Verifying an Operating System Kernel

Michael Norrish
Windows

An exception 06 has occurred at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
The Problem
Goals:

• Formal specification of kernel and machine
• Verified production quality, high-performance kernel

Address problems in L4:

• Communication control
• Kernel resource accounting
• No performance penalty for new features
  – 30 cycles per syscall ok. Maybe.
Overview

• The seL4 Kernel
  – Interface
  – State
  – Kernel Objects

• Interesting Problems
  – Designing and formalizing an OS kernel
  – Refinement on monadic functional programs
seL4

secure embedded L4
Kernel Interface

Kernel is a state transformer:

\[
\text{kernel} :: \text{Event} \rightarrow \text{KernelState} \rightarrow \text{KernelState}
\]
Kernel State

• Physical memory
  Storage: `obj_ref` ) `kernel_object` option

• Mapping database
  Capability derivations: `cte_ref` ) `cte_ref` option

• Current thread
  Pointer: `obj_ref`

• Machine context
  Registers, caches, etc
Kernel Objects (simplified)

- Capability Table
  
  \[
  \text{cap_ref} \rightarrow \text{capability}
  \]

- Thread Control Block (TCB)
  
  \[
  \text{record} \quad \text{ctable, vtable :: capability}
  \\
  \text{state :: thread_state}
  \\
  \text{result_endpoint, fault_endpoint :: cap_ref}
  \\
  \text{ipc_buffer :: vpage_ref}
  \\
  \text{context :: user_context}
  \]

- Endpoint:
  
  \[
  \text{Idle} \mid \text{Receive (obj_ref list)} \mid \text{Send (obj_ref list)}
  \]

- Data Page
Designing and Formalising

concrete syntax is everything
How to design and formalise a new kernel

- Design & Specify
- Formal Model
- Proof
- High-Performance C implementation
- Safety Theorem
Standard Kernel Design

Kernel Hacker View

- Design & Specify
- High-Performance C implementation
- Prototype on Real Hardware
- White-board
- Step 2
- Formal Model
- Safety Theorem
- Proof

Formal Proof

High-Performance C implementation

Safety Theorem

Prototype on Real Hardware

Design & Specify

Step 2

White-board
Formal Design

Formal Methods View

- Design & Specify
- Formal Model
- Safety Theorem
- High-Performance C implementation

Proof

Step 2

Design in Theorem Prover
Iterative Design and Formalisation

- Haskell Prototype
- Design & Specify
- Formal Model
- Safety Theorem

- High-Performance C implementation
- • prototype kernel executes native binaries on simulator
- • exposes usability issues early
- • tight formal design integration

Proof

- Proof
Haskell to Isabelle/HOL

- Needs to be quick and easy:

- Problems:
  - Size (3000 loc)
  - Real-life code (GHC extensions, no nice formal model)
  - Want Isabelle/HOL for safety and refinement proofs
  - Existing tools do not parse the code
Approach: Quick and Dirty

- In the end:
  - No “hard” translation correctness guarantee
  - Remaining issues:
    - Special features (“Dynamic”)
    - Termination
    - Monads
Termination

• **Haskell:**
  – Lazy evaluation
  – Non-terminating recursion possible

• **Isabelle/HOL:**
  – Logic of total functions
Termination

• **Haskell:**
  – Lazy evaluation
  – Non-terminating recursion possible

• **Isabelle/HOL:**
  – Logic of total functions

• **But:**
  – All system calls terminate
  – We prove termination
  – So far: done, relatively easy, not much recursion
    (cheated once, not really, though)
Monads

• **Haskell kernel:**
  – Imperative, monadic style throughout

• **Isabelle/HOL:**
  – Type system too weak to implement monads in the abstract
Monads

• **Haskell kernel:**
  – Imperative, monadic style throughout

• **Isabelle/HOL:**
  – Type system too weak to implement monads in the abstract

• **But:**
  – Strong enough to implement concrete monads (state, exception)
  – Nice do-style syntax in theorem prover
  – So far: needed more concrete than abstract properties for proofs
The Proof

Refinement on monadic functional programs
Overview

**Abstract Model**

Formal proof:
Concrete behaviour captured at abstract level

Monadic functional programs

**Executable Model**

Hoare Logic
Separation Logic

C Code

HW

Manual System Specification
(Isabelle/HOL)

Haskell Prototype

High Performance Implementation
(C/asm)
Refinement

• The old story:
  – C refines A if all behaviors of C are contained in A

• Sufficient: forward simulation

![Diagram showing relationships between states and transitions in a system, illustrating the concept of refinement.]
State Monad in Isabelle

- Nondeterministic state monad:

  \[
  \text{types } (\sigma, \alpha) \text{ monad} = \sigma \times (\alpha \times \sigma) \text{ set}
  \]

  \[
  \text{return} :: \alpha \times (\sigma, \alpha) \text{ monad}
  \]

  \[
  \text{return } x \ s = \{(x,s)\}
  \]

  \[
  \text{bind } (\gg=) :: (\sigma, \alpha) \text{ monad } \times (\alpha \times (\sigma, \beta) \text{ monad}) \times (\sigma, \beta) \text{ monad}
  \]

  \[
  f \gg= g = \lambda s. \{ v, t \} \downarrow (f s)
  \]

  \[
  \text{fail} :: (\sigma, \alpha) \text{ monad}
  \]

  \[
  \text{fail } s = \{}
  \]
Hoare Logic for the State Monad

• Hoare triples with result values:

\[ \{ P \} \ f \ \{ Q \} == \forall s. P \ s ! (\forall (r,s') \in f \ s. Q \ r \ s') \]

• WP-Rules:

\[ \{ P \ x \} \ \text{return} \ x \ \{ P \} \]

\[ \{ P \} \ f \ \{ Q \} \ \forall x. \ \{ Q \ x \} \ g \ x \ \{ R \} \]

\[ \{ P \} \ f \gg= \ g \ \{ R \} \]

\[ \{ P \} \ \text{fail} \ \{ Q \} \]
State Monad Refinement

- Forward Simulation

```
corr S R A C ==
8(s,s') 2 S.
8(r', t') 2 C s'.
9(r, t) 2 A s. (t, t') 2 S Æ (r, r') 2 R
```
State Monad Refinement

• Forward Simulation

\[
\begin{align*}
\text{corres } & S \ R \ P \ P' \ A \ C == \\
8(s, s') & 2 S. \ P \ s \ AE \ P' \ s' \rightarrow \\
9(r', t') & 2 C \ s'.
\end{align*}
\]
A Small Refinement Calculus

\[
\begin{align*}
\text{corres } S & \ R & P & \ P' & A & \text{ fail} \\
(x, y) & \ 2 & R \\
\text{corres } S & \ R & P & \ P' & (\text{return } x) & (\text{return } y) \\
\text{corres } S & \ R & P & \ P' & (f \gg= g) & (f' \gg= g')
\end{align*}
\]
A Small Refinement Calculus

\[
\text{corres } S \ R \ P \ P' \ \text{A fail}
\]

\[
(x, y) \ 2 \ R
\]

\[
\text{corres } S \ R \ P \ P' \ (\text{return } x) \ (\text{return } y)
\]

\[
\text{corres } S \ R' \ P \ P' \ f \ f'
\]

\[
8x \ y. \ (x, y) \ 2 \ R' \rightarrow \text{corres } S \ R \ (Q \ x) \ (Q' \ y) \ (g \ x) \ (g' \ y)
\]

\[
\{P\} \ f \ \{Q\}
\]

\[
\{P'\} \ f' \ \{Q'\}
\]

\[
\text{corres } S \ R \ P \ P' \ (f \gg= \ g) \ (f' \gg= \ g')
\]
Summary

- Monadic style supports Refinement and Hoare Logic nicely
  - get, put, modify, select, or, assert, when, if, case, etc analogous

- Statistics:
  - 3.5kloc abstract, 7kloc concrete spec (about 3k Haskell)
  - 35kloc proof so far (estm. 50kloc final, about 10kloc/month)
  - 22 patches to Haskell kernel, 90 to abstract spec

- Invariants:
  - well typed references, aligned objects
  - thread states and endpoint queues
  - well formed current thread, scheduler queues

- Failure refines everything -> separate proof of "does not fail"
Thank You