1. What is a $\mu$-kernel?

2. Design process of seL4

3. Formal methods of the correctness proof

4. Layers of the correctness proof

5. Conclusion
What is a $\mu$-kernel?
What is a kernel anyway?

• Necessary abstractions for applications
• Interaction via system calls
• Loaded into protected memory region
• Bugs are potentially fatal
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

⇒ Bugs are potentially fatal
A concept is tolerated inside the $\mu$-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 $\mu$-kernels

OS based on Monolithic Kernel

- Applications
- Device Drivers
- File System
- IPC, Virtual Memory, Scheduling
- etc.
- Hardware

OS based on Microkernel

- Applications
- Application IPC
- UNIX-Server
- Device Drivers
- File System
- Basic IPC, Virtual Memory, Scheduling
- Hardware
The seL4 $\mu$-kernel

- Member of the L4-kernel family
- Correctness verified with Isabelle
- High performance
The seL4 $\mu$-kernel

- Member of the L4 $\mu$-kernel family
The seL4 $\mu$-kernel

- Member of the L4 $\mu$-kernel family
- Correctness verified with Isabelle
The seL4 $\mu$-kernel

- Member of the L4 $\mu$-kernel family
- Correctness verified with Isabelle
- High performance
Design process of seL4
Design process for verification
Design process for verification

Stage 1

Requirements
Design process for verification

Stage 1

Requirements

Implementation

Haskell Prototype

Stage 1

6
Design process for verification

- Requirements
- Implementation
  - Haskell Prototype
  - Automatic Translation
  - Executable Specification

Stage 1
Design process for verification

Stage 1

Requirements

Implementation

Haskell Prototype

Design Improvement

Automatic Translation

Executable Specification

Proof

Abstract Specification
Design process for verification

Stage 1

- Abstract Specification
- Design Improvement
- Proof
- Automatic Translation
- Haskell Prototype
- Implementation
- Requirements

Stage 2
Design process for verification

Stage 1:
- Requirements
  - Implementation
  - Haskell Prototype
    - Automatic Translation
    - Abstract Specification
    - Design Improvement

Stage 2:
- Implementation
  - C Implementation
    - Proof
  - Executable Specification
Formal methods of the correctness proof
Hoare logic

\[
P \begin{cases} x = 1 \end{cases} \quad C \begin{cases} x := x + 1 \end{cases} \quad Q \begin{cases} x = 2 \end{cases}
\]
More Hoare logic

\{ x = 0 \land x = 1 \} \quad y \leftarrow 2 \times x \quad \{ \}
More Hoare logic

\{ x \text{ is even} \}
\begin{align*}
y & := 2 \times x \\
\end{align*}
More Hoare logic

\{ x \text{ is even} \} \quad y := 2 \times x \quad \{ x \text{ and } y \text{ are even} \}
Partial correctness of Hoare logic

{ } WHILE true DO c { }
Data refinement

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

10
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

Isabelle/HOL

Abstract Specification

Executable Specification

C implementation (Semantics)

Automatic translation

Proof

Haskell prototype

C implementation
The abstract specification is the most high-level layer still fully encapsulating the behaviour of the kernel.
Scheduler on the abstract level

\[
\text{schedule} \equiv \begin{array}{l}
\text{do} \\
\text{threads} \leftarrow \text{all_active_tcbs}; \\
\text{thread} \leftarrow \text{select threads}; \\
\text{switch_to_thread thread} \\
\text{od OR switch_to_idle_thread}
\end{array}
\]
Executable specification

Fill in the details left open by the abstract specification.
Haskell implementation of the scheduler

```haskell
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

\[
schedule = \text{do}
\quad \text{action} \leftarrow \text{getSchedulerAction}
\quad \text{case} \ \text{action} \ \text{of}
\quad \quad \text{ChooseNewThread} \rightarrow \text{do}
\quad \quad \quad \text{chooseThread}
\quad \quad \quad \text{setSchedulerAction} \ \text{ResumeCurrentThread}
\quad \quad \ldots
\]

\[
\text{chooseThread} = \text{do}
\quad r \leftarrow \text{findM} \ \text{chooseThread'} \ \text{(reverse [minBound .. maxBound])}
\quad \text{when} \ (r == \text{Nothing}) \ \text{switchToIdleThread}
\]

\[
\text{chooseThread'} \ \text{prio} = \text{do}
\quad q \leftarrow \text{getQueue} \ \text{prio}
\quad \text{liftM} \ \text{isJust} \ \text{switchToIdleThread}
\]

\[
\text{chooseThread'}' \ \text{thread} = \text{do}
\quad \text{runnable} \leftarrow \text{isRunnable} \ \text{thread}
\quad \text{if not} \ \text{runnable then do}
\quad \quad \text{tcbSchedDequeue} \ \text{thread}
\quad \quad \text{return} \ \text{False}
\quad \text{else do}
\quad \quad \text{switchToThread} \ \text{thread}
\quad \quad \text{return} \ \text{True}
\]

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.
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.
Translate the Haskell implementation to C.
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

Abstract operations in $\mathcal{M}_1$

$\sigma_1 \rightarrow \sigma_2 \rightarrow \cdots \rightarrow \sigma_n$

Concrete operations in $\mathcal{M}_2$

$s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$
Data refinement for state machines

Abstract operations in $M_1$

$s_1$ $\rightarrow$ $s_2$ $\rightarrow$ $\cdots$ $\rightarrow$ $s_n$

Concrete operations in $M_2$

$\sigma_1$ $\rightarrow$ $\sigma_2$ $\rightarrow$ $\cdots$ $\rightarrow$ $\sigma_n$

State relation
Data refinement for state machines

Abstract operations in $M_1$

Concrete operations in $M_2$
Refinement by forward simulation

State Relation

\( \sigma \)  \rightarrow  \text{Concrete Operation in } \mathcal{M}_2  \rightarrow  \text{State Relation}  \rightarrow  \sigma' \n
\sigma \rightarrow  \text{Abstract Operation in } \mathcal{M}_1  \rightarrow  \sigma'  

\( s \)  \rightarrow  \text{Concrete Operation in } \mathcal{M}_2  \rightarrow  \text{State Relation}  \rightarrow  s' \n
\![20]
Example for forward simulation

On the Board
Types of state transitions

- Kernel Mode
- User Mode
- Idle Mode
Main result

\[ \mathcal{M}_A \]

\[ \mathcal{M}_E \]

\[ \mathcal{M}_C \]
Main result

\[ \mathcal{M}_A \]

\[ \mathcal{M}_E \]

\[ \mathcal{M}_C \]

refines

refines

refines
Main result

\[ \mathcal{M}_A \] refines \[ \mathcal{M}_E \] refines \[ \mathcal{M}_C \] refines \[ \mathcal{M}_A \]
Conclusion
## Expenditure of time

<table>
<thead>
<tr>
<th>Artefact</th>
<th>Effort (py)</th>
<th>Total (py)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell impl.</td>
<td>2.0</td>
<td>2.2</td>
</tr>
<tr>
<td>C impl.</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Generic framework</td>
<td>9.0</td>
<td></td>
</tr>
<tr>
<td>Abstract spec.</td>
<td>0.3</td>
<td>20.5</td>
</tr>
<tr>
<td>Executable spec.</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Refinement $\mathcal{M}_A \leftrightarrow \mathcal{M}_E$</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>Refinement $\mathcal{M}_E \leftrightarrow \mathcal{M}_C$</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>
How does the effort compare?

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

- L4 Pistachio kernel: 6 py
- seL4 kernel: 2.2 py
How does the effort compare?

- EAL7: 1000$/LOC ↔ seL4: 370$/LOC
How does the effort compare?

- EAL7: 1000$/LOC ↔ seL4: 370$/LOC
- L4 Pistachio kernel: 6 py ↔ seL4 kernel: 2.2 py
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?