Verifying the seL4 Microkernel

Formal Proof in Mathematics and Computer Science

Lukas Stevens

21st June 2018

Outline

1. What is a μ-kernel?
2. Design process of seL4
3. Formal methods of the correctness proof
4. Layers of the correctness proof
5. Conclusion

What is a μ-kernel?

What is a kernel anyway?

What is a kernel anyway?

- Necessary abstractions for applications

What is a kernel anyway?

- Necessary abstractions for applications
- Interaction via system calls

What is a kernel anyway?

- Necessary abstractions for applications
- Interaction via system calls
- Loaded into protected memory region

What is a kernel anyway?

- Necessary abstractions for applications
- Interaction via system calls
- Loaded into protected memory region

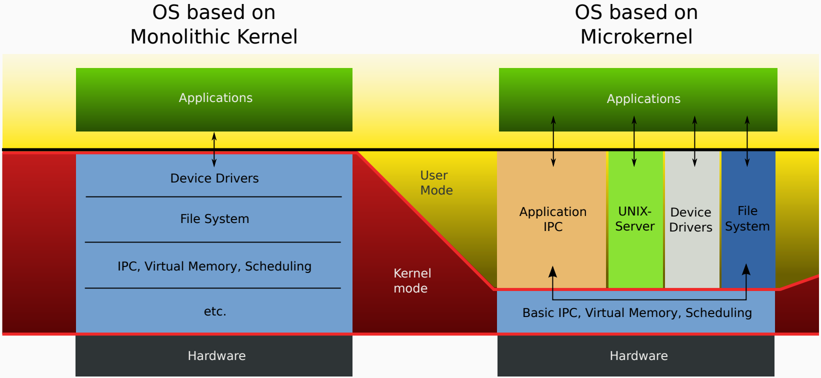
$\Rightarrow$ **Bugs are potentially fatal**

Definition: Microkernel

A concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

— Jochen Liedtke

Monolithic kernels and μ-kernels



The seL4 μ-kernel

The seL4 μ-kernel

- Member of the L4 μ-kernel family

The seL4 μ-kernel

- Member of the L4 μ-kernel family
- Correctness verified with Isabelle

The seL4 μ-kernel

- Member of the L4 μ-kernel family
- Correctness verified with Isabelle
- High performance

Design process of seL4

Design process for verification

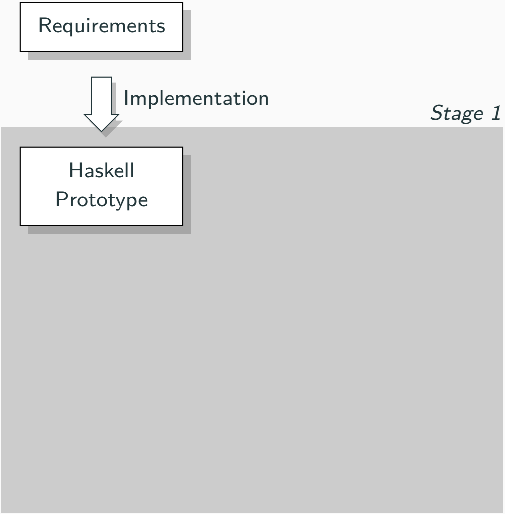
Requirements

Design process for verification

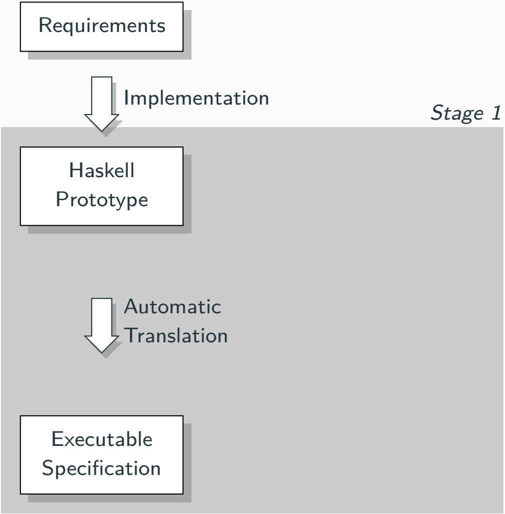
Requirements

Stage 1

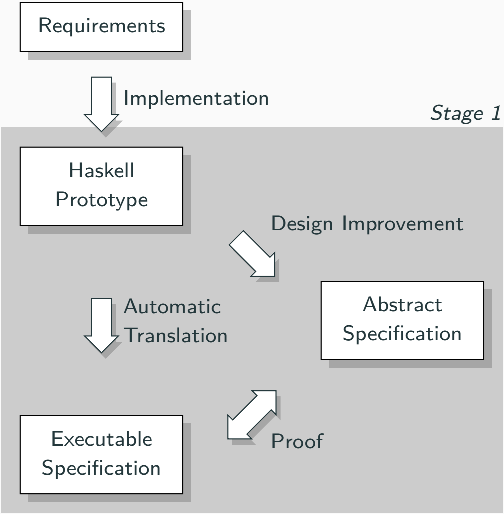
Design process for verification



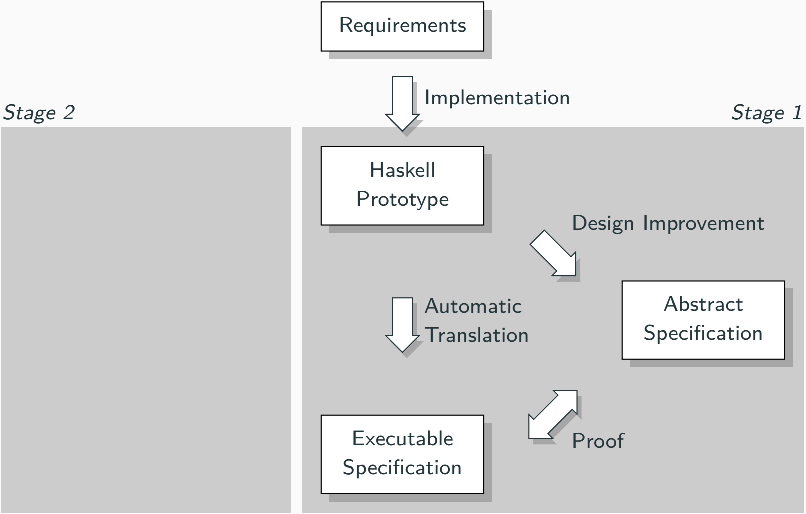
Design process for verification



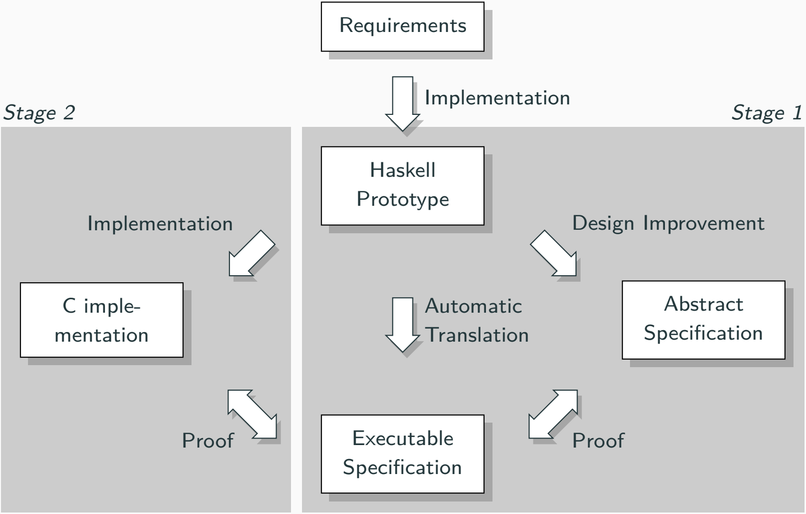
Design process for verification



Design process for verification



Design process for verification



Formal methods of the correctness proof

Hoare logic

$$\overbrace{\{x = 1\}}^{P} \quad \overbrace{x := x + 1}^{C} \quad \overbrace{\{x = 2\}}^{Q}$$

More Hoare logic

$$\{x = 0 \land x = 1\} \quad y := 2 * x \quad \{ \qquad\qquad\qquad\qquad \}$$

More Hoare logic

$$\{x \text{ is even}\} \quad y := 2 * x \quad \{ \qquad\qquad\qquad\qquad \}$$

More Hoare logic

$$\{x \text{ is even}\} \quad y := 2 * x \quad \{x \text{ and } y \text{ are even}\}$$

Partial correctness of Hoare logic

$\{\quad\}$ `WHILE true DO c` $\{\quad\}$

Data refinement

Data refinement

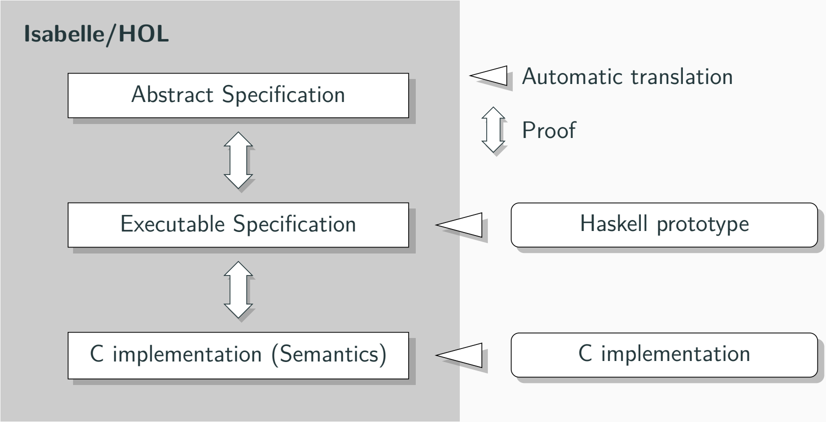
A concrete system C refines an abstract specification A if the behaviour of C is contained in that of A.

Data refinement: Examples

- The scheduler selects runnable threads
- System calls return non-zero values on error

Layers of the correctness proof

Proof structure



Abstract specification

The abstract specification is the most high-level layer still fully encapturing the behaviour of the kernel.

Scheduler on the abstract level

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

Executable specification

Fill in the details left open by the abstract specification.

Haskell implementation of the scheduler

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

chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
        tcbSchedDequeue thread
        return False
    else do
        switchToThread thread
        return True
```

Haskell implementation of the scheduler

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

chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
        tcbSchedDequeue thread
        return False
    else do
        switchToThread thread
        return True
```

Call chooseThread to select next thread.

Haskell implementation of the scheduler

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

chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
        tcbSchedDequeue thread
        return False
    else do
        switchToThread thread
        return True
```

Call chooseThread to select next thread.

Get runnable thread with highest priority using `chooseThread'` or schedule idle thread.

Haskell implementation of the scheduler

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

chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
        tcbSchedDequeue thread
        return False
    else do
        switchToThread thread
        return True
```

Call chooseThread to select next thread.

Get runnable thread with highest priority using `chooseThread'` or schedule idle thread.

Try to find runnable thread in Queue.

Haskell implementation of the scheduler

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

chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread

chooseThread' prio = do
    q <- getQueue prio
    liftM isJust $ findM chooseThread'' q

chooseThread'' thread = do
    runnable <- isRunnable thread
    if not runnable then do
        tcbSchedDequeue thread
        return False
    else do
        switchToThread thread
        return True
```

Call chooseThread to select next thread.

Get runnable thread with highest priority using `chooseThread'` or schedule idle thread.

Try to find runnable thread in Queue.

Check if thread is runnable and act accordingly.

C implementation

Translate the Haskell implementation to C.

Machine Model

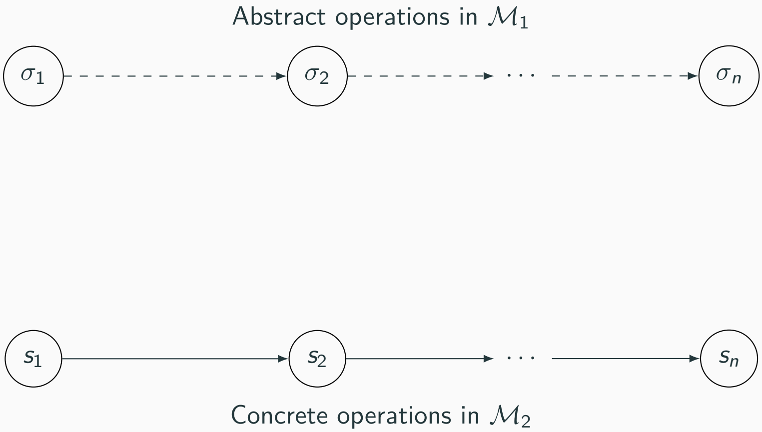
```
invalidateTLB :: unit machine_m => unit machine_m

invalidateCacheRange ::
    unit machine_m => word => word => unit machine_m
```

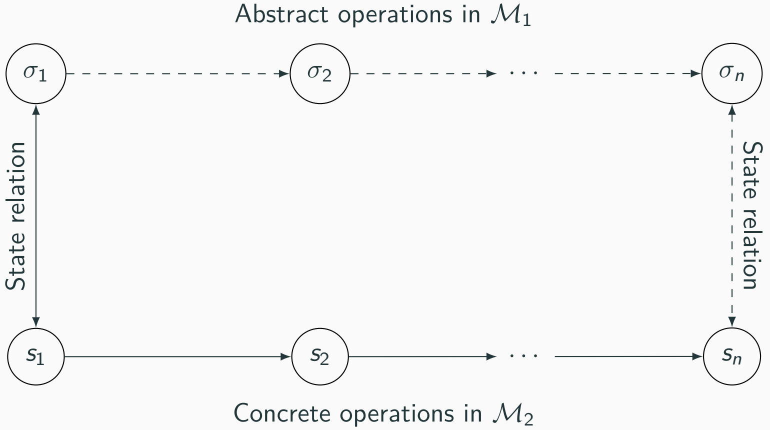
Data refinement for state machines

Concrete operations in $\mathcal{M}_2$

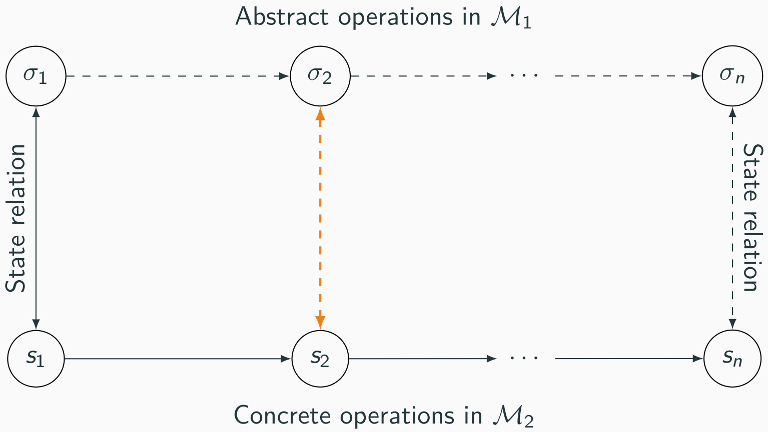
Data refinement for state machines



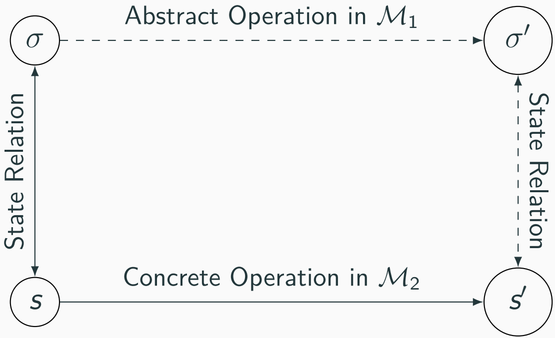
Data refinement for state machines



Data refinement for state machines



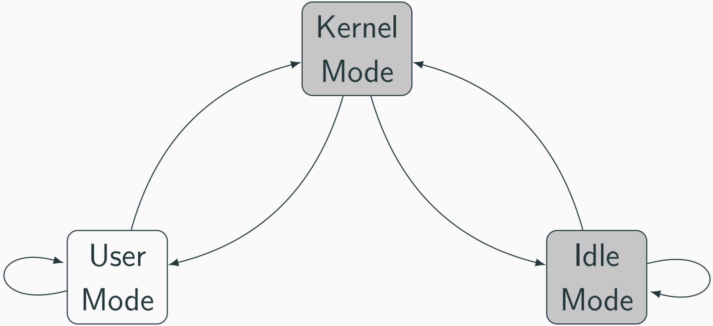
Refinement by forward simulation



Example for forward simulation

On the Board

Types of state transitions



Main result

$$\mathcal{M}_A$$

$$\mathcal{M}_E$$

$$\mathcal{M}_C$$

Main result

$$\begin{tikzcd} \mathcal{M}_A \\ \mathcal{M}_E \arrow[u, "\text{refines}"'] \\ \mathcal{M}_C \arrow[u, "\text{refines}"'] \end{tikzcd}$$

Main result

$$
\begin{tikzcd}
\mathcal{M}_A \\
\mathcal{M}_E \arrow[u, "\text{refines}"'] \\
\mathcal{M}_C \arrow[u, "\text{refines}"'] \arrow[uu, bend left=60, "\text{refines}"]
\end{tikzcd}
$$

Conclusion

Expenditure of time

Artefact	Effort (py)	Total (py)
Haskell impl.	2.0	2.2
C impl.	0.2	
Generic framework	9.0	20.5
Abstract spec.	0.3	
Executable spec.	0.2	
Refinement $\mathcal{M}_A \leftrightarrow \mathcal{M}_E$	8.0	
Refinement $\mathcal{M}_E \leftrightarrow \mathcal{M}_C$	3.0	

How does the effort compare?

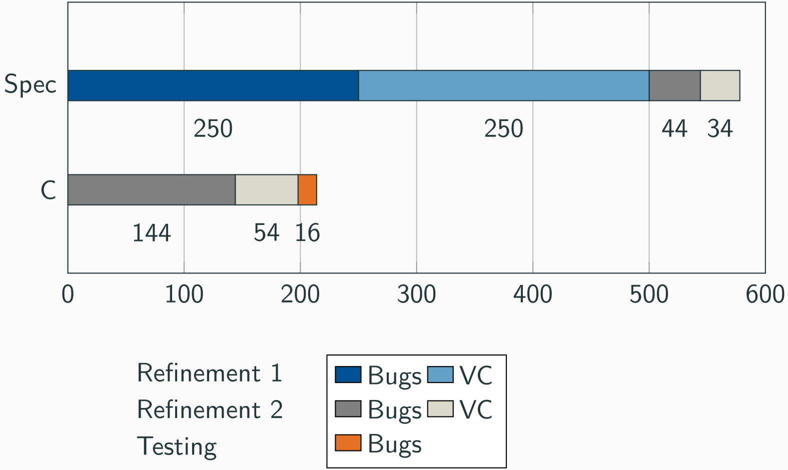
How does the effort compare?

- EAL7: 1000\$/LOC $\leftrightarrow$ seL4: 370\$/LOC

How does the effort compare?

- EAL7: 1000\$/LOC $\leftrightarrow$ seL4: 370\$/LOC
- L4 Pistachio kernel: 6 py $\leftrightarrow$ seL4 kernel: 2.2 py

Changes due to verification



What was achieved?

- Correctness proof down to binary level
- Trust in hardware

What was achieved?

- Correctness proof down to binary level
- Trust in hardware
- **What about Spectre and Meltdown?**

The future of seL4

- More architectures
- Multicore support

The future of seL4

- More architectures
- Multicore support
- Exclude timing-channel attacks

Questions?