seL4 Microkernel Case Study

CS446 2011

Derek Rayside

What criteria do we use to analyze designs?

Analytical Criteria

- Fitness for Purpose
  - Does the design do what it is supposed to do?
  - correctness, performance, security, usability, etc.

- Fitness for Future
  - Will the design ***adapt*** to changing needs?
  - Can portions of the design be ***reused*** on other projects?

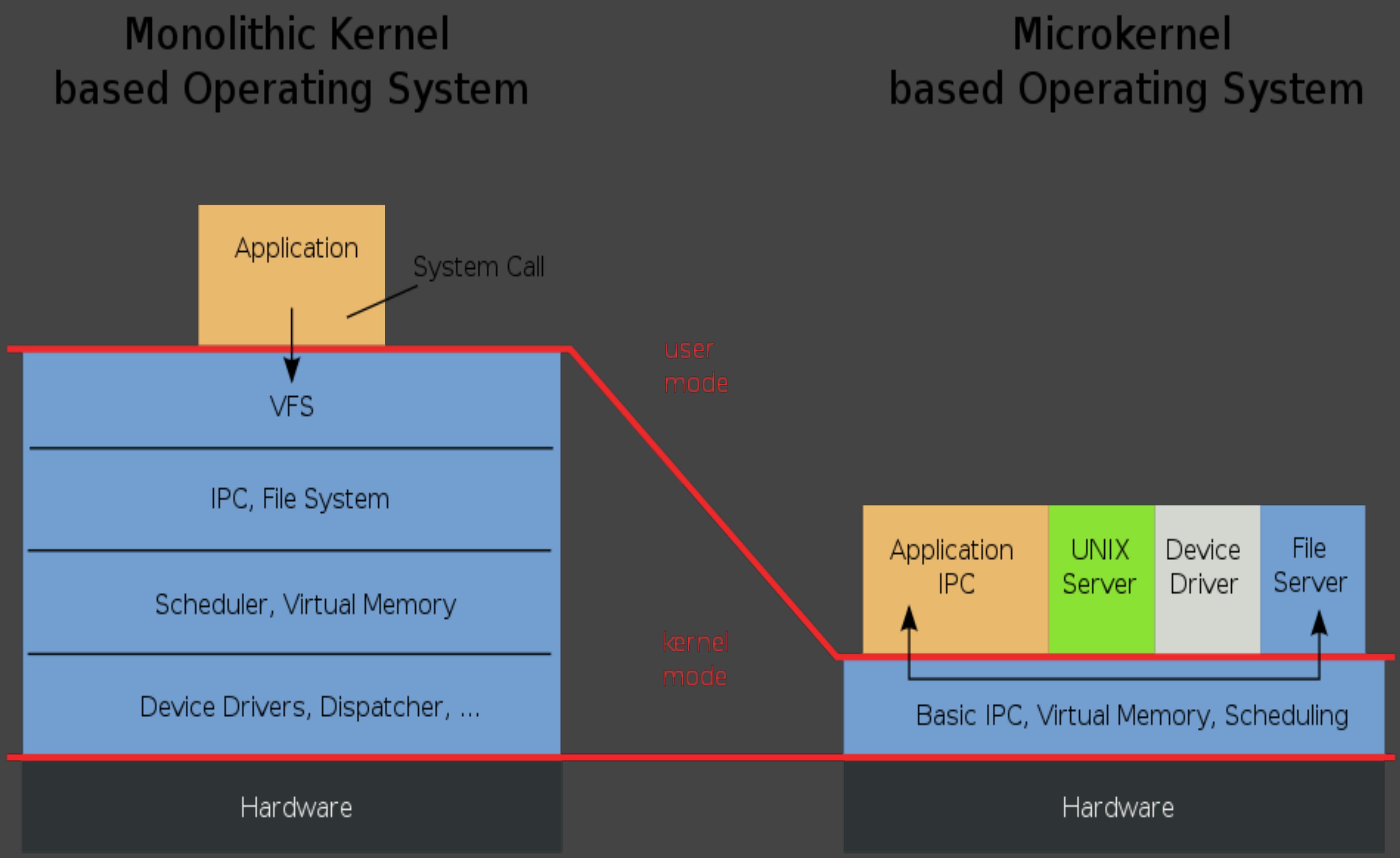
- Cost of Production
  - parts
  - labour (time, skills, parallelism)
  - capital/tools

- Cost of Operation

What are the important criteria for an Operating System?

What are the main reference architectures for Operating Systems?

Monolithic vs Microkernel



Contrasting Readings

Gerwin Klein et alia
seL4: Formal Verification of an Operating-System Kernel
SOSP'09 Best Paper
Reprinted in CACM 2010

- fitness for purpose
- performance

Think about:
- seL4 fitness for future
- seL4 interrupt during system call issue

Richard P. Gabriel
Worse is Better

- fitness for future is necessary for survival
- ITS vs Unix
- what to do when an interrupt occurs during a system call?
  - ITS: handle it
  - Unix: bail

Worse Is Better [Gabriel]

	The New Jersey Style (C/Unix)	*The MIT Approach (Lisp/Scheme/ITS)*
Simplicity	#1 impl. > interface	interface > impl.
Correctness	mostly	100%
Consistency	mostly	100%
Completeness	meh	mostly

Worse Is Better [Gabriel]

	The New Jersey Style (C/Unix)	*The MIT Approach (Lisp/Scheme/ITS)*
Simplicity ***Fitness for Future vs Fitness for Purpose***	#1 impl. > interface	interface > impl.
Correctness ***Fitness for Purpose***	mostly	100%
Consistency ***Fitness for Purpose***	mostly	100%
Completeness ***Fitness for Purpose***	meh	mostly

3 Generations of MicroKernels

1. Mach (1985-1994)
    - replace pipes with IPC (more general)
    - improved stability (vs monolithic kernels)
    - poor performance
2. L3 & L4 (1990-2001)
    - order of magnitude improvement in IPC performance
        - written in assembly
        - sacrificed CPU portability
        - only synchronus IPC (build async on top of sync)
    - very small kernel: more functions moved to userspace
3. seL4, Coyotos, Nova (2000-2010)
    - platform independence
    - verification, security, multiple CPUs, etc.

What does a Mach kernel do?

1. Asynchronus IPC
2. Threads
3. Scheduling
4. Memory management
5. Resource access permissions
6. Device drivers (in some variants)

All other functions are implemented outside the kernel.

API Size: 140 functions

What caused Mach's performance problems?

1. Checking resource access permissions on system calls.
   - Single user machines do not need to do this.
2. Cache misses.
   - Critical sections were too large.

3. Asynchronus IPC
   - Most calls only need synchronus IPC.
   - Synchronous IPC can be faster than asynchronous.
   - Asynchronous IPC can be built on top of synchronous.

4. Virtual memory
   - How to prevent key processes from being paged out?

What does an L4 kernel do?

1. Synchronous IPC
2. Threads
3. Scheduling
4. Memory management

All other functions are implemented outside the kernel.

API Size: 7 functions (vs 140 for Mach3)

Jochen Liedtke's *minimality principle* for L4:

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

cf. Fred Books on *conceptual integrity* [Mythical Man Month]

Conceptual Integrity [F.Brooks]

Unix	Everything is a file.
Mach	IPC generalizes files.
L4	Can it be put outside the kernel?

seL4 MicroKernel

What properties do we expect from a kernel?

Sidebar: Invariants

A property that is true of every state*.

* at least at public method boundaries.

For example, inserting a node into a linked list may cause the list to become temporarily disconnected.

Invariants may need to be verified for every part of the system, not just the parts of the system that obviously manipulate the structure in question.

Desired file synchronizer properties?

Desired file synchronizer properties?

The synchronizer doesn't eat my files.

- partial file updates work correctly
- conflicts are handled in some sane manner
- massive deletes are not propagated without warning

I always have the latest version of my files.

- what about network latency?
    - prioritize file transfers

Synchronization is idempotent.

- $f(f(x)) = f(x)$
- e.g., set union is idempotent

What properties do we expect from a kernel?

What properties do we expect from a kernel?

- Every system call terminates.
- No exceptions thrown.
- No arithmetic problems (e.g., overflow, divide by zero)
- No null pointer de-references.
- No ill-typed pointer de-references.
- No memory leaks.
- No buffer overflows.
- No unchecked user arguments.
- Code injection attacks are impossible.
- Well-formed data structures.
- Correct book-keeping.
- No two objects overlap in memory
- *etc.*

How to design a kernel with these properties?

Very carefully.

Iterative Co-design of Kernel & Proof

Kernel Team

1. Initial prototype (Haskell)
   - no interrupts
   - single address space
   - generic linear page table

2. Complete prototype
   - add missing functionality

3. Implementation

Proof Team

1. Infrastructure

2. Abstract Spec
   - prototype vs spec

3. Spec vs Implementation

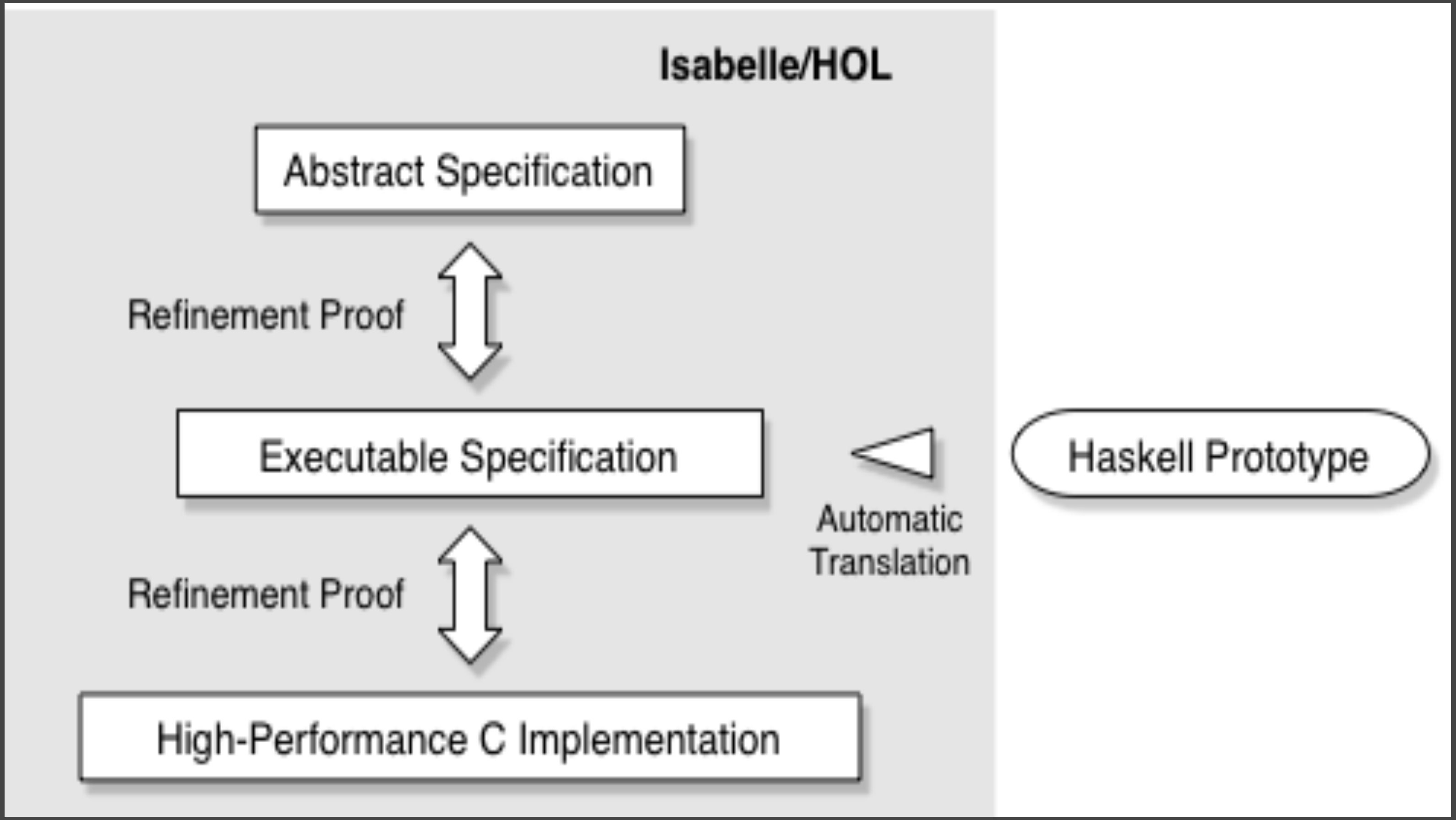
Sidebar: What is a Prototype?

Evolutionary

- eventually becomes the real thing
- advocated by F. Brooks, R. Gabriel, agile, and others ("organic growth")

Experimental

- used to explore an idea, then thrown away
- advocated by F. Brooks ("plan to throw one away")
- may be one of:
  - ***Horizontal***: shallow ptype of whole system
  - ***Vertical***: detailed ptype of single subsys


Isabelle/HOL
Abstract Specification
Refinement Proof
Executable Specification
Automatic
Translation
Haskell Prototype
Refinement Proof
High-Performance C Implementation

Abstract Spec (Isabelle/HOL)

Executable Spec (Haskell)

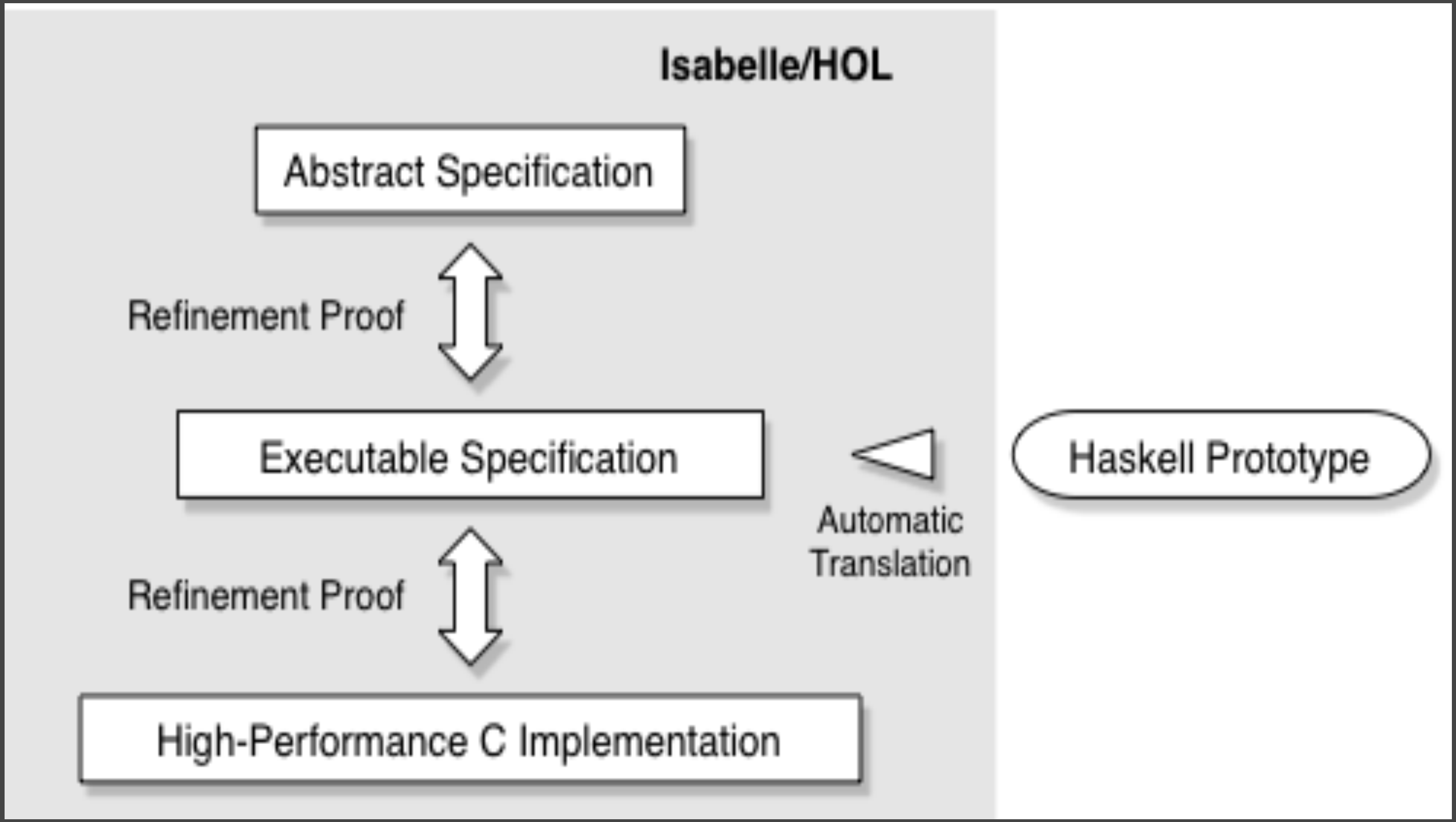
```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

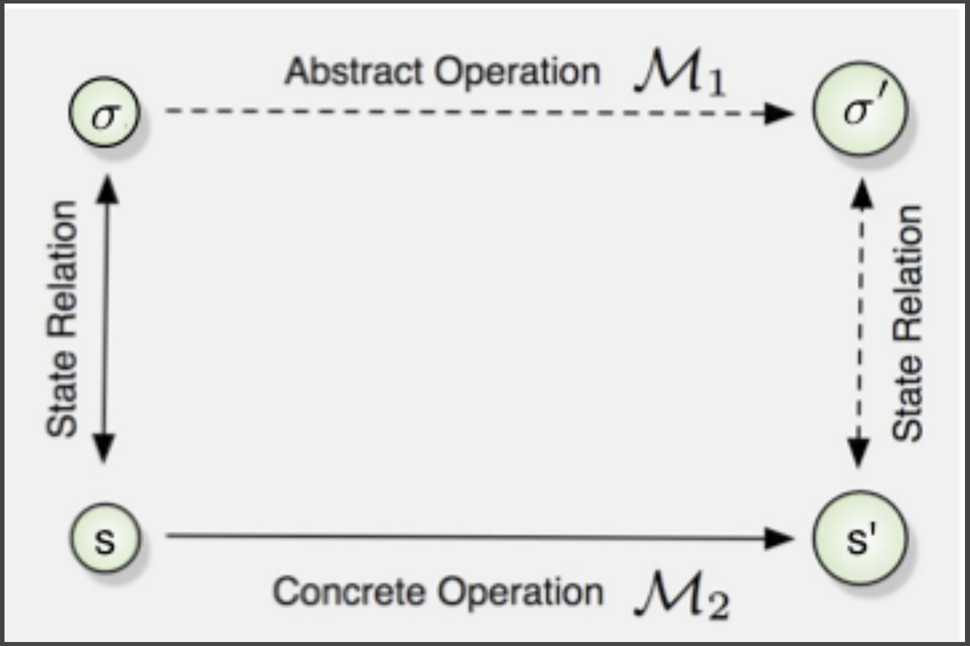
```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread
chooseThread' 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
```

The implementation in C is much larger.

```
void setPriority(tcb_t *tptr, prio_t prio) {
  prio_t oldprio;
  if(thread_state_get_tcbQueued(tptr->tcbState)) {
    oldprio = tptr->tcbPriority;
    ksReadyQueues[oldprio] =
      tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
    if(isRunnable(tptr)) {
      ksReadyQueues[prio] =
        tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
    }
    else {
      thread_state_ptr_set_tcbQueued(&tptr->tcbState,
                                     false);
    }
  }
  tptr->tcbPriority = prio;
}
```

A smidgen of C from the scheduler


Isabelle/HOL
Abstract Specification
Refinement Proof
Executable Specification
Haskell Prototype
Automatic
Translation
Refinement Proof
High-Performance C Implementation



Refinement Proof by Forward Simulation

Cost of Production

Implementation (including Haskell prototype):

- 2.5 person years: 5,700 LOC Haskell; 8,700 LOC C
- vs 4 py estimated by the SLOCCount tool
- vs 6 py actual effort by Karlsruhe L4 team
- => evidence that Haskell prototype saved time

Proof: 20 py actual effort; 200,000 lines

- 9 py infrastructure
- 11 py seL4 verification

Estimate for repeat effort of kernel & proof:

- 8 py estimated (cf 4py & 6py above)

Effort of Each Proof

Abstract Spec vs Exec Spec

- 3 times more effort
- 300 changes to a-spec
- 200 changes to e-spec
- 50% of changes due to bugs
- 50% of changes for verification convenience

Exec Spec vs C-code

- easier
- 160 bugs found
- 16 of which were also found in testing (so testing missed 90%)
- mostly typos etc.

Cost of Change

Changing the proof is more expensive than changing the code. (W
the proof was more expensive too.) Cases:

1. Local, low-level change.
2. Adding new, independent features.
3. Adding new, large, cross-cutting features.
4. Fundamental changes to existing features.

However, no bug fixes required.

Important Design Decisions

1. Global variables & Side Effects
2. Kernel Memory Management
3. Concurrency & Non-Determinism
4. I/O

Global Variables & Side Effects

Use sparingly. Expensive to verify because they require invariants, which need to be checked against all code. Keep them modular and under control.

Haskell prototype helped with this, since side-effects in Haskell have to be made explicit.

Kernel Memory Management

Kernel only has mechanism.

Push policy to userspace.

- => don't need to verify policy

Invariants about the state of memory book-keeping data structures mean that certain checks can be done quickly at runtime. (This would not be safe without the proof.)

Concurrency & Non-Determinism

Short system calls.

Disable interrupts during system calls.

Therefore, no concurrency in the kernel.

Easy.

I/O

Hardware devices generate interrupts.

These are converted to IPC messages for the userspace device drivers. Hence, much complexity removed from kernel.

seL4 interrupt during system call?

seL4 fitness for future?

Better is Better
Less is More

Minimality $\Rightarrow$ Fitness for Purpose + Future

Extra Slides

	Haskell/C		Isabelle	Invar-	Proof	
	pm	kloc	kloc	iants	py	klop
abst.	4	—	4.9	~ 75	8	110
exec.	24	5.7	13	~ 80	3	55
impl.	2	8.7	15	0		

THEOREM 1. $\mathcal{M}_E$ *refines* $\mathcal{M}_A$.

THEOREM 2. $\mathcal{M}_C$ *refines* $\mathcal{M}_E$.

Therefore, because refinement is transitive, we have

THEOREM 3. $\mathcal{M}_C$ *refines* $\mathcal{M}_A$.

The Theorems to be Proved