



8,700 LoC 1 Microkernel 0 Bugs*

Gernot Heiser

John Lions Professor of Operating Systems, University of New South Wales
Leader, Trustworthy Embedded Systems, NICTA
CTO and Founder, Open Kernel Labs



Australian Government
Department of Communications,
Information Technology and the Arts
Australian Research Council



Department of State and
Regional Development



*Conditions apply

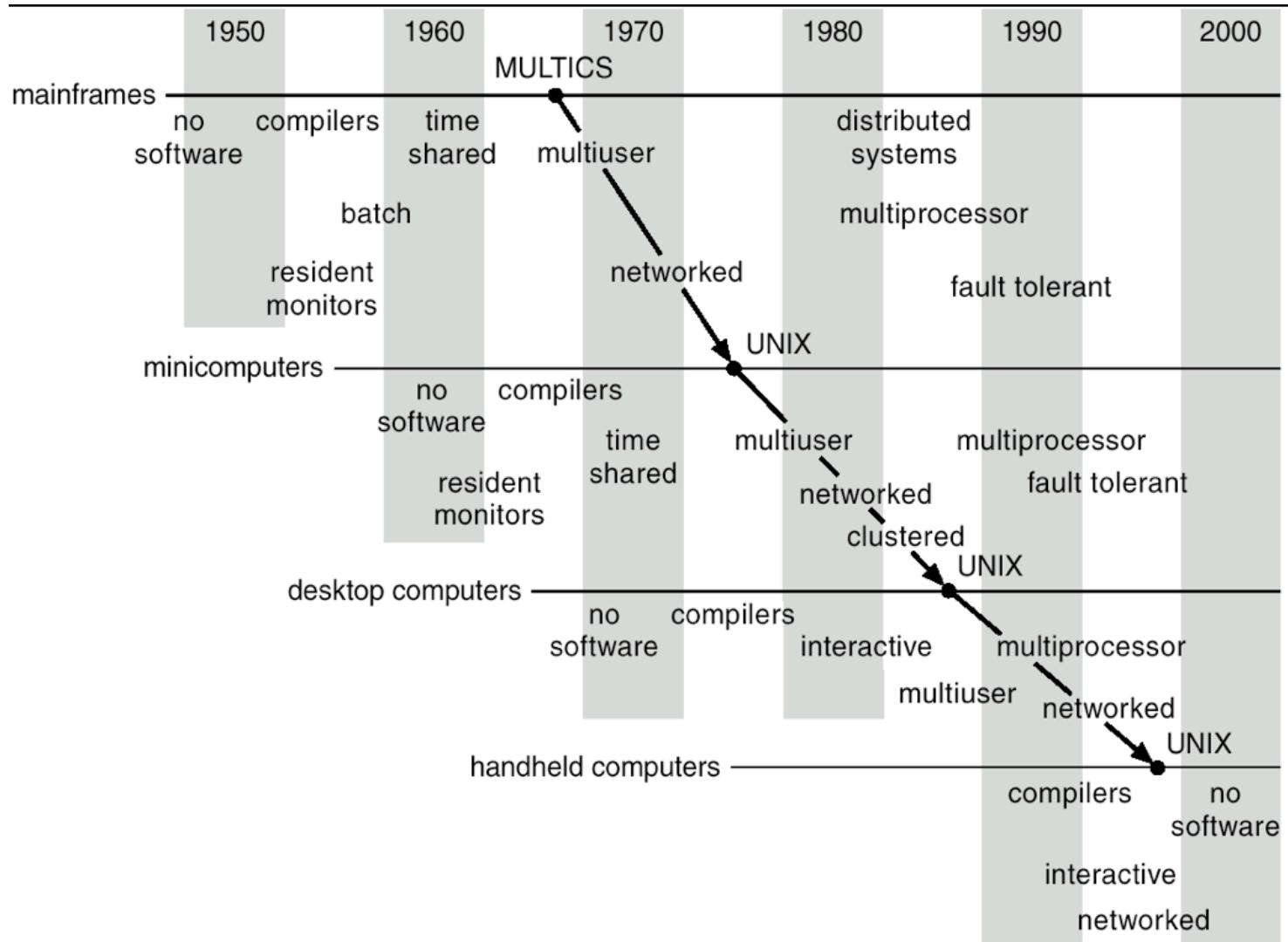
Windows

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

System-Software Timeline



The Problem



Microkernel Approach

Small trustworthy foundation

- Fault isolation
- Fault identification
- IP protection
- Modularity
- High assurance components in presence of other

Designed for verification

- small API

Designed for security

- novel kernel resource management

Untrusted



Trusted



seL4 Microkernel



Aim: Suitable for Real-World Use



Model: OKL4 microkernel

- resulting from L4-based research at NICTA/UNSW
- spun out to independent company Open Kernel Labs in 2006
- deployed in >300 M devices

seL4 API based on L4:

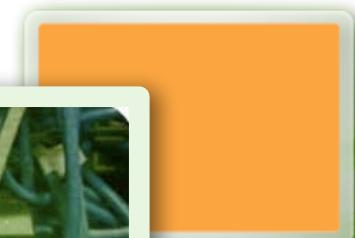
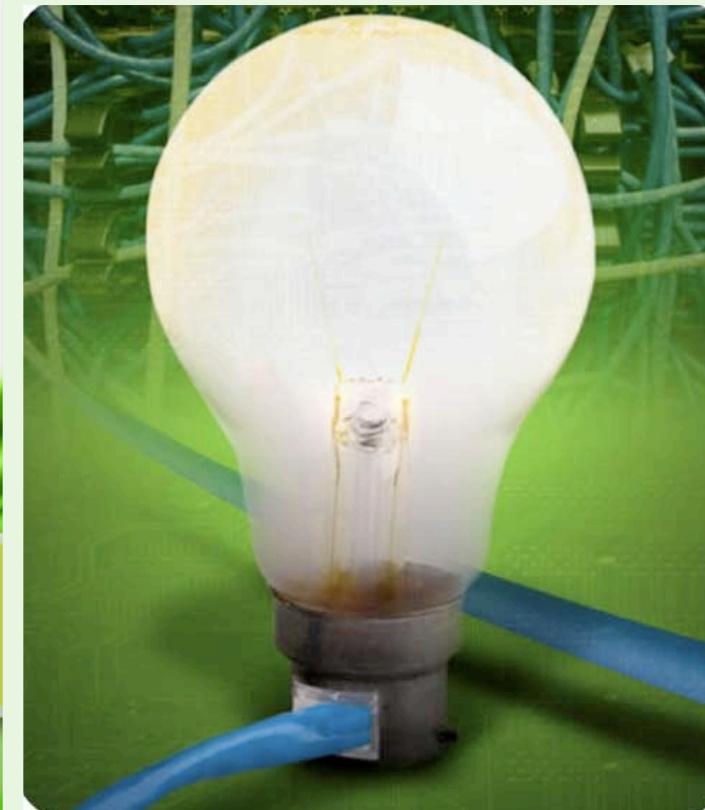
- IPC
- Threads
- Virtual Memory
- IRQs, exception redirection
- Capabilities



Open Kernel LabsTM
Be open. Be safe.



The Proof



What

Specification

definition

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

Proof

How

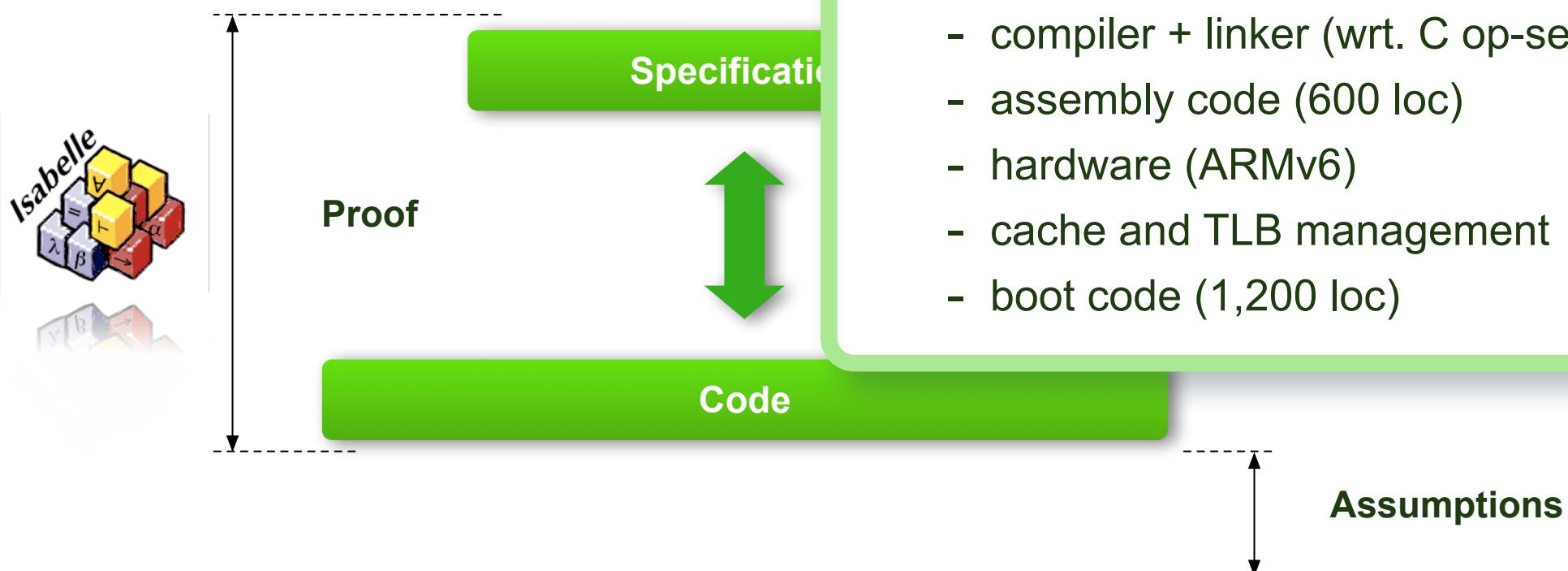
```
void
schedule(void) {
  switch ((word_t)ksSchedulerAction) {
    case (word_t)SchedulerAction_ResumeCurrentThread:
      break;

    case (word_t)SchedulerAction_ChoseNewThread:
      chooseThread();
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;

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

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

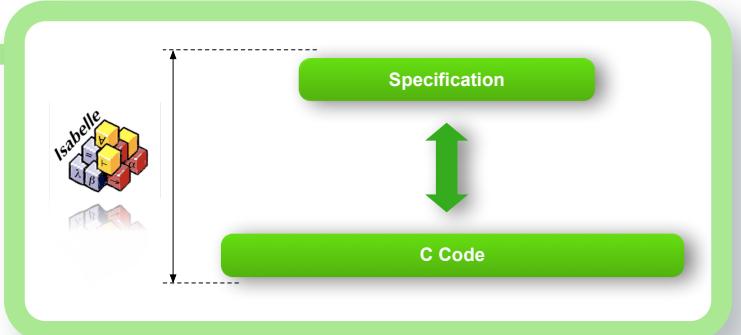
*conditions apply



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

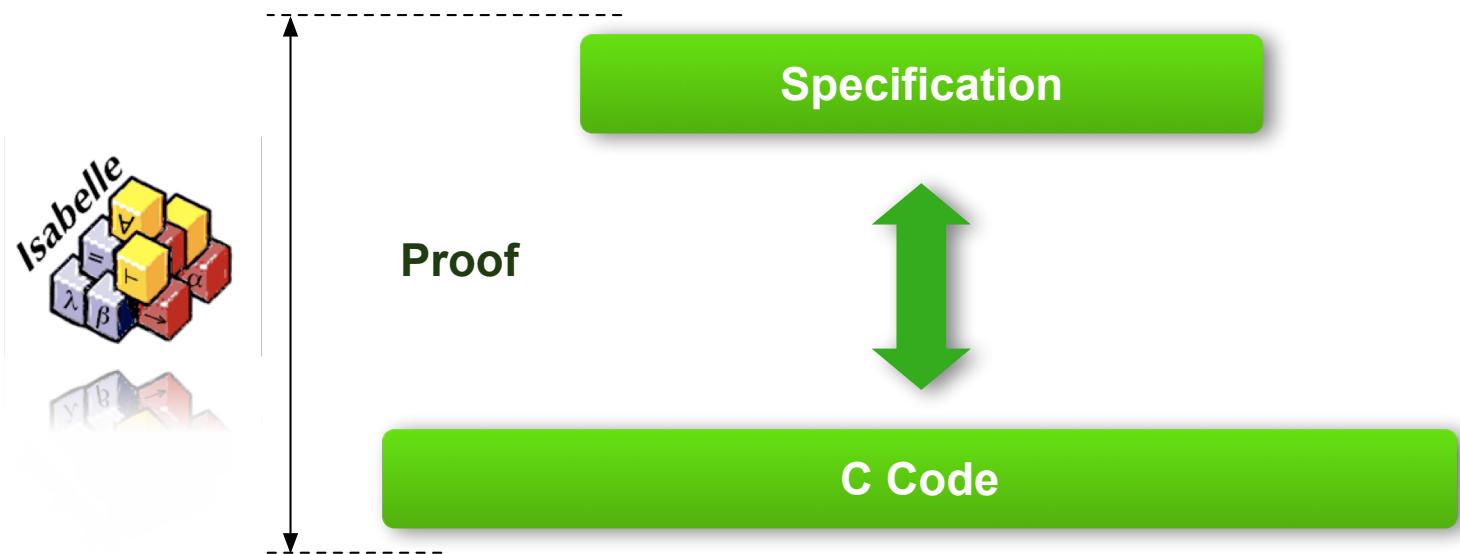


Not implied:

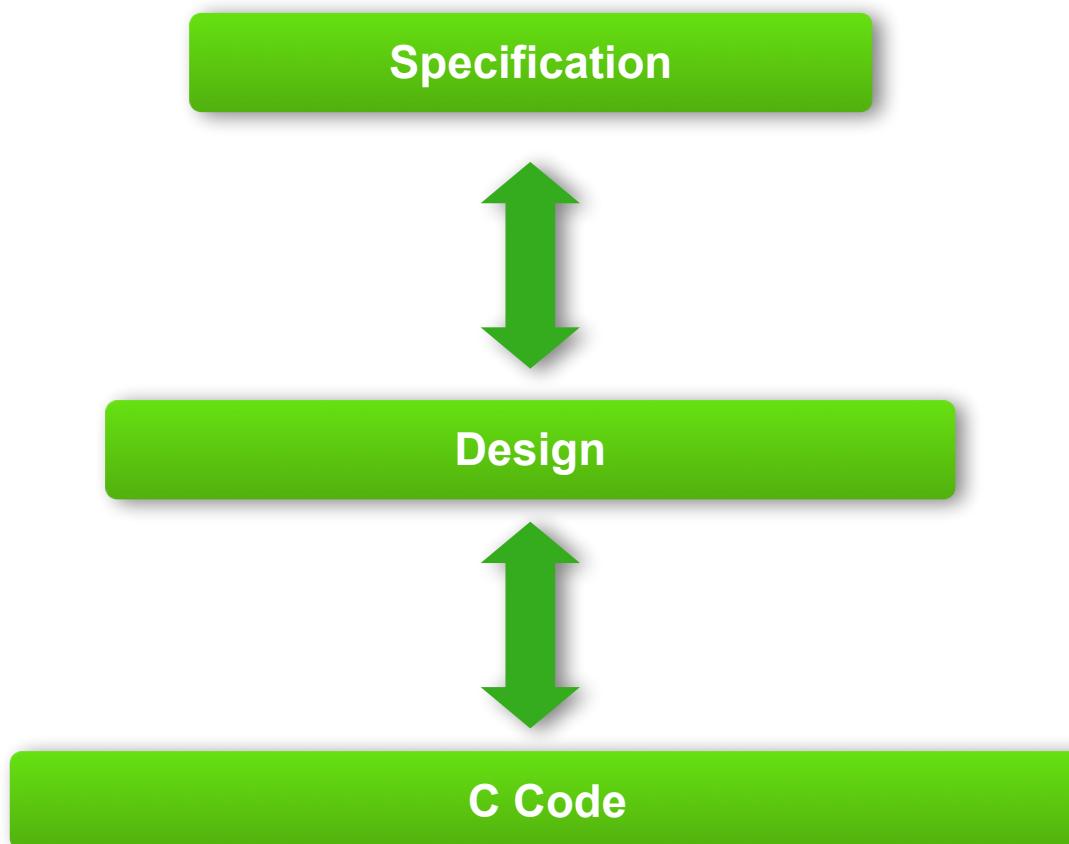
- “secure” (define secure)
- zero bugs from expectation to physical world
- covert channel analysis



Proof Architecture



Proof Architecture



Proof Architecture

Access Control Spec

Confinement



Specification

definition

```
schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads.
```



Design

schedule :: Kernel ()

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

    case (word_t)SchedulerAction_ChoseNewThread:
      chooseThread();
      ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
      break;

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

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



C Code

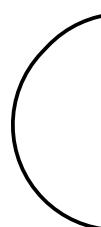
System Model

States:

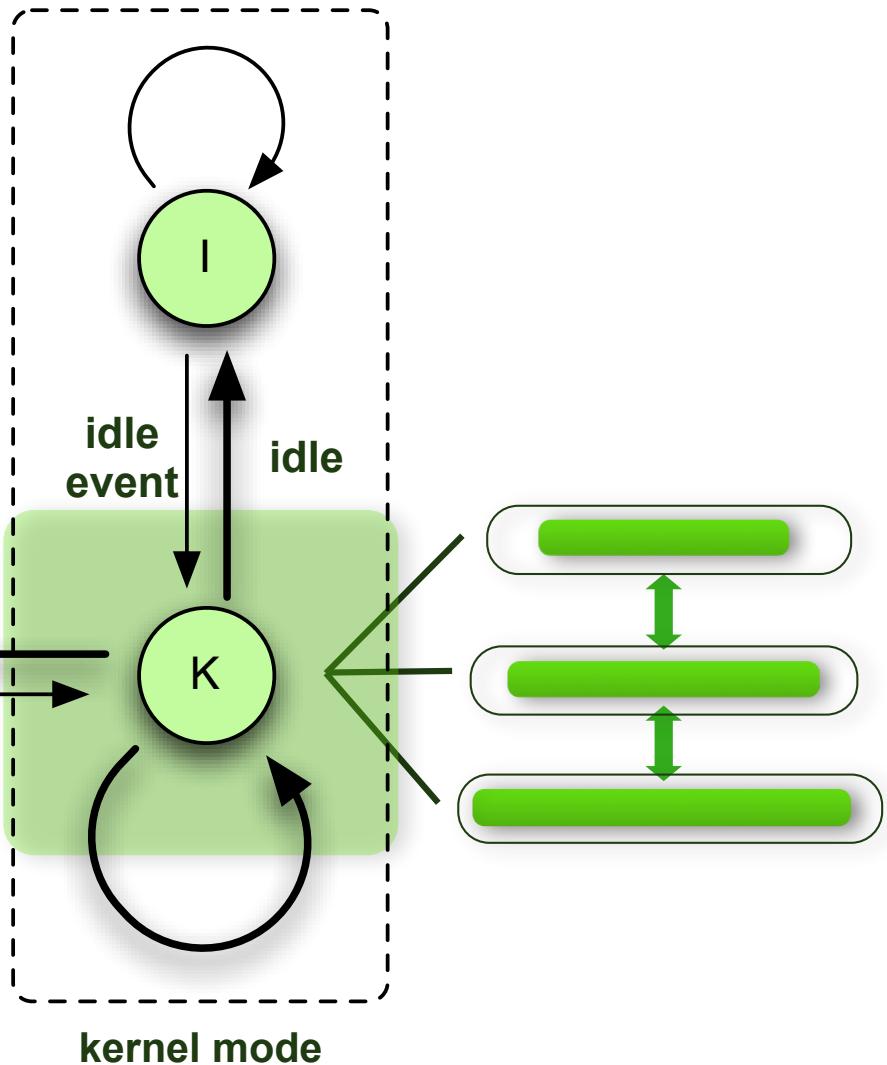
User, Kernel, Idle

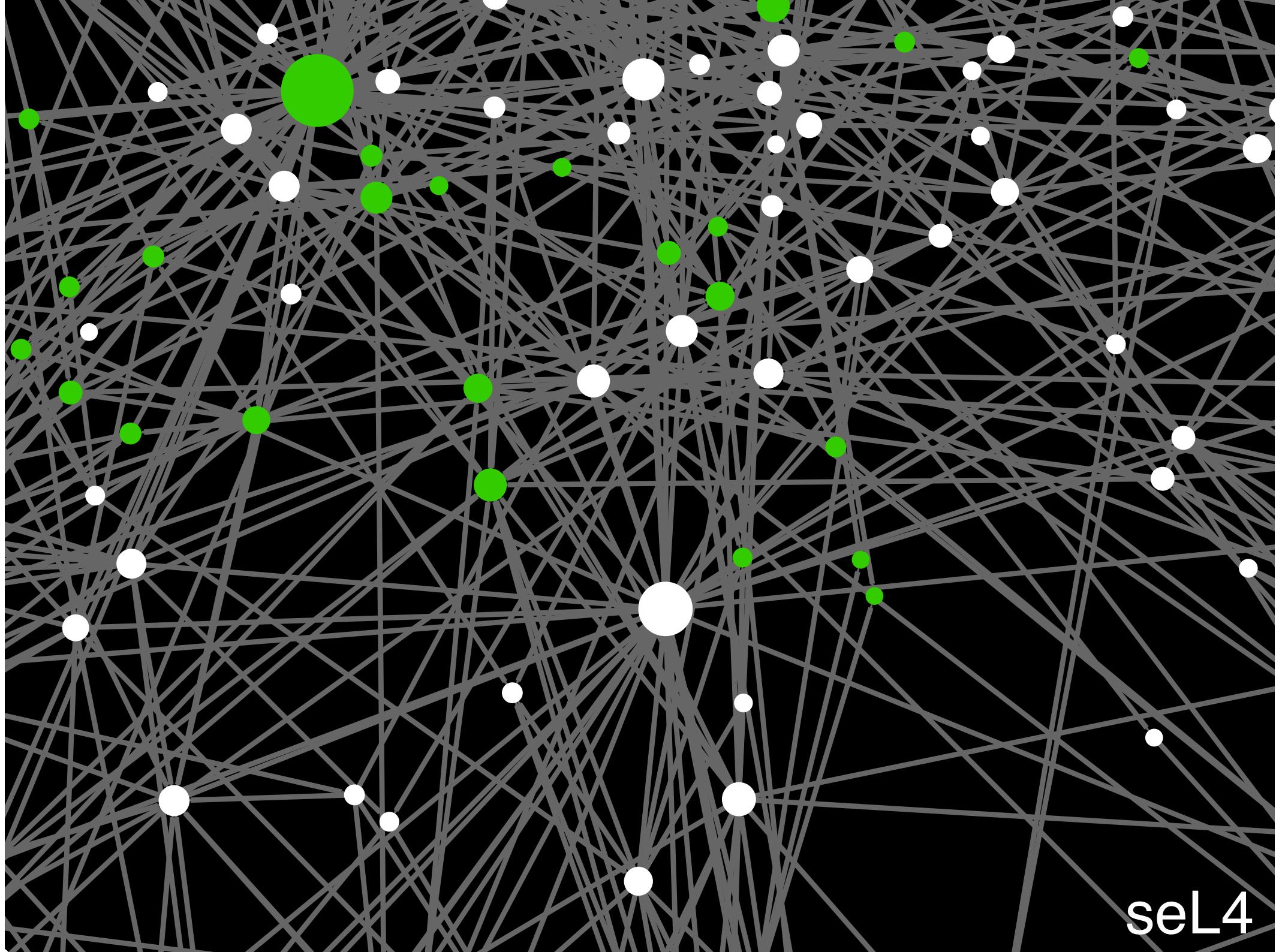
Events:

Syscall, Exception, IRQ, VM Fault



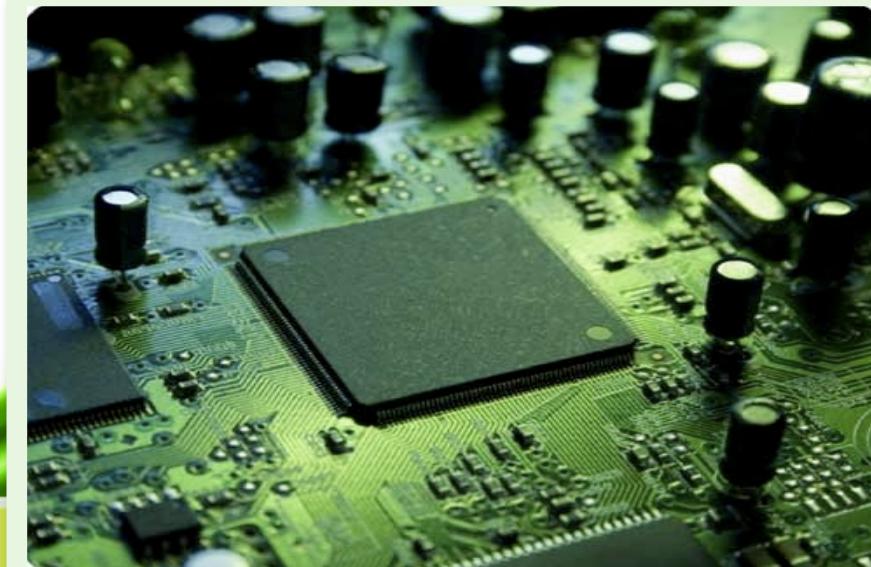
kernel exit
event





seL4

Kernel Design for Verification



Formal Methods Practitioners

Kernel Developers



**The Power of
Abstraction**

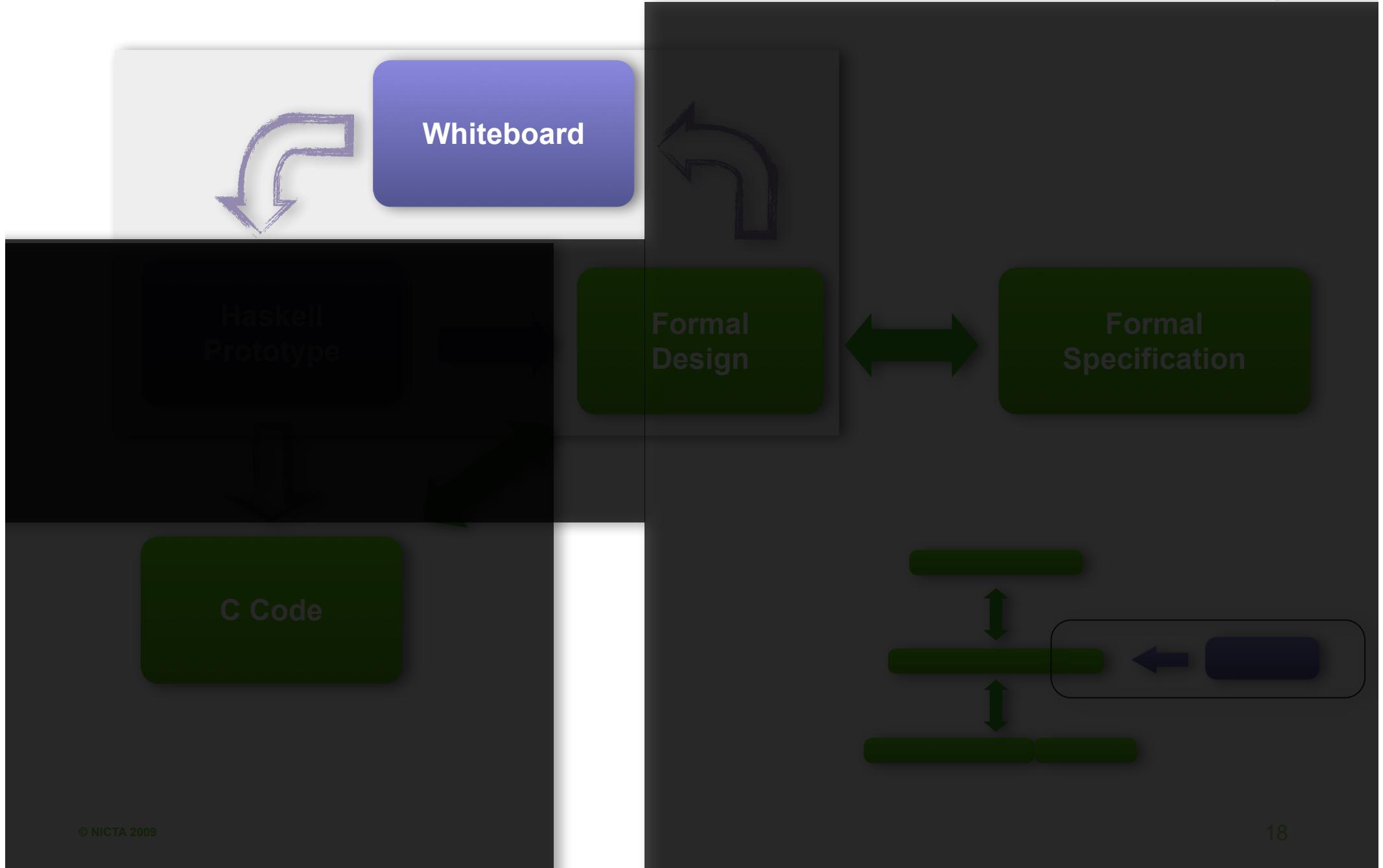
[Liskov 09]



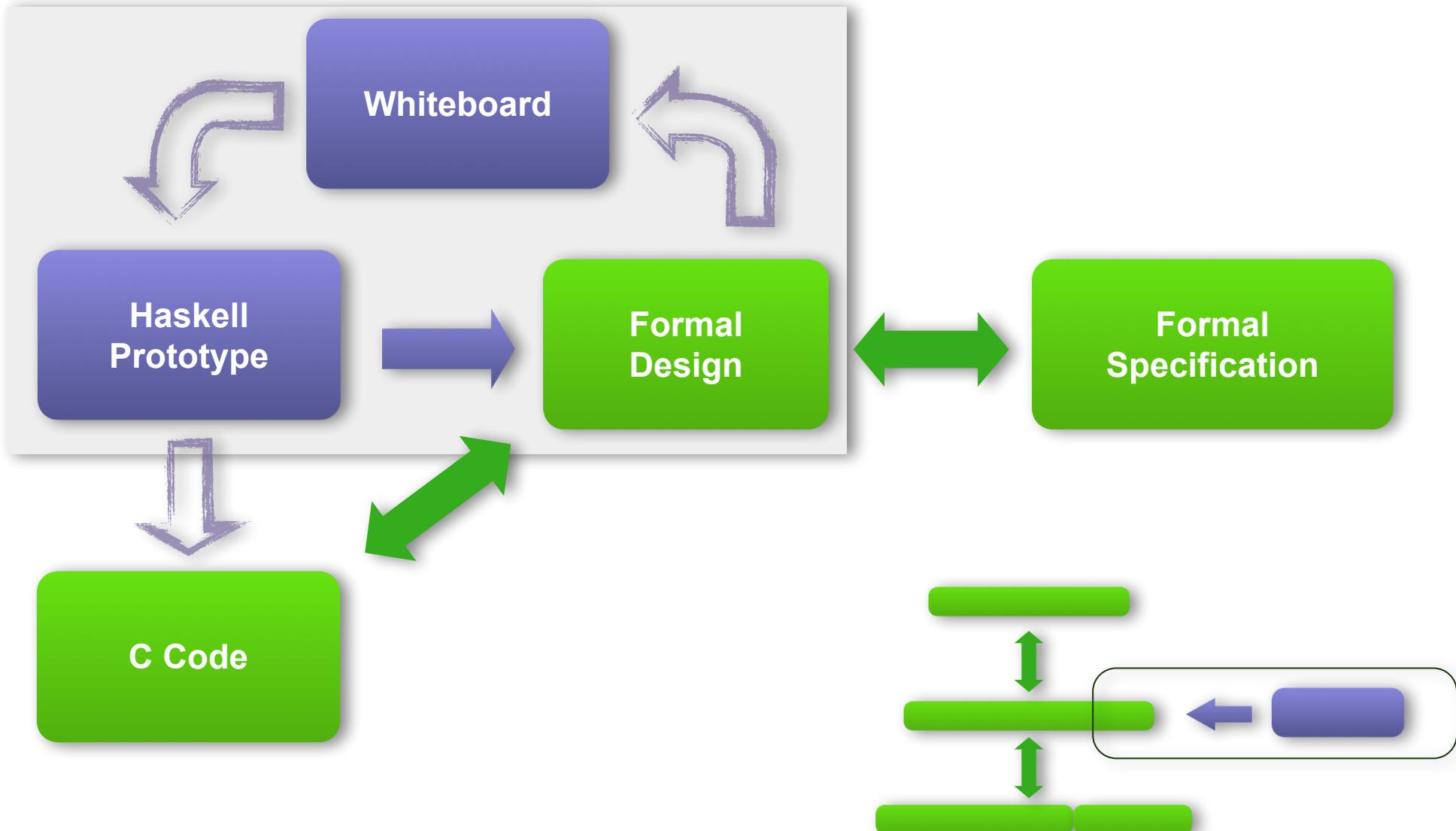
**Exterminate All
OS Abstractions!**

[Engler 95]

Iterative Design and Formalisation



Iterative Design and Formalisation



Reducing Complexity

Hardware

- drivers outside kernel

Concurrency

- event-based kernel
- limit preemption

Code

- derive from functional representation

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

        case (word_t)SchedulerAction_ChooseNewThread:
            chooseThread();
            SchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        /* SwitchToThread */
        case (word_t)SchedulerAction_SwitchToThread:
            switchToThread(ksSchedulerAction);
            SchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        default:
            /* If we get here, then we have an invalid scheduler action */
            /* We can't do anything but resume the current thread */
            SchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;
    }
}

void
chooseThread() {
    /* If there is no ready queue, then we can't do anything */
    if (ksReadyQueues[0].head == NULL)
        return;

    /* If there is only one thread in the ready queue, then we can
     * just resume it */
    if (ksReadyQueues[0].head == ksReadyQueues[0].tail)
        resumeThread(ksReadyQueues[0].head);

    /* If there are multiple threads in the ready queue, then we
     * need to choose one */
    else {
        /* We need to find the highest priority thread in the ready
         * queue */
        word_t maxPrio = 0;
        word_t prio;
        word_t head;
        word_t next;
        word_t thread;
        word_t pos;
        word_t maxPos;

        /* We start at the head of the ready queue */
        head = ksReadyQueues[0].head;
        pos = 0;
        maxPos = 0;

        /* We iterate through the ready queue until we find the
         * highest priority thread */
        while (head != NULL) {
            /* We check if the current thread has a higher priority
             * than the current maximum priority */
            if (head->prio > maxPrio) {
                maxPrio = head->prio;
                maxPos = pos;
            }

            /* We move to the next thread in the ready queue */
            head = head->tcbSchedNext;
            pos++;
        }

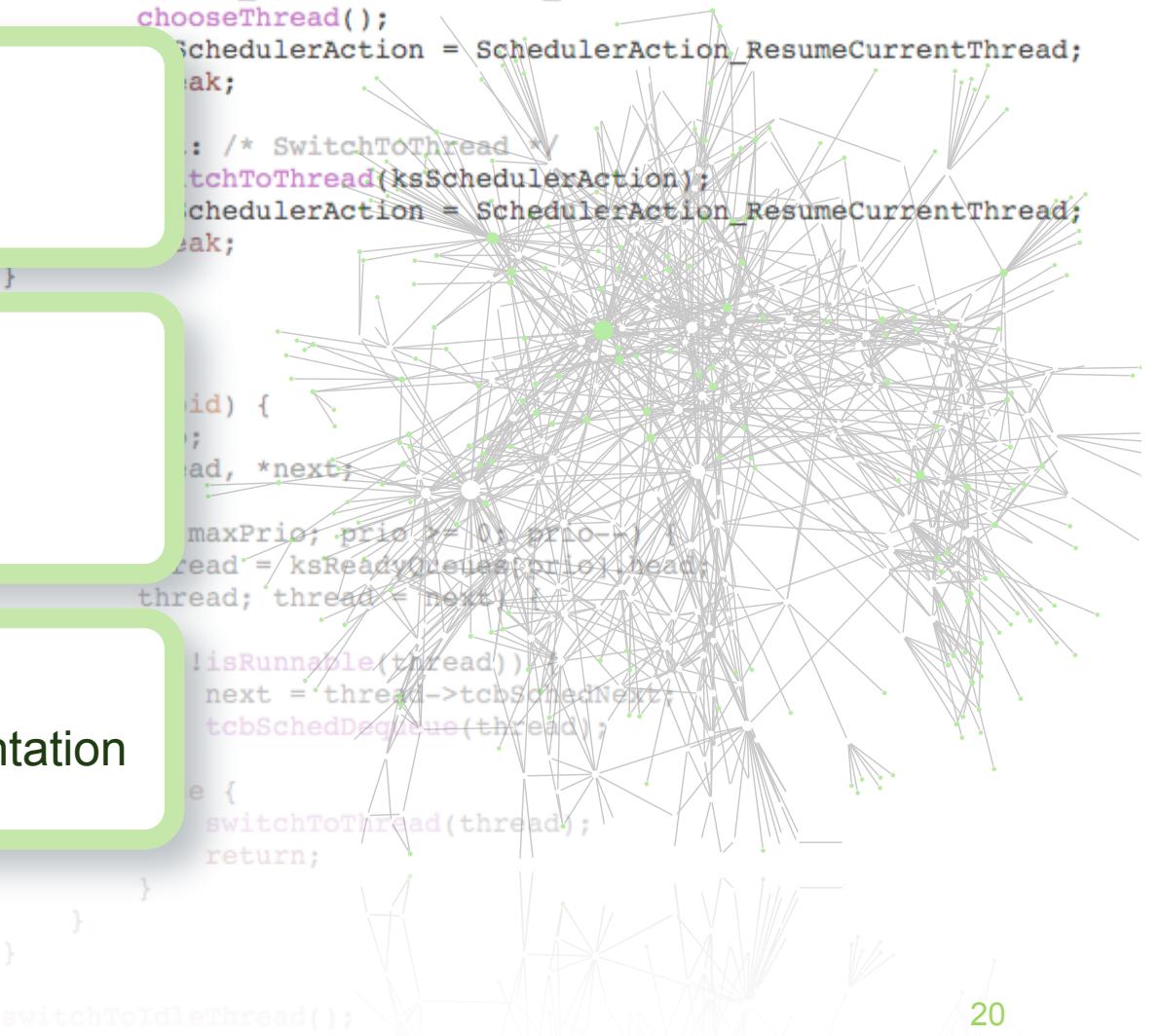
        /* Once we have found the highest priority thread, we
         * resume it */
        if (maxPrio != 0) {
            resumeThread(ksReadyQueues[0].head);
        }
    }
}

void
resumeThread(thread_t *thread) {
    /* We switch to the thread */
    switchToThread(thread);
}

void
switchToThread(thread_t *thread) {
    /* We update the current thread */
    ksCurrentThread = thread;

    /* We update the tcbSchedDequeue field of the current thread
     * to point to the next thread in the ready queue */
    thread->tcbSchedDequeue = ksReadyQueues[0].head;

    /* We switch to the thread */
    switchToIdleThread();
}
```



Everything from C standard

- **including:**

- pointers, casts, pointer arithmetic
- data types
- structs, padding
- pointers into structs
- precise finite integer arithmetic

- **minus:**

- goto, switch fall-through
- reference to local variable
- side-effects in expressions
- function pointers (restricted)
- unions

- **plus** compiler assumptions on:

- data layout, encoding, endianess

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

        case (word_t)SchedulerAction_ChoseNewThread:
            chooseThread();
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        default:
            break;
    }
}

void
chooseThread()
{
    word_t
    lt
    wi
    ss
    re
    vo
    io
    re
    =
    hr
    chre
    if(!isRunnable(thread)) {
        next = thread->tcbSchedNext;
        tcbSchedDequeue(thread);
    }
    else {
        switchToThread(thread);
        return;
    }
}

void
switchToIdleThread();

```

Did you find any Bugs?



NICTA

Bugs found

during testing: 16



during verification:

- in C: 160
- in design: ~150
- in spec: ~150

460 bugs

Effort

Haskell design	2	py
First C impl.	2	weeks
Debugging/Testing	2	months
Kernel verification	12	py
Formal frameworks	10	py
Total	25	py

Comparison of approaches

Trad. engineering 4-6 py
Repeat verification 6 py

Cost

Common Criteria EAL6: \$87M
L4.verified: \$6M

Formal proof all the way from spec to C

- **200 kLoC** handwritten, machine-checked proof, **10 k theorems**
- **~460** bugs (160 in C)
- Verification on **code, design, and spec**
- **Hard in the proof** → **Hard in the implementation**

Formal Code Verification up to 10 kLoC:

It works.
It's feasible. **(It's fun, too...)**
It's cheaper.





Remove limitations

- verify assembler code
- verify bootstrap code
- verify MMU operations
- multicore version
- verify x86 version
- temporal isolation
- information flow

Towards real systems

- 1 MLoC
- real-time analysis
- power management



The Team (Past and Present)



- June Andronick
- Timothy Bourke
- Andrew Boyton
- David Cock
- Jeremy Dawson
- Philip Derrin
- Dhammadika Elkaduwe
- **Kevin Elphinstone**
 - *leader, kernel design*
- Kai Engelhardt
- David Greenaway
- Lukas Haenel
- Gernot Heiser
- **Gerwin Klein**
 - *leader, verification*
- Rafal Kolanski
- Jia Meng
- Catherine Menon
- Michael Norrish
- Thomas Sewell
- David Tsai
- Harvey Tuch
- Michael von Tessin
- Adam Walker
- Simon Winwood

Thank You

