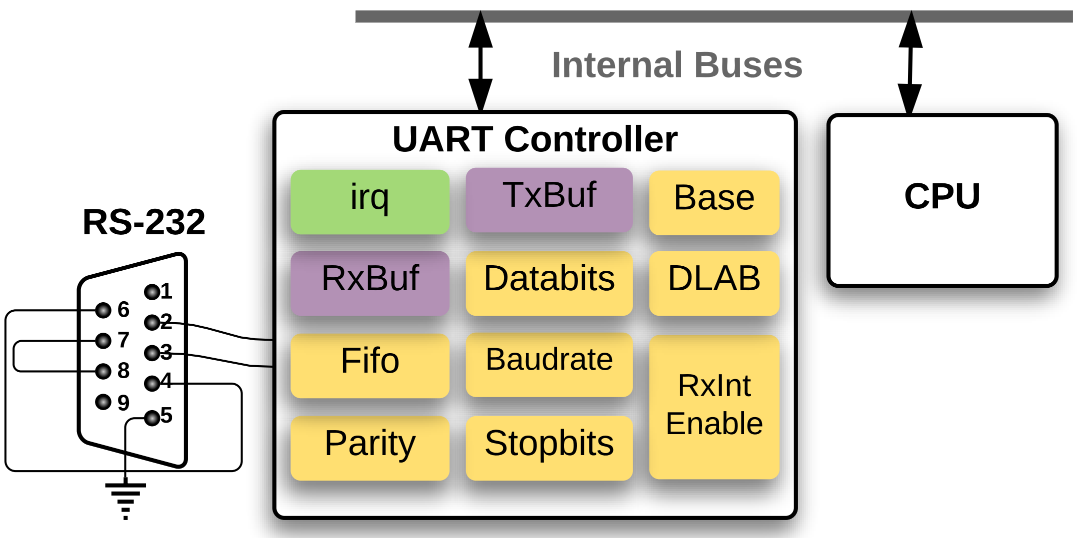
Case Study: Modeling HW Devices

- Serial Port, I/O APIC, Local APIC, CPU interrupt handling.



Case Study: Serial Device

- States: see figure
- Transitions: serial_trans_env + serial_trans_IO
- Read/Write primitives: serial_read / serial_write



Serial Interrupt Handler

```c
void serial_intr () {
  unsigned int hasMore;
  int t = 0;
  hasMore = serial_getc ();
  while (hasMore && t < CONSOLE_BUFFER_SIZE) {
    hasMore = serial_getc ();
    t++;
  }
}
```

```c
unsigned int serial_getc () {
  unsigned int rv = 0;
  unsigned int rx;
  if (serial_exists()) {
    if (serial_read(COM1 + COM_LSR, BIT1) % 2 == 1)
     {
      rx = serial_read(COM1 + COM_RX, M_ALL);
      cons_buf_write(rx);
      rv = 1;
    }
  }
  return rv;
}
```

Serial Driver

```
void serial_puts(char * s, int len) {
  int i = 0;
  while (i < len && s[i] != 0) {
    serial_intr_disable ();
    serial_putc (s[i]);
    serial_intr_enable ();
    i++;
  }
}
```

```
void serial_putc (unsigned int c) {
  unsigned int lsr = 0, i;
  if ( serial_exists() ){
    for (i = 0; !lsr && i < 12800; i++) {
      lsr = serial_read(0x3FD) & 0x20;
      delay();
    }
    serial_write (0x3F8, c);
    ...
```

What We Have Proved

- Total functional correctness.
- Safety.
- *Contextual refinement* between the lowest and the top level abstract machine:
  $\forall P,\ [\![K \bowtie P]\!]_{\mathrm{x86}} \sqsubseteq [\![P]\!]_{\mathrm{mCertiKOS}}.$
- Data invariants:
  - Console's circular buffer is always well-formed.
  - Interrupt controller states are always consistent.
- The framework also ensures that:
  - No code injection attacks, buffer overflow, integer overflow, null pointer access, *etc*.

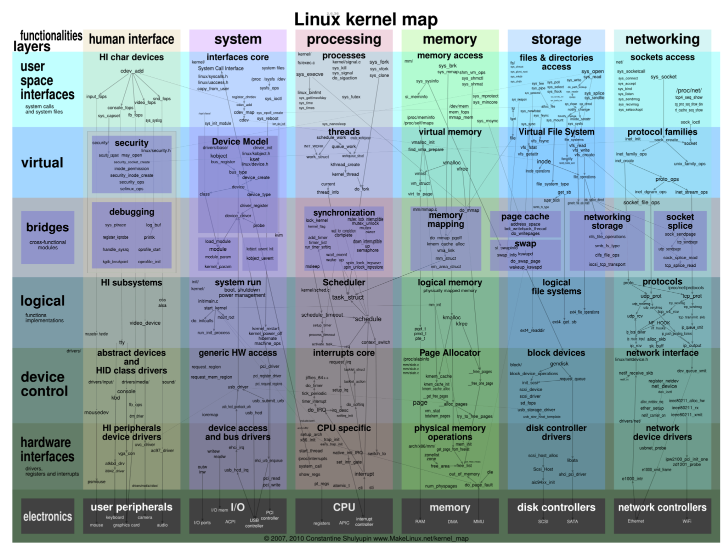
Size of TCB and Spec/Proof

- In the TCB
  - X86 hardware model
  - Hardware device/interrupt model (510 LOC)
  - System call specification (126 LOC)
  - Bootloader
  - Coq proof checker
  - Pretty-printing phase of the CompCert compiler

- Rest of the spec/proof (about 20k LOC)
  - Intermediate and auxiliary specifications and definitions
  - Coq proof scripts

Conclusion

- Compositional framework for building certified interruptible kernel with device drivers.
  - Certified abstraction layers with multiple logical CPUs.
  - An abstraction-layer-based approach for expressing interrupts.
- The first formally verified interruptible OS kernel with device drivers.
- Extensions:
  - Other drivers
  - Concurrency
  - Larger kernel



Thank You