Specification & Refinement in OS Verification

Gerwin Klein
1 microkernel

8,700 lines of C

0 bugs*

qed

*conditions apply
An exception 06 has occurred at 0028:C11B3ADC in \x\d DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in \x\d 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
Small trustworthy foundation

- hypervisor, microkernel, nano-kernel, virtual machine, separation kernel, exokernel ...

- High assurance components in presence of other components

seL4 API:
- IPC
- Threads
- VM
- IRQ
- Capabilities

Untrusted: Legacy Apps, Linux Server

Trusted: Sensitive App, Trusted Service

Hardware
Small Kernels

Small trustworthy foundation

- hypervisor, microkernel, nano-kernel, virtual machine, separation kernel, exokernel ...

- High assurance components in presence of other components

seL4 API:
- IPC
- Threads
- VM
- IRQ
- Capabilities
The Proof
The Proof
Functional Correctness

Proof

Specification

Code
## Functional Correctness

### What

### Proof

#### Specification

**Definition**

```
schedule :: unit s_monad where
schedule ≡ do
  threads ← allActiveTCBs;
  thread ← select threads;
  switch_to_thread thread
  od
  OR switch_to_idle_thread
```

#### Code

```
constdefs
  switch_to_thread :: thread_ptr ⇒ unit s_monad
  switch_to_thread t
  ≡
  do
    state ← get;
    assert yget_tcb t state = None;
    arch_switch_to_thread t;
    modify y λ s. s (| cur_thread := t |)
  od

constdefs
  switch_to_idle_thread :: unit s_monad
  switch_to_idle_thread
  ≡
  do
    thread ← gets idle_thread;
    arch_switch_to_idle_thread;
    modify y λ s. s (| cur_thread := thread |)
  od

definition
  schedule :: unit s_monad
  where
  schedule
  ≡
  do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
  OR switch_to_idle_thread
```
## Functional Correctness

### What

**Definition**

```
definition schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
OR switch_to_idle_thread
end
```

### Proof

**Specification**

```
constdefs
  switch_to_thread :: thread_ptr ⇒ unit s_monad
  switch_to_thread t ≡
    do
      state ← get;
      assert yget_tcb t state ≟ None;
      arch_switch_to_thread t;
      modify y λ s.
        s (| cur_thread := t |)
    od

constdefs
  switch_to_idle_thread :: unit s_monad
  switch_to_idle_thread ≡
    do
      thread ← gets idle_thread;
      arch_switch_to_idle_thread;
      modify y λ s.
        s (| cur_thread := thread |)
    od

definition
  schedule :: unit s_monad
  where
    schedule ≡ do
      threads ← allActiveTCBs;
      thread ← select threads;
      switch_to_thread thread
    od
OR switch_to_idle_thread
end
```

### How

**Code**

```
void schedule(void) {
  switch ((word_t)ksSchedulerAction) {
    case (word_t)SchedulerAction_ResumeCurrentThread:
      break;
    case (word_t)SchedulerAction_ChooseNewThread:
      chooseThread();
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;
    default: /* SwitchToThread */
      switchToThread(ksSchedulerAction);
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;
  }
}

void chooseThread(void) {
  prio_t prio;
  tcb_t *thread, *next;
```
*conditions apply
*conditions apply
*conditions apply

Assume correct:
- compiler + linker (wrt. C op-sem)
- assembly code (600 loc)
- hardware (ARMv6)
- cache and TLB management
- boot code (1,200 loc)
Implications

Execution always defined:
- no null pointer de-reference
- no buffer overflows
- no code injection
- no memory leaks/out of kernel memory
- no div by zero, no undefined shift
- no undefined execution
- no infinite loops/recursion

Not implied:
- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis
Implications

Execution always defined:

• no null pointer de-reference
• no buffer overflows
• no code injection
• no memory leaks/out of kernel memory
• no div by zero, no undefined shift
• no undefined execution
• no infinite loops/recursion

Not implied:

• “secure” (define secure)
• zero bugs from expectation to physical world
• covert channel analysis
Implications

Execution always defined:

- no null pointer de-reference
- no buffer overflows

Not implied:

- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis
Implications

Execution always defined:
• no null pointer de-reference
• no buffer overflows
• no code injection
• no memory leaks/out of kernel memory
• no div by zero, no undefined shift
• no undefined execution
• no infinite loops/recursion

Not implied:
• “secure” (define secure)
• zero bugs from expectation to physical world
• covert channel analysis
Proof Architecture

- Specification
- Proof
- C Code
Proof Architecture

Specification

Design

C Code
Proof Architecture

Access Control Spec

Specification

Design

C Code

Confinement
Proof Architecture

Access Control Spec

Specification

Design

C Code

Confinement

Haskell Prototype
definition
schedule :: unit s_monad where
schedule ≡ do
  threads ← allActiveTCBs;
  thread ← select threads;
  switch_to_thread thread
od
OR switch_to_idle_thread
Proof Architecture

Access Control Spec

Specification

Design

C Code

Confinement

C Code

```c
void schedule(void) {
    switch ((word_t)ksSchedulerAction) {
    case (word_t)SchedulerAction_ResumCurrentThread:
        break;

    case (word_t)SchedulerAction_ChooNewThread:
        chooseThread();
        ksSchedulerAction = SchedulerAction_ResumCurrentThread;
        break;

    default: /* SwitchToThread */
        switchToThread(ksSchedulerAction);
        ksSchedulerAction = SchedulerAction_ResumCurrentThread;
        break;
    }
}

void chooseThread(void) {
    prio_t prio;
    tcb_t *thread, *next;
```
System Model

States:
- User, Kernel, Idle

Events:
- Syscall, Exception, IRQ, VM Fault

States diagram:
- User (U)
- Kernel (K)
- Idle (I)

Events:
- kernel exit
- event
- idle event
- idle

Kernel mode
System Model

States:
User, Kernel, Idle

Events:
Syscall, Exception, IRQ, VM Fault

Diagram:
- States: User (U), Idle (I), Kernel (K)
- Events: kernel exit, idle event, event
- Transition: from User to Kernel, from Idle to Kernel, from Kernel to Idle, from Kernel to User
- Kernel mode

NICTA Copyright 2010
What is Isabelle?

• A generic interactive proof assistant

  – **generic:**
    not specialised to one particular logic (HOL, FOL, ZF)

  – **interactive:**
    not fully automatic, more power to the user
    can guide the system

  – **proof assistant:**
    helps to explore, find, maintain proofs
    checks proofs
Isabelle/HOL

• **Isabelle/HOL:**
  – Higher Order Logic with Simple Types
  – Notation like functional programming and maths.

• **Functions and Types:**
  – \( f \ x \) = “function \( f \) applied to \( x \)”
  – \( x::\text{nat} \) = “\( x \) has type \( \text{nat} \)”
  – \( f :: \text{nat} \Rightarrow \text{bool} \) = “\( f \) has type function from \( \text{nat} \) to \( \text{bool} \)”

• **Theorems**
  – “⟦ \( P \ x; \ Q \ x \ y \ ⟧ \Rightarrow P \ y \”
  – from assumptions “\( P \ x \)” and “\( Q \ x \ y \)” follows conclusion “\( P \ y \)”
OS Specification
What’s special about Operating Systems?

- low level
- very state based
- memory, registers, devices
- reasonably complex computation
- data types (lists, trees)
- implemented in ugly languages like C
Goal

• What to specify?
  – functional behaviour
  – temporal behaviour (general, wcet)
  – security/information flow
  – resource usage

• Here:
  – functional specification
System Model

States:
- User, Kernel, Idle

Events:
- Syscall, Exception, IRQ
State Space

• **Example: Very Simple Kernel**
  – Threads, IPC, Memory, no access control

• **State Space:**
  – current thread
  – memory
  – machine state

• **Memory Objects:**
  – thread control blocks
  – IPC endpoints
  – user pages
Formally

• State = Record

record state =
    cur_thread :: addr
    mstate     :: machine_state
    kheap      :: "addr ⇒ object option"

types addr = "32 word"

datatype object =
    TCB tcb | Endpoint endpoint | UserPage page

types page = "12 word ⇒ 8 word"

datatype endpoint =
    IdleEP | RecvEP "addr list" | SendEP "addr list"
TCBs

record tcb =
  tcb_state :: thread_state
  tcb_context :: user_context
  tcb_ipcframe :: addr

types user_context = "register ⇒ 32 word"

datatype thread_state
  = Running
  | Inactive
  | BlockedOnReceive addr
  | BlockedOnSend addr
Behaviour

• What does the kernel do on this state?

• Outside interface:
  – events come in
  – kernel + user state changes
  – some user thread is activated and runs

• Events:
  – System calls:
    send/recv IPC, start/stop Thread, map/unmap Page
  – Exceptions/Faults:
    illegal read/write, div by zero, etc
Behaviour, formally

datatype event =
    SyscallEvent "32 word"
    | FaultEvent "32 word" "32 word"
    | ExceptionEvent "32 word" "32 word"

kernel :: event ⇒ state ⇒ state
Toplevel function

\[ \text{kernel} :: \text{event} \Rightarrow \text{state} \Rightarrow \text{state} \]

**In something imperative:**

```c
void kernel(event ev) {
    invocation i;
    uint e, thread;

    i = decode(ev);
    e = handle_invocation(i);
    if (e != -1) send_error(e);
    thread = schedule();
    activate(thread);
}
```
Toplevel function

kernel :: event ⇒ state ⇒ state

Functional, first try:

decode :: event ⇒ state ⇒ invocation
handle_invocation :: invocation ⇒ state ⇒ (error * state)
send_error :: error ⇒ state ⇒ state
schedule :: state ⇒ (addr * state)
activate :: addr ⇒ state ⇒ state

void kernel(event ev) {
  invocation i;
  uint e, thread;

  i = decode(ev);
  e = handle_invocation(i);
  if (e != -1) send_error(e);
  thread = schedule();
  activate(thread);
}
Toplevel function

```
kernel :: event ⇒ state ⇒ state

Functional, first try:

kernel ev s =
  let i = decode ev s;
  (e,s1) = handle_invocation i s;
  s2 = if e ≠ None then send_error e s1 else s1;
  (thread,s3) = schedule s2;
  s4 = activate thread s3
  in s4

void kernel(event ev) {
  invocation i;
  uint e, thread;

  i = decode(ev);
  e = handle_invocation(i);
  if (e != -1) send_error(e);
  thread = schedule();
  activate(thread);
}
Not Good

kernel :: event ⇒ state ⇒ state

Functional, first try:

- formal
- but: fairly horrible, hard to read
Composition

kernel :: event ⇒ state ⇒ state

Functional, first try:

- would be nicer with function composition
- no explicit state weaving

- if everything was of type ... ⇒ state ⇒ state

activate thread
schedule o
(if e then send_error e else id) o
handle_invocation i o
decode
Functional, first try:

- would be nicer with function composition
- no explicit state weaving
- if everything was of type ... ⇒ state ⇒ state

```plaintext
decode;
handle_invocation i;
if e then send_error e else id;
schedule;
activate thread
```
kernel :: event ⇒ state ⇒ state

Functional, first try:

- would be nicer with function composition
- no explicit state weaving

- if everything was of type ... ⇒ state ⇒ state

decode;
handle_invocation i;
if e then send_error e else id;
schedule;
activate thread

- but where do i, e, thread come from?
Taking A Step Back

• More general composition operator
  – need state change and return value

• State Monad
  – ('a,'s) monad = 's ⇒ ('a * 's)
  – can express all our functions:

```haskell
state ⇒ state: 'a = unit, 's = state
state ⇒ result: 'a = result, 's = state
```
Composition

• State Monad
  - (\texttt{\texttt{a}}, \texttt{\texttt{s}}) monad = \texttt{\texttt{s}} \Rightarrow (\texttt{\texttt{a}} \times \texttt{\texttt{s}})

• The bind operator
  - \texttt{bind} :: (\texttt{\texttt{a}}, \texttt{\texttt{s}}) monad \Rightarrow
    (\texttt{\texttt{a}} \Rightarrow (\texttt{\texttt{b}}, \texttt{\texttt{s}}) monad) \Rightarrow
    (\texttt{\texttt{b}}, \texttt{\texttt{s}}) monad
  - \texttt{bind f g} also written \texttt{f >>= g}
  - Execute \texttt{f}, then execute \texttt{g} with result of \texttt{f}
Composition

• Definition

\[(f >>= g) s = let (r,s') = f s \quad\text{in} \ g r s'\]

• Example

\[
\text{kernel ev =} \\
\quad \text{decode ev } >>= \\quad \text{(handle_invocation } >>= \ \\
\text{ (fn e. when (e } \neq \text{ None) (send_error e } >>= \ \\
\text{ (fn _ . schedule } >>= \ \\
\text{ activate))})
\]
Syntax

• do - Syntax:
  – notation for: \( f >>= (\text{fn } x. \ g \ x) \)

\[
\text{do} \\
x \leftarrow f; \\
g \ x \\
\text{od}
\]

• Example:

\[
\text{kernel ev = do} \\
i \leftarrow \text{decode ev}; \\
e \leftarrow \text{handle_invocation i}; \\
\text{when (e \neq None) (send_error e)}; \\
\text{thread} \leftarrow \text{schedule}; \\
\text{activate thread} \\
\text{od}
\]
Basic Monads

- **Basic**
  
  bind
  
  return :: ‘a ⇒ (‘a,’s) monad
  
  return x = fn s. (x,s)

- **State**
  
  put :: ‘s ⇒ (unit, ‘s) monad
  
  put s' = fn s. ((), s')

  get :: (‘s,’s) monad
  
  get = fn s. (s,s)

  gets :: (‘s ⇒ ‘a) ⇒ (‘a,’s) monad
  
  gets f = do s ← get;
              return (f s)
         od
More Basic Monads

• Building blocks

modify :: (‘s ⇒ ‘s) ⇒ (unit, ‘s) monad
modify f = do s ← get;
           put (f s)
           od

when :: bool ⇒ (unit,’s) monad ⇒ (unit,’s) monad
when P f = if P
           then f
           else return ()
Exercise

• Write some basic state manipulation functions

get_object :: addr ⇒ (object, state) monad
set_object :: ?

set_cur_thread :: addr ⇒ (unit, state) monad

get_tcb_state :: addr ⇒ (thread_state, state) monad
set_tcb_state :: ?
Problem

• How to write

\[
\text{schedule} :: (\text{thread, state}) \text{ monad}
\]

Would like:

\[
\text{active_threads} :: \text{state} \Rightarrow \text{addr set}
\]

\[
\text{active_threads } s = \{ t. \text{kheap } s \text{ } t = \text{Some (TCB } tcb) \land \text{ thread_state } tcb = \text{Running}\}
\]

\[
\text{schedule } = \text{ do }
\]
\[
\text{threads } \leftarrow \text{gets active_threads}
\]
\[
\text{t } \leftarrow \text{“select” threads}
\]
\[
\text{return } t
\]
Nondeterminism

- Produce set of states, not just one:

\[ ('a,'s) \text{ n\_monad} = 's \Rightarrow ('a \times 's) \text{ set} \]

- Bind:

\[
(f >>=g) s = \text{ let } S = f s; \\
S' = (\text{fn } (r,s'). g r s') \setminus S \\
in \text{ Union } S'
\]

- Return

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

• Write

put, get

• Write

\[
\text{alt} :: ('a,'s) \text{n Monad} \Rightarrow ('a,'s) \text{n Monad} \Rightarrow ('a,'s) \text{n Monad}
\]
Nondeterminism

• Choice

select :: 'a set ⇒ ('a,'s) n_monad

select S = fn s. S x {s}

• Assertion

assert :: bool ⇒ (unit,'s) n_monad

assert P = if P
  then return ()
  else chaos

chaos = fn s. UNIV
Hoare Logic
Hoare Logic
Reasoning

• Can express behaviour

• What now?
  – Proving properties about the kernel
  – Invariants
  – Functional properties

• For instance:
  – “cur_thread always points to a TCB”
  – “only running threads are scheduled”
  – “spec S’ refines S”
Reasoning

• Reasoning about state-based programs:
  – Hoare Logic
  – pre/post conditions

• On State Monads:
  – need some adjustments
Hoare Logic for State Monad

• Hoare Triple

\{P\} f \{Q\}

\(f :: (\text{'a,}'s) \text{n_monad}\)

\(P :: \text{'s} \Rightarrow \text{bool}\)

\(Q :: ?\)
Hoare Logic for State Monad

• Hoare Triple

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

\[ f :: (\text{'a,}'s) \; \text{n\_monad} \]

\[ P :: \text{'s} \Rightarrow \text{bool} \]

\[ Q :: \text{'a} \Rightarrow \text{'s} \Rightarrow \text{bool} \]

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

\[ \forall s. \; P \; s \rightarrow (\forall (r,s') \in f \; s. \; Q \; r \; s') \]
Hoare Logic for State Monad

• Rules

bind rule like semicolon rule in imperative languages

\[
\begin{align*}
\{P\} & \ f \ \{Q\} \quad \forall x. \ \{Q \ x\} \ g \ x \ \{R\} \\
\hline
\{P\} & \ f \ >>=} \ g \ \{R\}
\end{align*}
\]
Hoare Logic for State Monad

• **Weakest Precondition Style**
  – for postcondition $P$ and $f$
  – find weakest precondition $P'$ such that $f$ establishes $P$

• **Example**

\[
\{P \; x\} \quad \text{return} \; x \quad \{P\}
\]
Exercise

• Find WP Rules for

  – get, gets
  – put, modify
  – select
  – assert
Automation

• Prove rules in Isabelle
  – from definition of pre/post
  – means rules are sound
  – can be used as lemmas and proof rules

• Automate rule application
  – can write tool that tries wp rules until one fits
  – backwards application over bind
  – case distinctions interesting

  – weakest precondition not always practical
Exercise

• Prove (manually)
  – schedule only returns running threads
Refinement
Refinement

• Can reason about specs.

• What now?
  – Given spec S, is implementation I correct?
  – “I refines S”

  – Can express as Hoare triple
  – Reasoning tedious
  – > Design better rules
Definition

- C refines A if
  - all behaviours of C are behaviours of A
  - (eqv) all Hoare-Properties of A are true of C

- Sufficient: forward simulation
State Monad Refinement

- Forward Simulation

\[ \text{corres } S \ R \ A \ C \iff \]
\[ \forall (s, s') \in S. \]
\[ \forall (r', t') \in C \ s'. \]
\[ \exists (r, t) \in A \ s. \ (t, t') \in S \land (r, r') \in R \]
State Monad Refinement

• Forward Simulation

corres \( S \leadsto R \) if

\[
\forall (s, s') \in S. \quad P s \land P' s' \\
\forall (r', t') \in C s'.
\]

\[
\exists (r, t) \in A s. \quad (t, t') \in S \land (r, r') \in R
\]
State Monad Refinement

• As Hoare Triple

∀s. \{ fn s'. P s ∧ P' s' ∧ (s, s') ∈ S \}

C

{ fn r' t'. ∃(r, t) ∈ A s. (t, t') ∈ S ∧ (r, r') ∈ R \}
A Small Refinement Calculus

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

\[
\begin{align*}
\text{corres } S & \quad R & \quad P & \quad P' & \quad \text{(return } x) & \quad \text{(return } y) \\
\end{align*}
\]

\[
\begin{align*}
\text{corres } S & \quad R & \quad P & \quad P' & \quad (f \gg= g) & \quad (f' \gg= g') \\
\end{align*}
\]
A Small Refinement Calculus

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

\[
\begin{align*}
\text{corres } S & \ R \ P \ P' \ (\text{return } x) \ (\text{return } y) \\
\forall \ x \ y. \ (x, y) & \in R' \rightarrow \text{corres } S \ R \ (Q \ x) \ (Q' \ y) \ (g \ x) \ (g' \ y) \\
& \{P\} \ f \ \{Q\} \\
& \{P'\} \ f' \ \{Q'\} \\
\end{align*}
\]

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

• Write rules for
  – get
  – when
Demo

- Example application
  - more concrete design
  - schedule with priorities

- prove

corres S Univ T T schedule schedule’
Summary
Summary

- **OS kernel specifications**
  - OS = low-level, stateful
  - clear notation with monads
  - explicit state for reasoning

- **Monad Reasoning with Hoare-Logic**
  - weakest useful preconditions
  - automation

- **Refinement**
  - exploit structure
Thank You
Thank You