seL4: Formal Verification of an OS Kernel

Prof. Phillip Gibbons

Spring 2020, Lecture 17
“seL4: Formal Verification of an OS Kernel”


- Gerwin Klein (NICTA, UNSW)
- Kevin Elphinstone (NICTA, UNSW)
- Gernot Heiser (NICTA, UNSW, Open Kernel Labs)
“seL4: Formal Verification of an OS Kernel”


SigOps HoF citation (2019):
The seL4 project was the first to provide a machine-checked proof of correctness and security properties of a high-performance microkernel. The authors used a unique approach that fuses formal and operating systems techniques, resulting in a general purpose operating system kernel that performs as well as a state-of-the-art microkernel and whose behavior can be precisely predicted for any input. The work has become the basis for a large amount of subsequent work in provably correct systems.
Formal Verification of seL4

Complete formal verification is the only known way to guarantee that a system is free of programming errors.

- 1st formal proof of functional correctness of a general-purpose OS
  - Implementation guaranteed to follow high-level specification
  - Kernel will never crash nor perform an unsafe operation
  - Can predict precisely how kernel will behave in every situation

- Assume correctness of compiler, assembly code, boot code, management of caches, the hardware—Everything else we prove

- Did not sacrifice performance

Hoare triple: {Precondition} Command {Postcondition}
Rapid Kernel Design & Implementation

seL4 features: Virtual address spaces, Threads, IPC, Capabilities for authorization
Kernel Design for Verification

- **Global variables & side effects**
  - Problems: Complex global data structure, Invariants that are temporarily violated during intermediate states
  - Solutions: Limit preemption points, Make side effects explicit by deriving from Haskell

- **Kernel memory management**
  - seL4 exports control of in-kernel allocation out of the kernel
  - Only need to prove: mechanism works
  - New objects get allocated from disjoint regions of free memory
  - Must invalidate all references before free the object
Concurrency & Non-determinism

• Restricted to sequential execution:  \( A ; X ; B \)

• Yielding at \( X \) (including preemption)
  – Challenge: Anything can run after \( A \) and before \( B \)
  – Solution: Use event-based execution with single kernel stack (originally done to reduce kernel’s memory footprint)

• Interrupts
  – Challenges: Non-deterministic execution of interrupt handlers, Timer ticks causing preemption
  – Solution: Disable interrupts, with carefully-selected interrupt points that use polling
  – Trade-off between proof complexity & interrupt latency: Need special treatment of object destruction
Concurrency & Non-determinism

• Exceptions
  – Non-issue: Exceptions other than memory/page faults
  – Solution for page faults:
    * Fixed region of VA space pinned in physical memory
    * Large enough so that kernel never page faults
    * To avoid page faults when kernel reads arguments passed to it, arguments stored in registers or in pinned region
Formal Verification

• Interactive, machine-assisted, machine-checked proof
  – Use Isabelle/HOL theorem prover

Abstract Spec

schedule ≡ do
  threads ← all_active_tCBS;
  thread ← select threads;
  switch_to_thread thread
  od OR switch_to_idle_thread

schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread

    chooseThread = do
      r <- findM chooseThread’ (reverse [)
      when (r == Nothing) $ switchToIdleThread
      chooseThread’ prio = do
        q <- getQueue prio
        liftM isJust $ findM chooseThread’
      chooseThread’’ thread = do
        runnable <- isRunnable thread
        if not runnable then do
          tcbSchedDequeue thread
          return False
Discussion: Summary Question #1

• **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Experience & Lessons Learnt

• Performance is comparable

• Verification effort: 20 person-years for the proof

• Iterative process: spec refinements, design debugging, proof

• Cost of future changes

There is one class of otherwise frequent code changes that does NOT occur after the kernel has been verified: implementation bug fixes
Forward Simulation
• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Next Class: Monday 3/16

Kernels & Virtual Machines (III)

“The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”
Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, Eddie Kohler 2013

“LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation”
Yizhou Shan, Yutong Huang, Yilun Chen, Yiyning Zhang 2018