Virtualization in Embedded Systems

Gernot Heiser
NICTA and UNSW and Open Kernel Labs
Sydney, Australia

Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Members
NICTA Partners

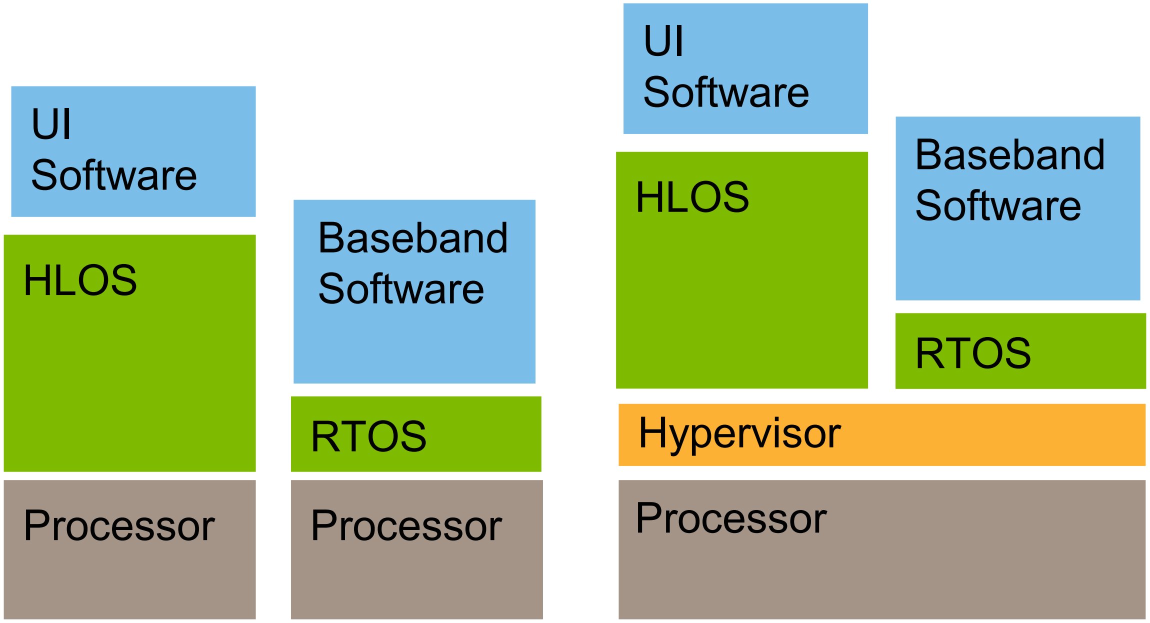
Outline

- Embedded virtualization use cases
- Enterprise vs embedded: main differences
- Trustworthy hypervisor for embedded systems
- Wishlist for Intel

Why Virtualization in Embedded Systems?

Use case 1: Mobile phone processor consolidation

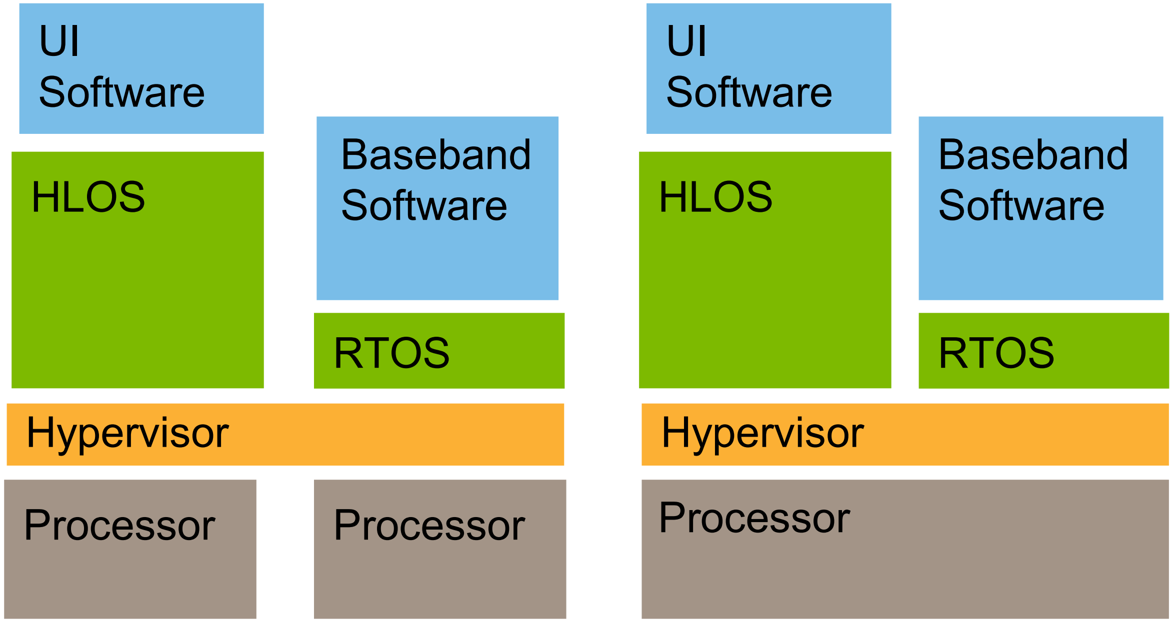
- High-end phones run high-level OS (Linux/WinCE/Symbian) on app processor
  - supports complex UI software
- Baseband processing supported by real-time OS (RTOS)
- Medium-range phone needs less grunt
  - can share processor
  - two VMs on one physical processor
  - hardware cost reduction



Why Virtualization in Embedded Systems?

Use case 1a: Software architecture abstraction

- Support for *product series*
  - range of related products of varying capabilities
- Same low-level software for high- and medium-end devices
- Benefits:
  - time-to-market
  - engineering cost

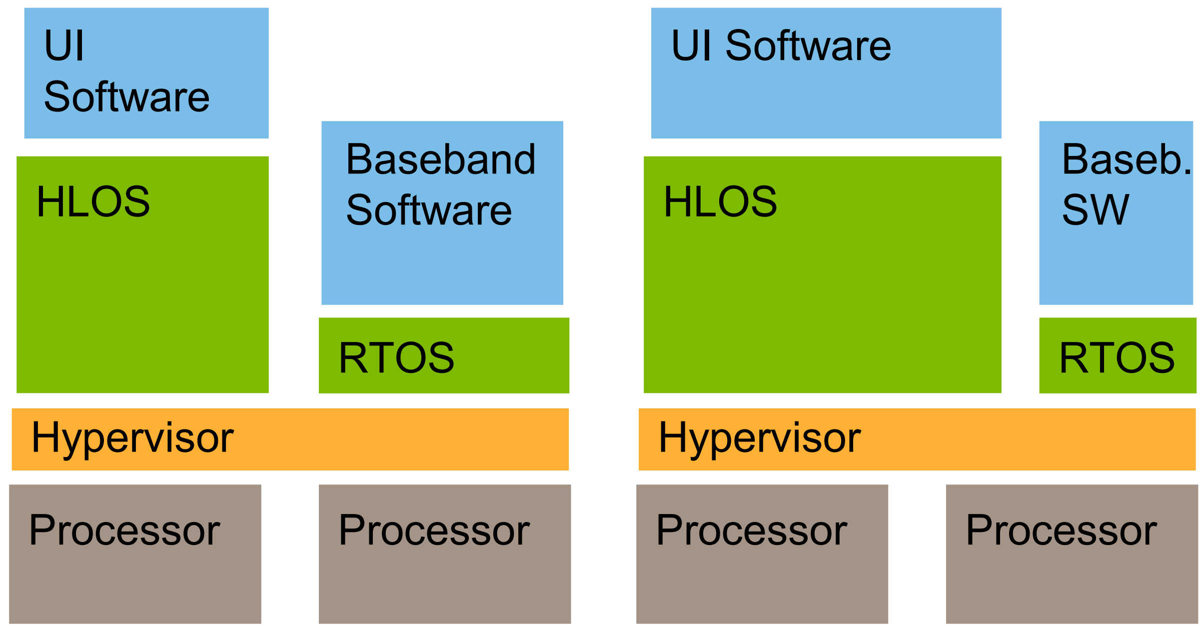


Why Virtualization in Embedded Systems?

Use case 1b: Dynamic processor allocation

→ Allocate share of baseband processor to application OS

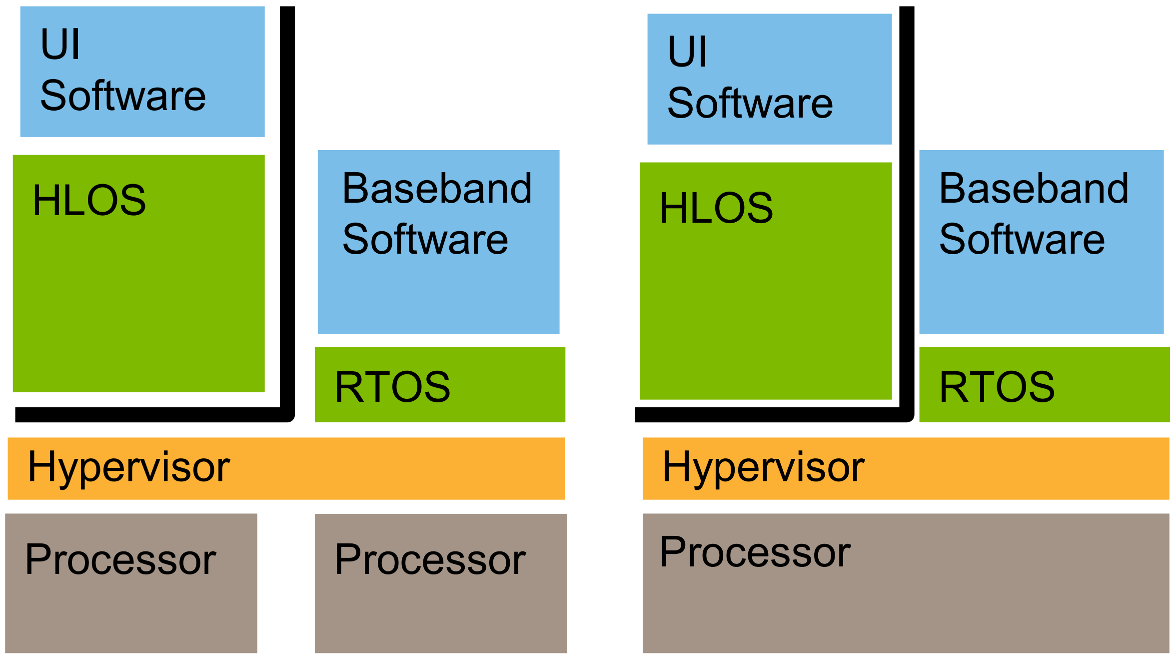
- Provide extra CPU power during high-load periods (media play)
- Better processor utilisation ⇒ higher performance with lower-end hardware
- HW cost reduction



Why Virtualization in Embedded Systems?

Use case 2: Certification re-use

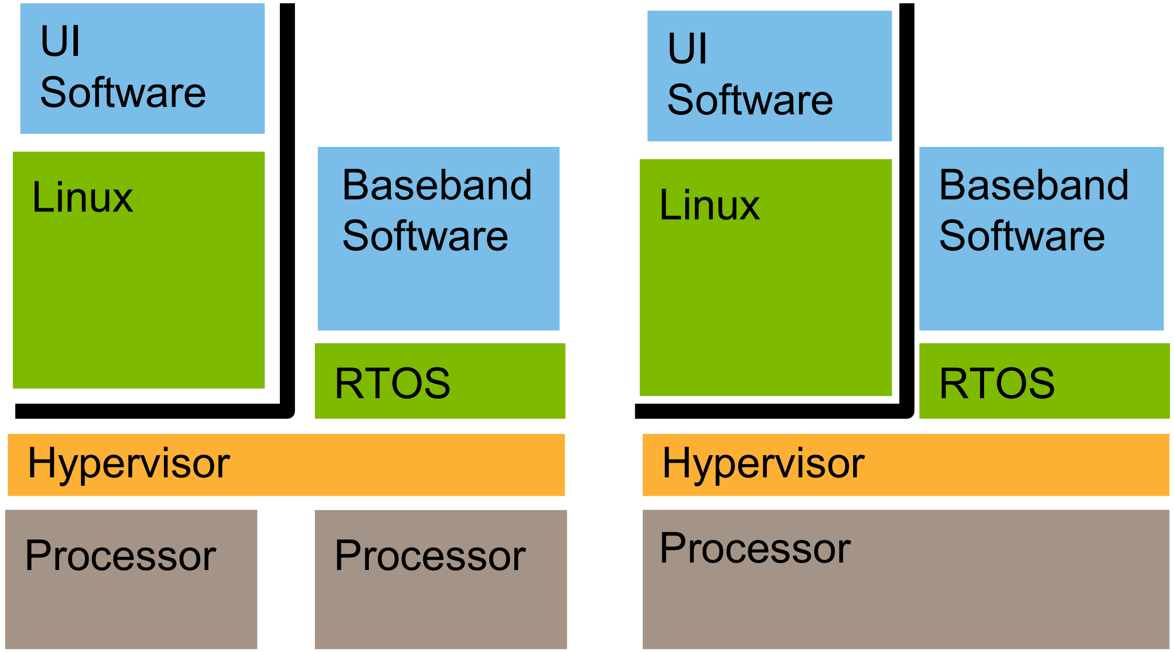
- Phones need to be certified to comply with communication standards
- Any change that (potentially) affects comms needs re-certification
- UI part of system changes frequently
- Encapsulation of UI
  - provided by VM
  - avoids need for costly re-certification



Why Virtualization in Embedded Systems?

Use case 2a: Open phone with user-configured OS

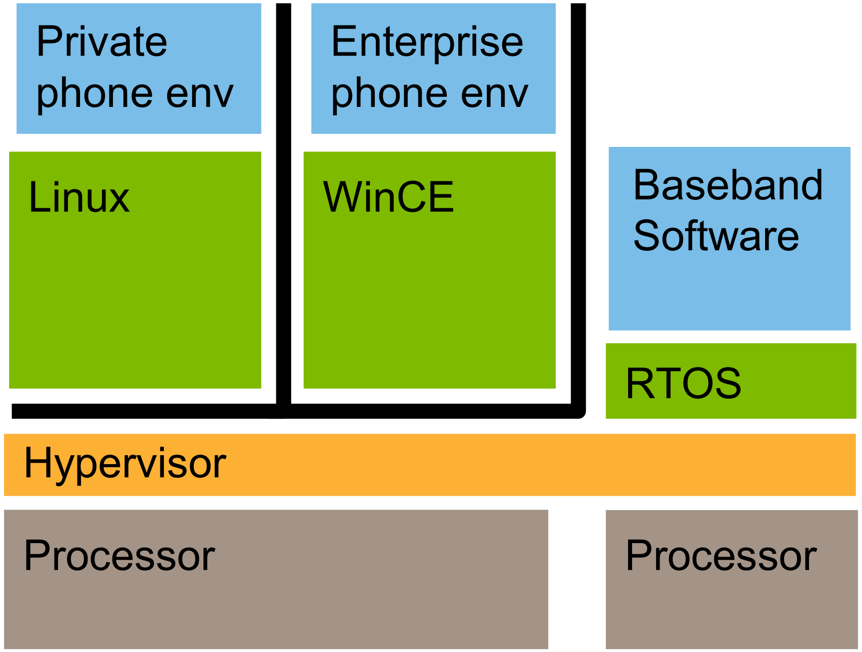
- Give users control over the application environment
  - perfect match for Linux
- Requires strong encapsulation of application environment
  - without undermining performance!



Why Virtualization in Embedded Systems?

Use case 2b: Phone with private and enterprise environment

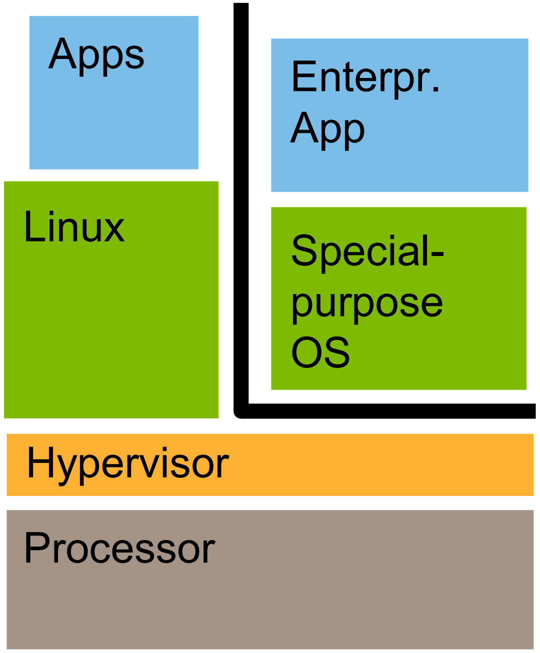
- → Work phone environment integrated with enterprise IT system
- → Private phone environment contains sensitive personal data
- → Mutual distrust between the environments ⇒ strong isolation needed



Why Virtualization in Embedded Systems?

Use case 3: Mobile internet device (MID) with enterprise app

- MID is open device, controlled by owner
- Enterprise app is closed and controlled by enterprise IT dept
- Hypervisor provides isolation



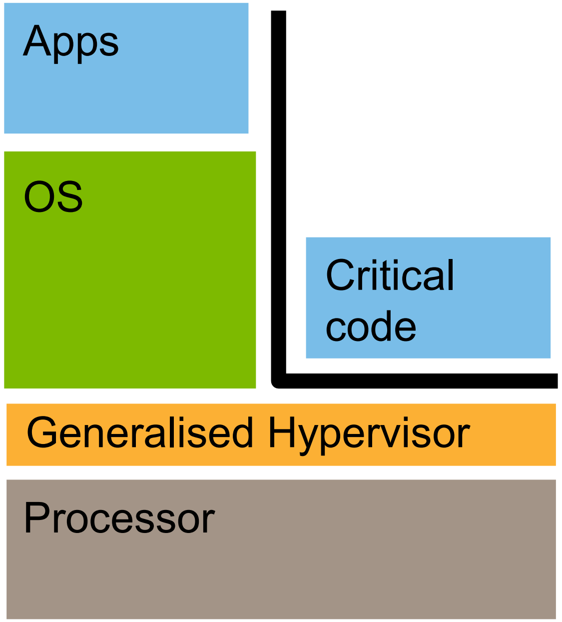
Why Virtualization in Embedded Systems?

Use case 3a: Environment with minimal *trusted computing base* (TCB)

→ Minimise exposure of highly security-critical service to other code

→ Avoid even an OS, provide minimal trusted environment

- need a minimal programming environment
- goes beyond capabilities of normal hypervisor
- requires basic OS functionality



Why Virtualization in Embedded Systems?

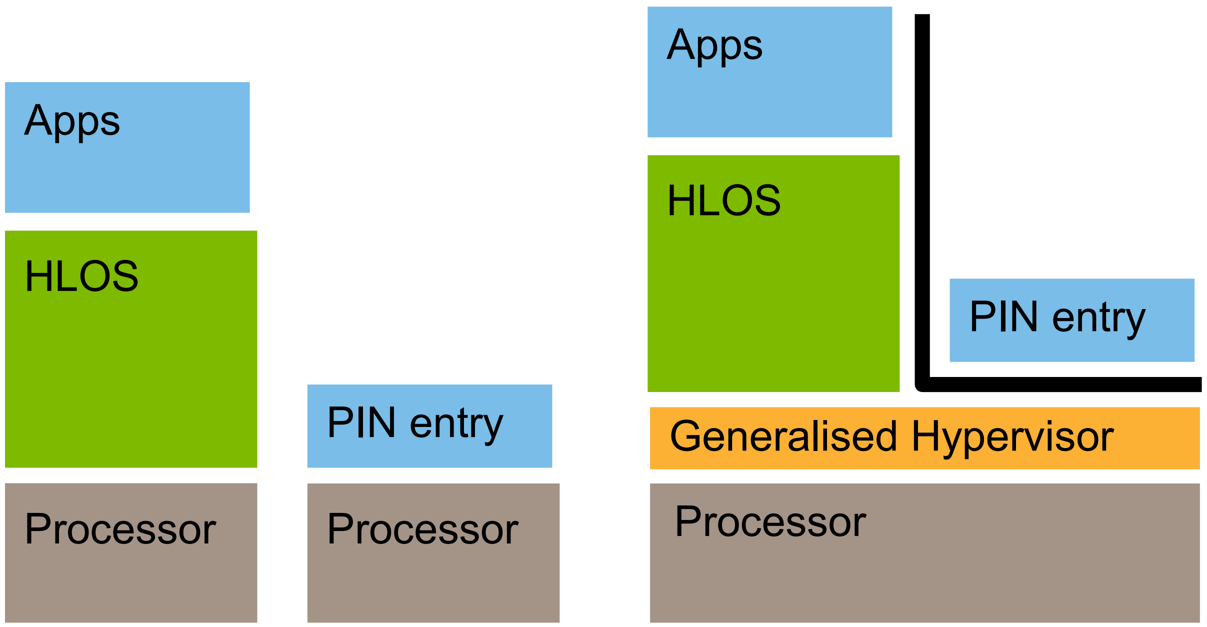
NICTA

Use case 3b: Point-of-sale (POS) device

→ May be stand-alone or integrated with other device (eg phone)

→ Financial services providers require strong isolation

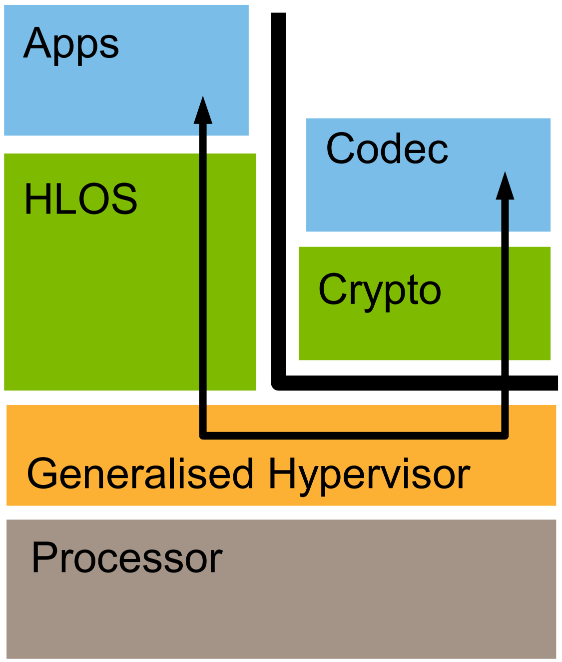
- dedicated processor for PIN/key entry
- use dedicated *virtual processor* ⇒ HW cost reduction



Why Virtualization in Embedded Systems?

Use case 4: DRM on open device

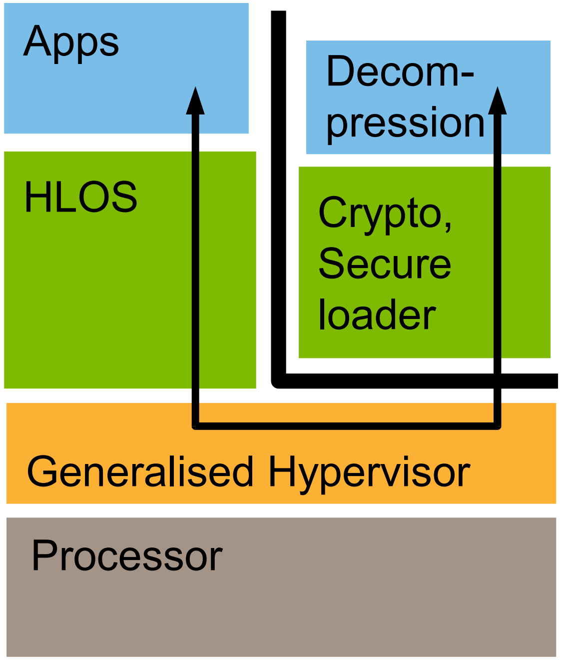
- → Device runs Linux as app OS, uses Linux-based media player
- → DRM must not rely on Linux
- → Need trustworthy code that
  - loads media content into on-chip RAM
  - decrypts and decodes content
  - allows Linux-based player to disply
- → Need to protect data from guest OS



Why Virtualization in Embedded Systems?

Use case 4a: IP protection in set-top box

- → STB runs Linux for UI, but also contains highly valuable IP
  - highly-efficient, proprietary compression algorithm
- → Operates in hostile environment
  - reverse engineering of algorithms
- → Need highly-trustworthy code that
  - loads code from Flash into on-chip RAM
  - decrypts code
  - runs code protected from interference

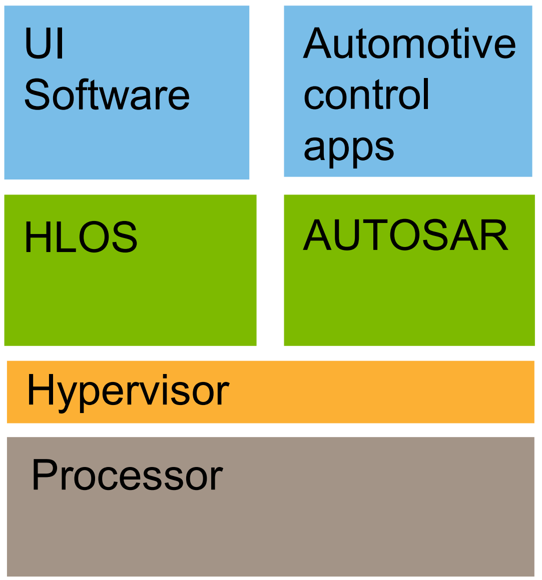


Why Virtualization in Embedded Systems?

NICTA

Use case 5: Automotive control and infotainment

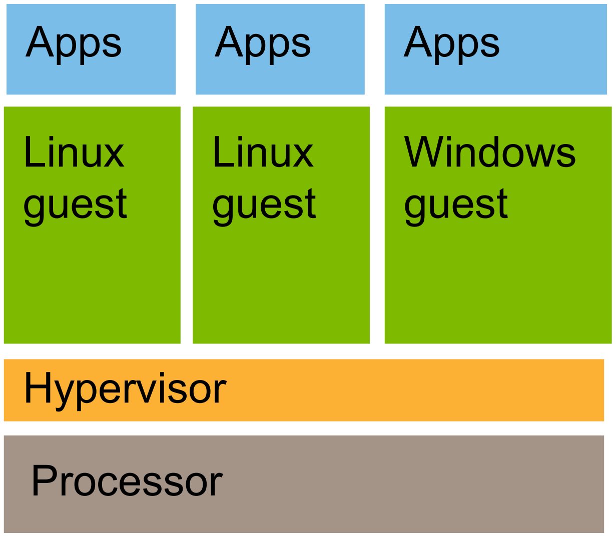
- Trend to processor consolidation in automotive industry
  - top-end cars have > 100 CPUs!
  - cost, complexity and space pressures to reduce by an order of magnitude
  - AUTOSAR OS standard addressing this for control/convenience function
- Increasing importance of *Infotainment*
  - driver information and entertainment function
  - not addressed by AUTOSAR
- Increasing overlap of infotainment and control/convenience
  - eg park-distance control using infotainment display
  - benefits from being located on same CPU

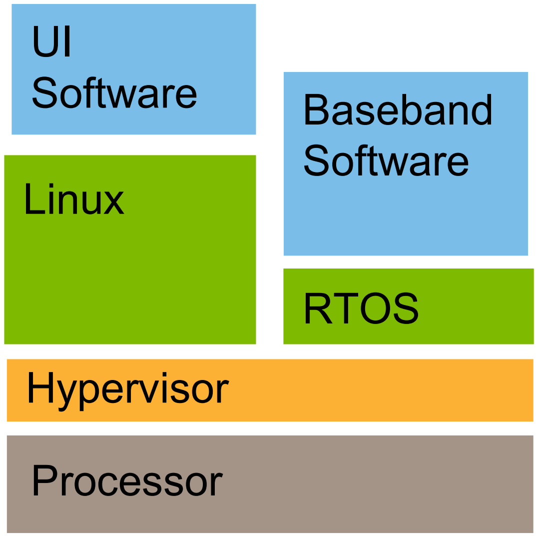


Enterprise vs Embedded Virtualization

Homogenous vs heterogenous guests

- Enterprise: many similar guests
  - hypervisor size irrelevant
  - VMs scheduled round-robin
- Embedded: 1 HLOS + 1 RTOS
  - hypervisor resource-constrained
  - interrupt latencies matter





Core Difference: Isolation vs Cooperation

Enterprise

- Independent services
- Emphasis on isolation
- Inter-VM communication is secondary
  - performance secondary
- VMs connected to Internet (and thus to each other)

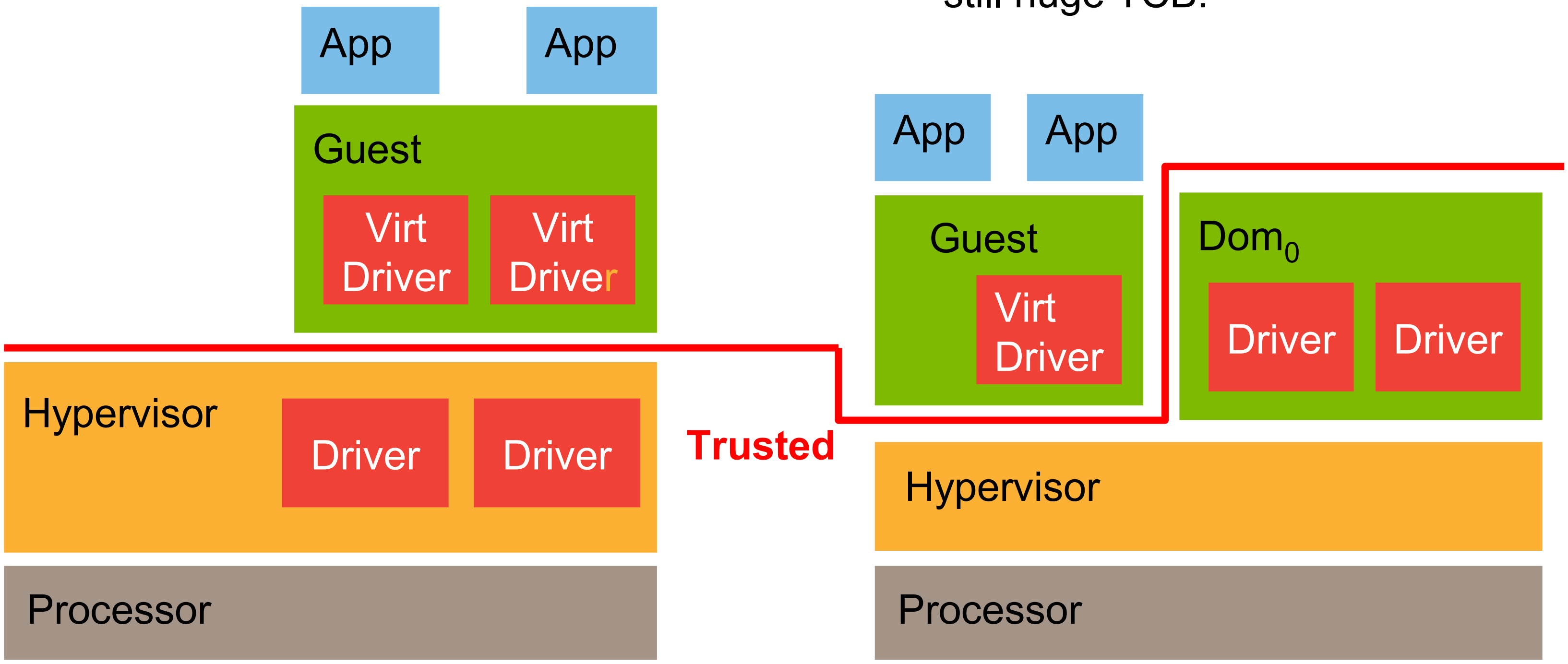
Embedded

- Integrated system
- Cooperation with protection
- Inter-VM communication is critically important
  - performance crucial
- VMs are subsystems accessing shared (but restricted) resources

Enterprise vs Embedded Virtualization

Devices in enterprise-style virtual machines

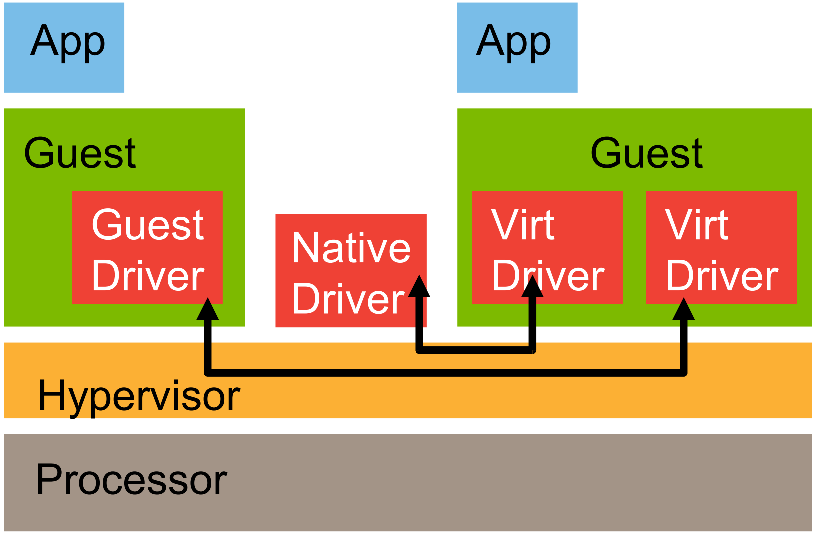
- Hypervisor owns all devices
- Drivers in hypervisor
  - need to port all drivers
  - huge TCB
- Drivers in privileged guest OS
  - can leverage guest's driver support
  - need to trust driver OS
  - still huge TCB!



Enterprise vs Embedded Virtualization

Devices in embedded virtual machines

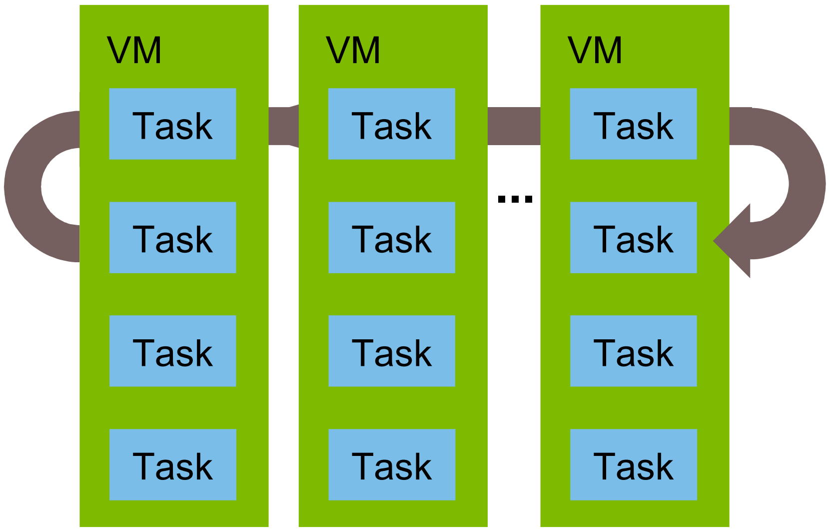
- Some devices owned by particular VM
- Some devices shared
- Some devices too sensitive to trust any guest
- Driver OS too resource hungry
- Use isolated drivers
  - protected from other drivers
  - protected from guest OSes



Isolation vs Cooperation: Scheduling

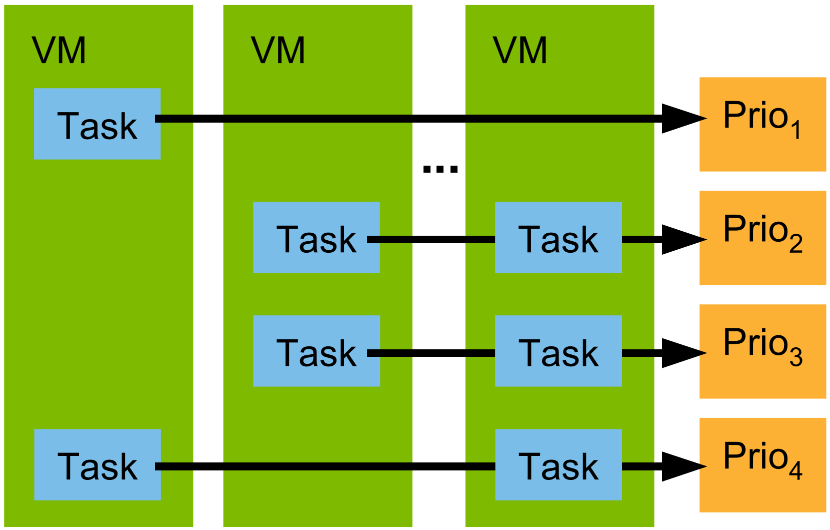
Enterprise

- Round-robin scheduling of VMs
- Guest OS schedules its apps



Embedded

- Global view of scheduling
- Schedule threads, not VMs

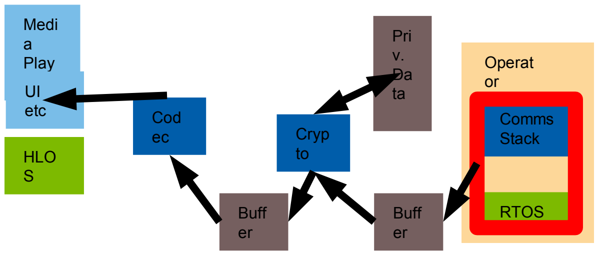


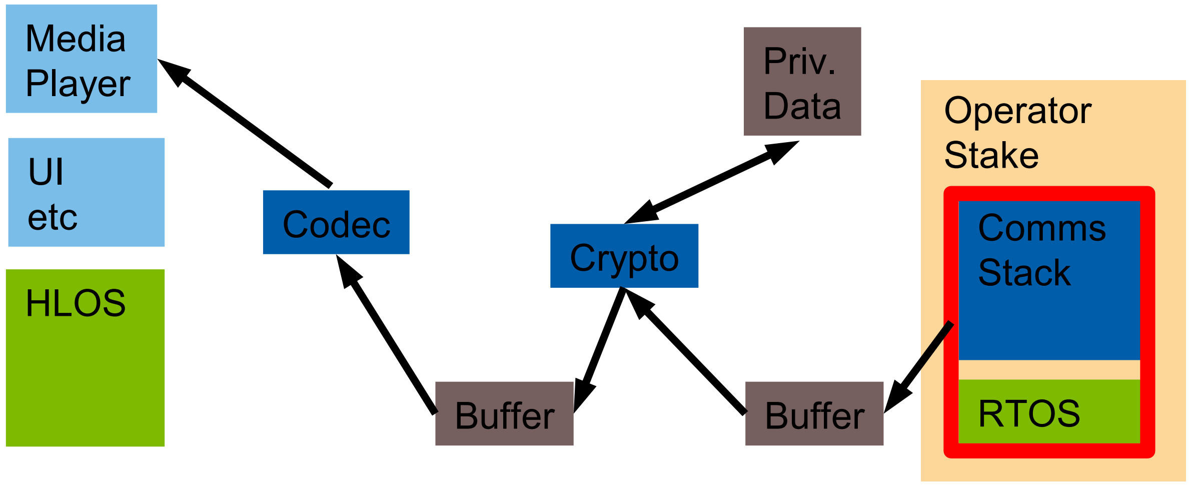
- Similar for *energy management*:
  - energy is a global resource
  - optimal per-VM energy policies are not globally optimal

Inter-VM Communication Control

Modern embedded systems are multi-user devices!

- Eg a phone has three *classes* of "users":
  - the network operator(s)
    - assets: cellular network

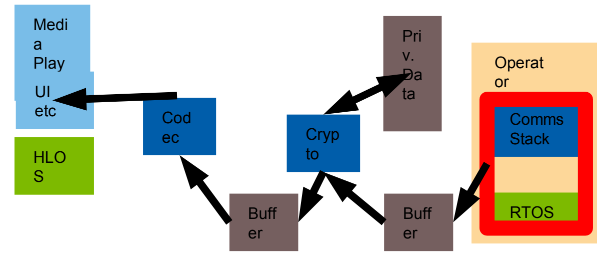


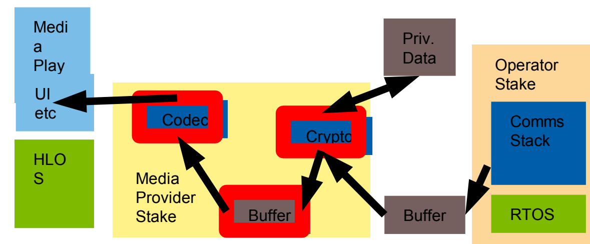


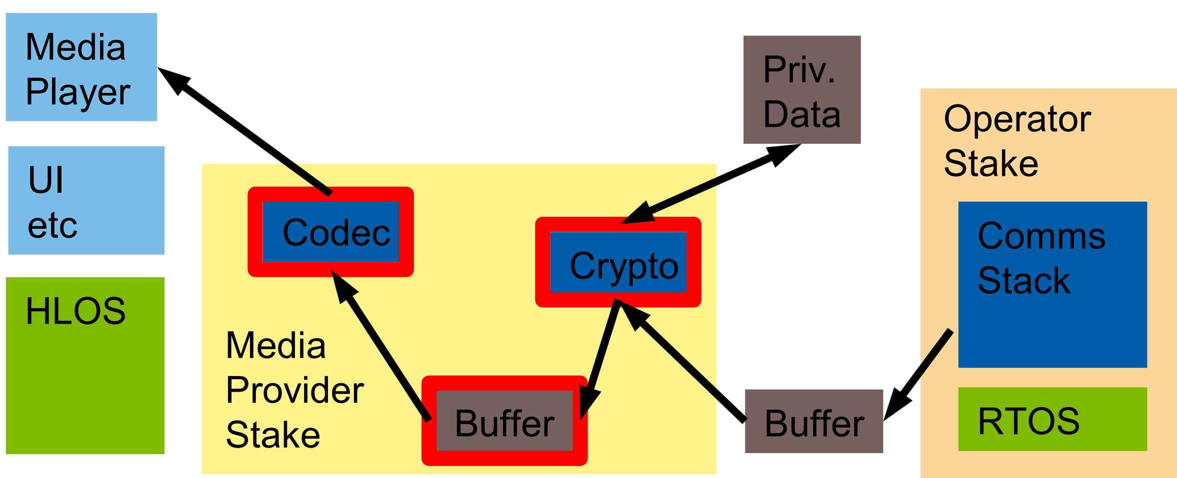
Inter-VM Communication Control

Modern embedded systems are multi-user devices!

- Eg a phone has three *classes* of "users":
  - the network operator(s)
    - assets: cellular network
  - content providers
    - media content



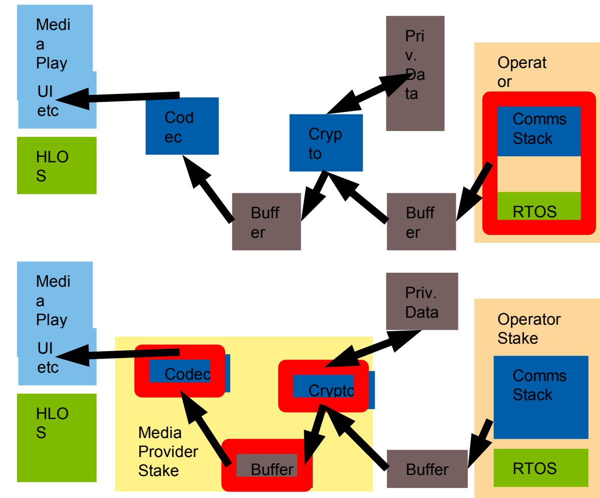


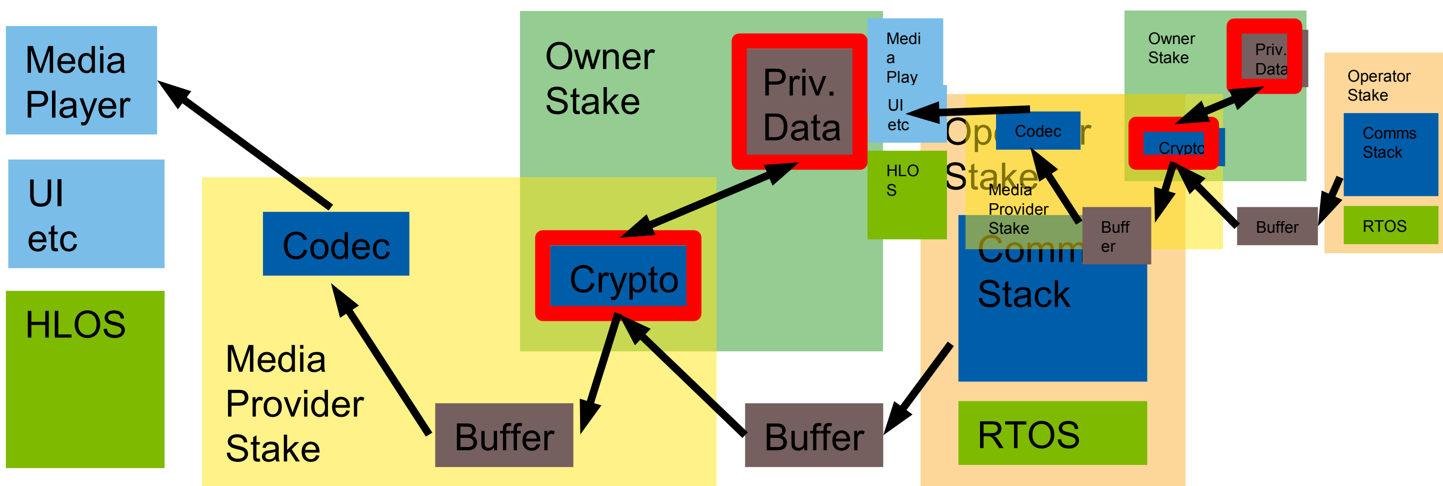


Inter-VM Communication Control

Modern embedded systems are multi-user devices!

- Eg a phone has three *classes* of “users”:
  - the network operator(s)
    - assets: cellular network
  - content providers
    - media content
  - the owner of the physical device
    - assets: private data, access keys

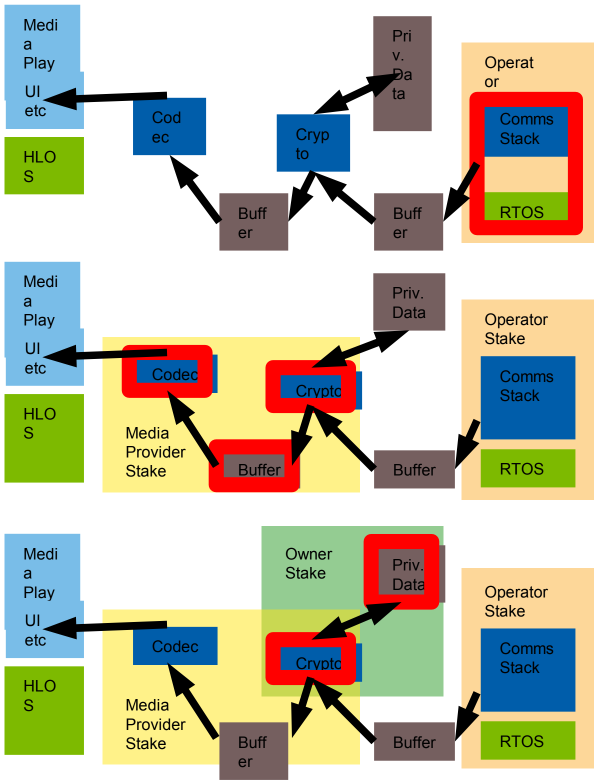




Inter-VM Communication Control

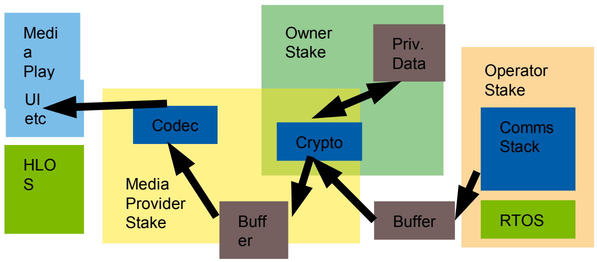
Modern embedded systems are multi-user devices!

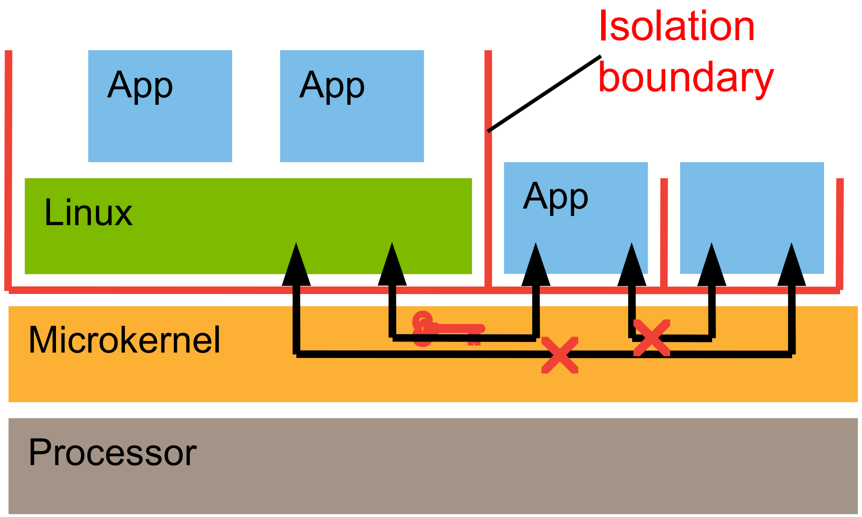
- Eg a phone has three *classes* of "users":
  - the network operator(s)
    - assets: cellular network
  - content providers
    - media content
  - the owner of the physical device
    - assets: private data, access keys
- They are mutually distrusting
  - need to protect integrity and confidentiality against *internal* exploits
  - need control over *information flow*
    - strict control over who has access to what
    - strict control over communication channels



Inter-VM Communication Control

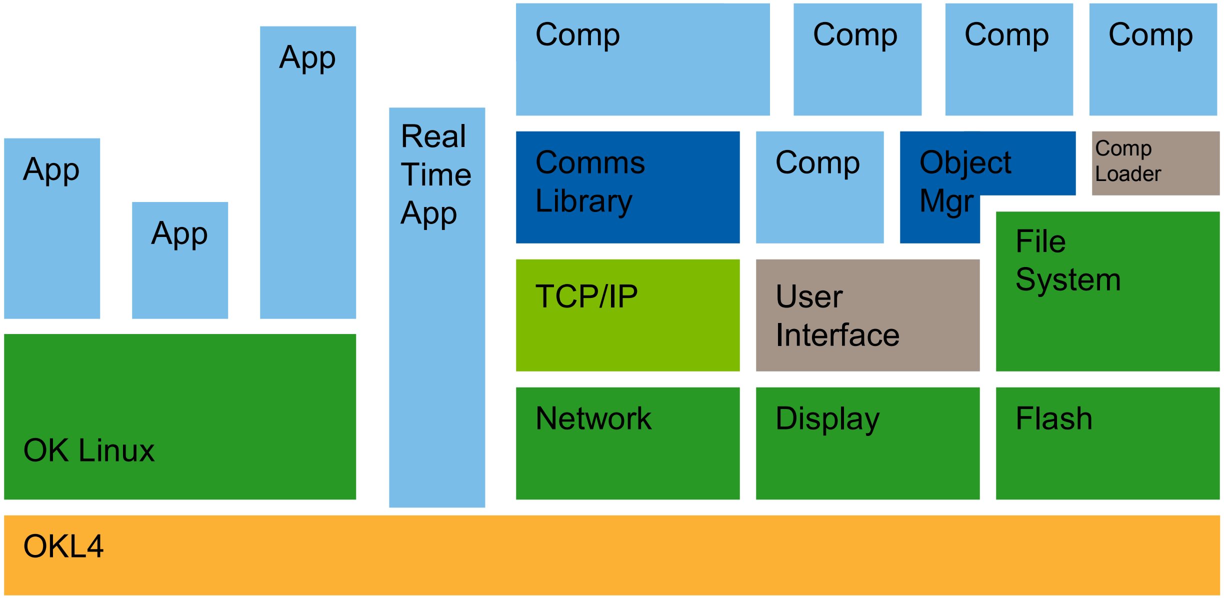
- Different "users" are mutually distrusting
- Need strong protection / information-flow control between them
- Isolation boundaries ≠ VM boundaries
  - some are much smaller than VMs
    - individual buffers, programs
  - some contain VMs
  - some overlap VMs
- Need to define information flow between isolation domains





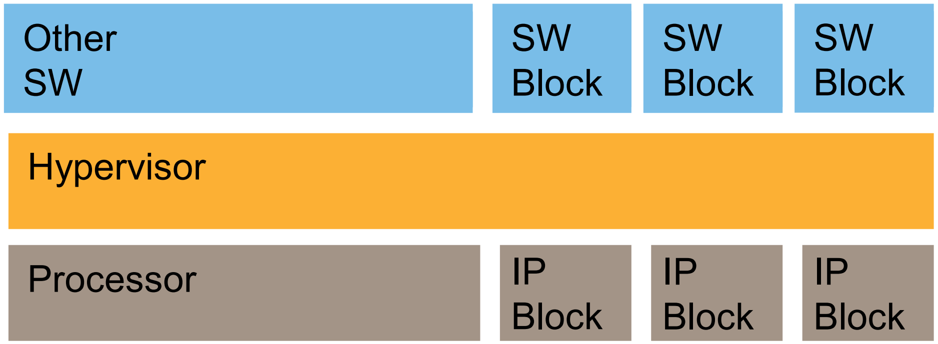
High Safety/Reliability Requirements

- Software complexity is mushrooming in embedded systems too
  - millions of lines of code
- Some have very high safety or reliability requirements
- Need divide-and-conquer approach to software reliability
  - Highly componentized systems to enable fault tolerance



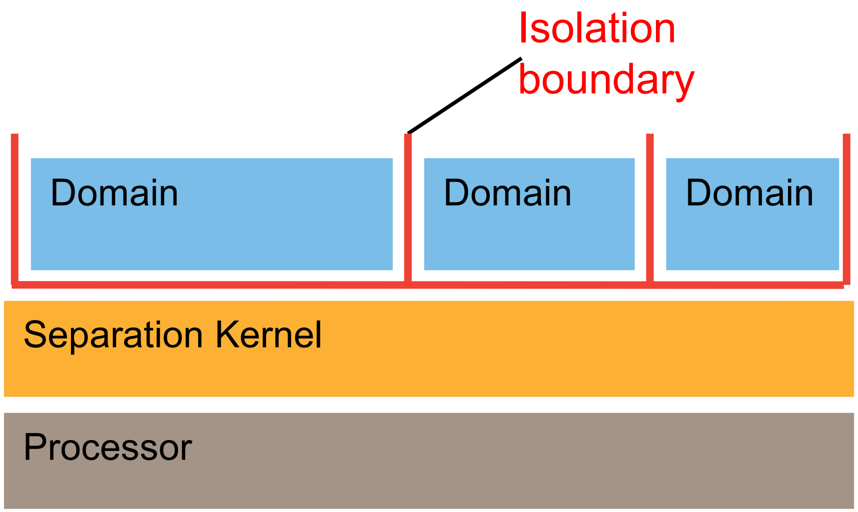
Componentization for IP Blocks

- Match HW IP blocks with SW IP blocks
- HW IP owner provides matching SW blocks
  - encapsulate SW to ensure correct operation
  - Stable interfaces despite changing HW/SW boundary



Componentization for Security — MILS

- *MILS architecture*: multiple independent levels of security
- Approach to making security verification of complex systems tractable
- *Separation kernel* provides strong security isolation between subsystems
- High-grade verification requires small components



Embedded Systems Requirements

- Sliding scale of isolation from individual program to VM running full-blown OS
  - isolation domains, information-flow control
- Global scheduling and power management
  - no strict VM-hypervisor hierarchy
  - increased hypervisor-guest interaction
- High degree of sharing is essential and performance-critical
  - high bandwidth, low latency communication, subject to security policies
- Real-time response
  - fast and predictable switches to device driver / RT stack
- High safety/security requirements
  - need to maintain minimal TCB
  - need to support componentized software architecture / MILS

Virtualization in embedded systems is good, but different from enterprise

- requires range of isolation granularities
- requires efficient context switching

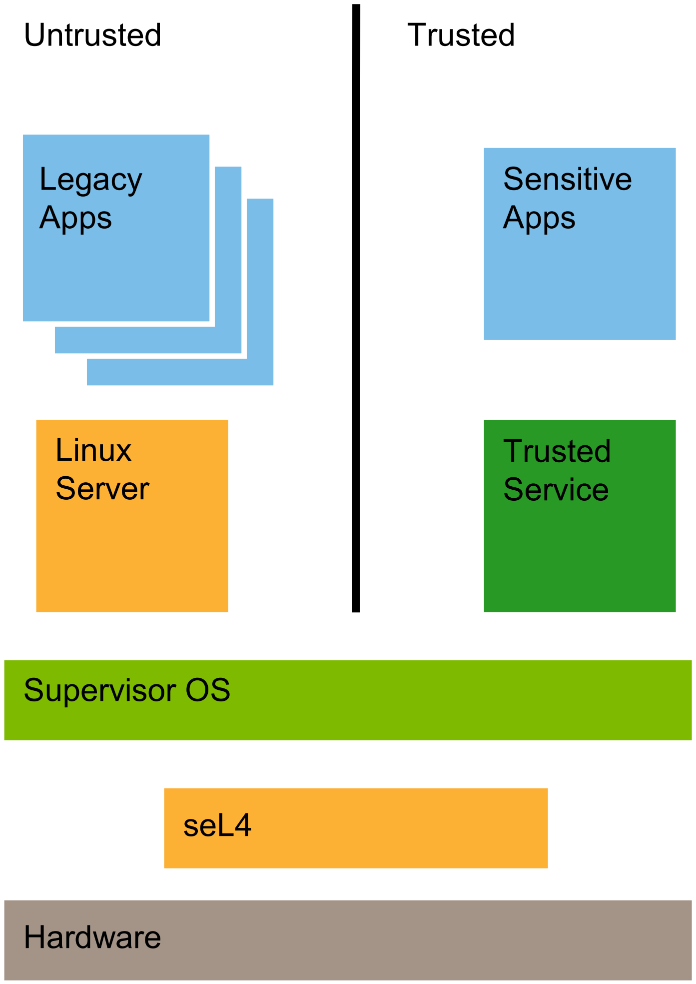
The seL4 Microkernel

Goals

- Platform for building arbitrary embedded-systems software
  - general OS
  - hypervisor
  - separation kernel
- High-performance implementation
  - no more than 15 cycles slower on IPC than L4
- Formal specification
- Formal proof of security properties
- Formal verification of implementation

Innovation over other L4 kernels:

- Access control based on capabilities
- Kernel resource accounting



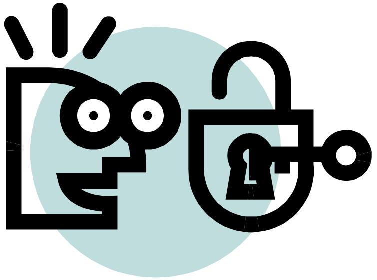
seL4 Capability-Based Protection

All authority conferred via capabilities

- Capabilities are like keys
  - Possess the key, and you can invoke the operation
- All system calls are invoked via capabilities
  - No ambient authority

Established body of knowledge on capabilities

- Can *reason* about them
- Models for confining authority



seL4 Physical Memory Management

Some kernel memory is statically allocated at boot time

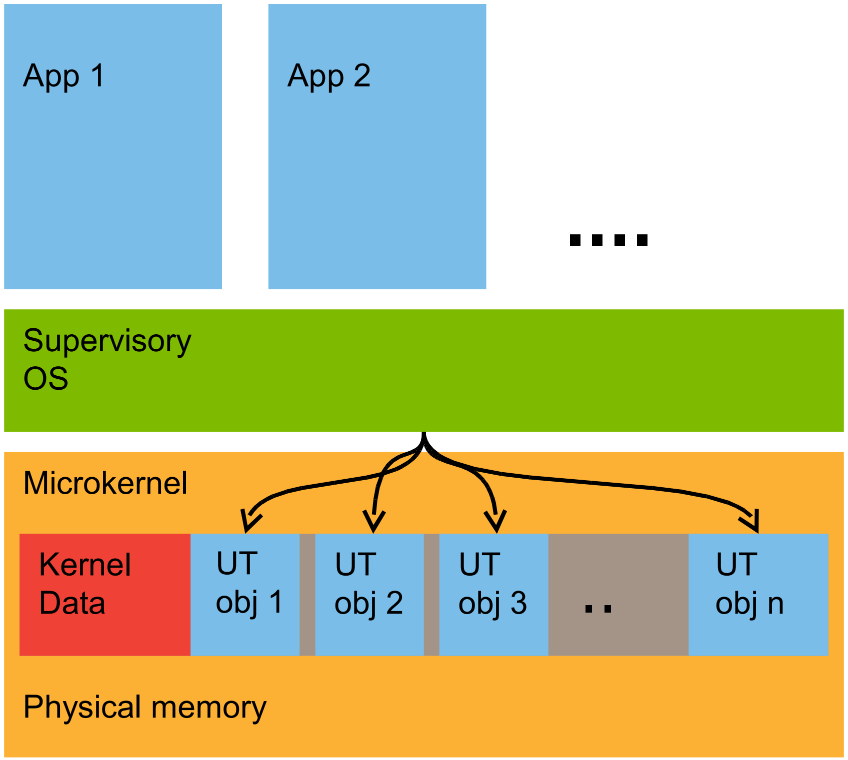
Remainder is divided into untyped (UT) objects

- 2^n region of physical memory
- Size aligned

Supervisor gets authority over these objects

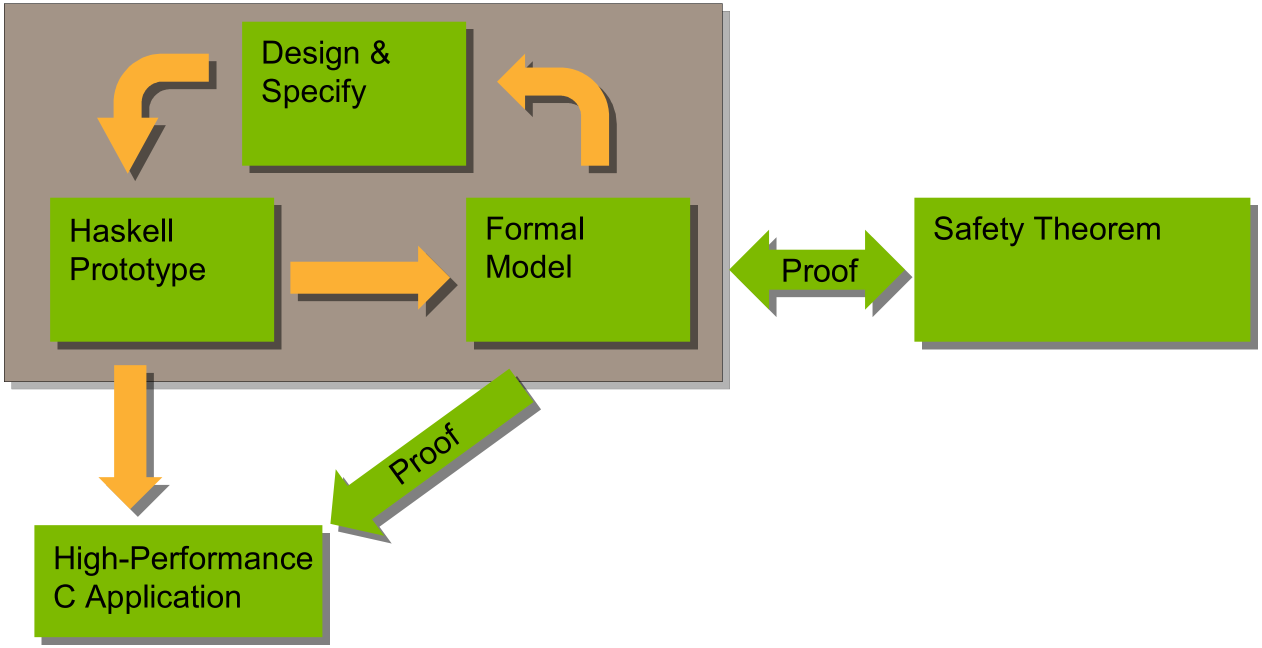
- Authority conferred by capabilities

Kernel never allocates dynamic memory

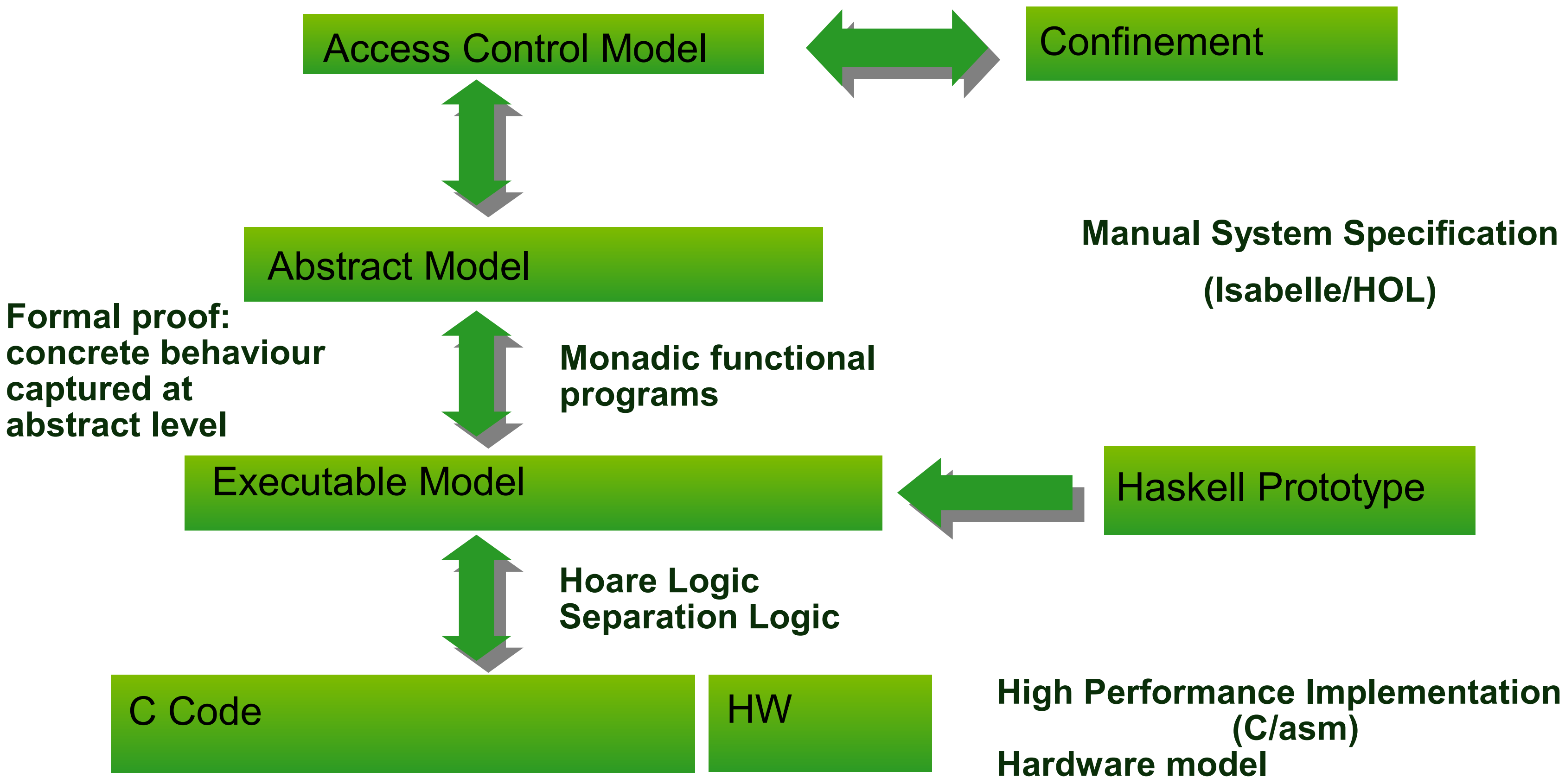


Iterative Design and Formalisation

- Prototype kernel executes native binaries on simulator
- Exposes usability issues early
- Tight formal design integration



The Proofs


Access Control Model
Confinement
Manual System Specification
(Isabelle/HOL)
Abstract Model
Formal proof:
concrete behaviour
captured at
abstract level
Monadic functional
programs
Executable Model
Haskell Prototype
Hoare Logic
Separation Logic
C Code
HW
High Performance Implementation
(C/asm)
Hardware model

NICTA

```
datatype
  rights = Read
         | Write
         | Grant
         | Create

record cap =
  entity :: entity_id
  rig
reco
ty
```

```
lemma isolation:
  "⟦sane s;
    s' ∈ execute cmds s;
    isEntityOf s e_s;
    sEntityOf s e;
    ntity c = e;
    :> subSysCaps s e_s⟧
    Caps s' e_s"
```

```
constdefs
  schedule :: "unit s_monad"
  "schedule ≡ do
```

```
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
                                                    read
```

```
tcb_t * scheduler_t::find_next_thread(prio_queue_t * prio_queue)
{
    ASSERT(DEBUG, prio_queue);

    if (prio_queue->index_bitmap) {
        word_t top_word = msb(prio_queue->index_bitmap);
        word_t offset = BITS_WORD * top_word;

        for (long i = top_word; i >= 0; i--)
        {
            word_t bitmap = prio_queue->prio_bitmap[i];

            if (bitmap == 0)
                goto update;

            do {
                word_t bit = msb(bitmap);
                word_t prio = bit + offset;
                tcb_t *tcb = prio_queue->get(prio);
```

ecification

DL)

be

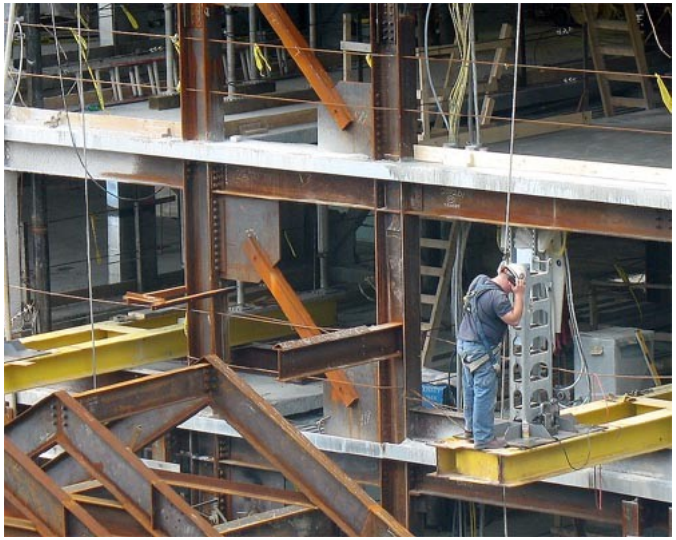
ce Implementation
C/asm)

Common Criteria Assurance and L4.verified

NICTA

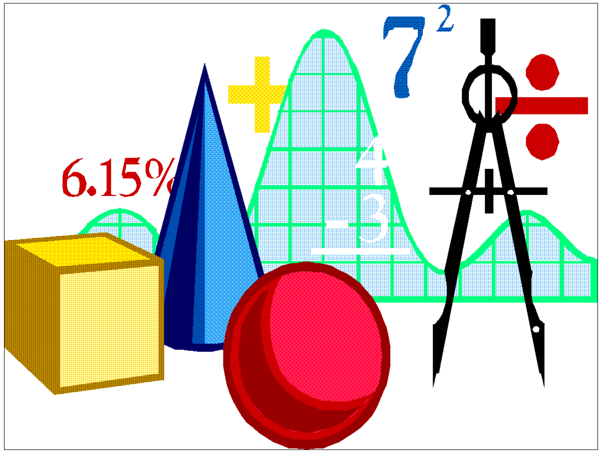
EAL	Requirem.	Funct Spec	HLD	LLD	Implem.
EAL 1	Informal	Informal	Informal	Informal	Informal
EAL 2	Informal	Informal	Informal	Informal	Informal
EAL 3	Informal	Informal	Informal	Informal	Informal
EAL 4	Informal	Informal	Informal	Informal	Informal
EAL 5	Formal	Semiformal	Semiformal	Informal	Informal
EAL 6	Formal	Semiformal	Semiformal	Semiformal	Informal
EAL 7	Formal	Formal	Formal	Semiformal	Informal
L4.verified	Formal	Formal	Formal	Formal	*Formal*

seL4 Summary



Implementation Status

- seL4 operational on ARM11
  - runs Linux etc...
- Performance in line with other L4 kernels
- Port to x86 in progress
- Security evaluation by Australian DoD
- To be integrated with commercial OKL4
  - OKL4 presently deployed in 150N devices

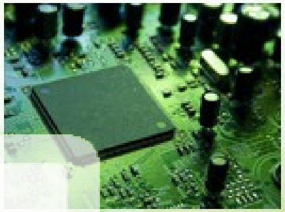


Proof Status

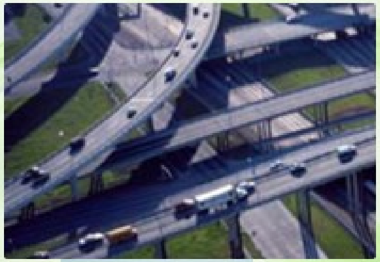
- Refinement proof to low-level model complete
- Already most deeply formally-verified general-purpose kernel ever
- C/asm implementation proof due December
- Working on proving more security properties

My Wishlist for Intel

- Need fine-grained isolation
  - *Please give us IO-MMUs on embedded processors!*
- Need fast context switches
  - *Please do something about context-switching costs*
  - *... such as tagged TLBs!*
- Need clear definition of hardware for formal verification
  - *Please give us an RTL-level description of the ISA!*



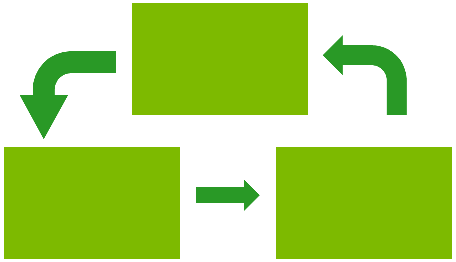
NICTA



From **imagination** to **impact**

seL4 Project Overview

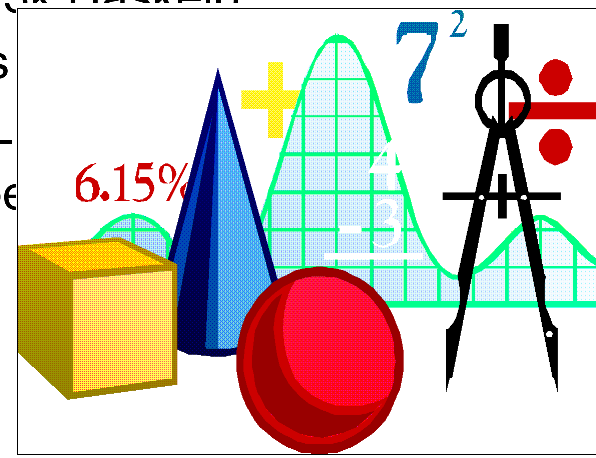
- Size of the project
  - Average 4–5 people (full time equivalent)
  - 5 years
  - Ends December 2008
- Interesting Problems
  - Designing and formalising an OS kernel
  - Refinement on monadic functional programs
  - Refinement on C programs
  - Formalizing machine details
  - Access control



seL4 Verification Summary

Statistics

- 3.5k LOC abstract, 7kLOC concrete spec (about 3k Haskell)
- Abstract / Haskell done: 100kLOP (more features
- Access control model + security proofs done (1kL
- 109 patches to Haskell kernel, 132 to abstract spe
- Performance in line with other L4 kernels



Kinds of properties proved

- Well typed references, aligned objects, ..
- Well formed thread states, endpoint and scheduler queues, ...
- All syscalls terminate, reclaiming memory is safe, ...
- Authority is distributed by caps only
- Access control is decidable
- Subsystems can be isolated / confined