From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?

Kevin Elphinstone and Gernot Heiser

Presented by: Yuzhong Wen

What is L4?

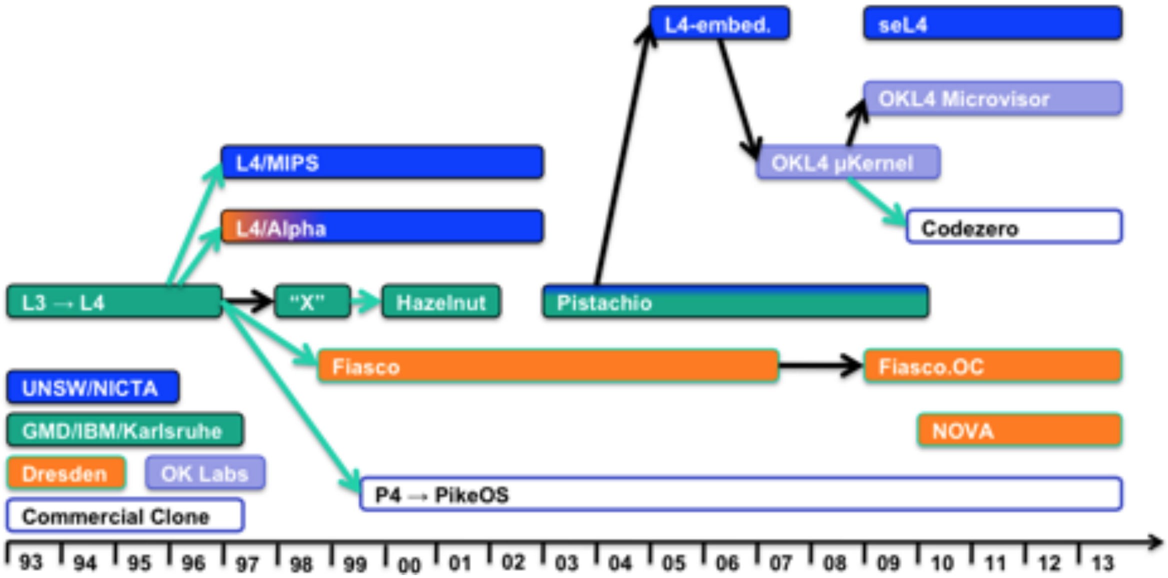
- Invented by Jochen Liedtke
- A family of microkernels
  - Active: seL4, NOVA, OKL4, Fiasco.OC
  - Deactive: L4Ka::Pistachio, NICTA::Pistachio-embedded, L4 Hazelnut, L4/Alpha, L4/MIPS...
- Widely used
  - Real-time systems
  - Resource limited systems
  - Security related systems

What is L4?

- Invented by Jochen Liedtke
- A family of microkernels
  - Active: seL4, NOVA, OKL4, Fiasco.OC
  - Deactive: L4Ka::Pistachio, NICTA::Pistachio-embedded, L4 Hazelnut, L4/Alpha, L4/MIPS…
- Widely used
  - Real-time systems
  - Resource limited systems
  - Security related systems

Verification

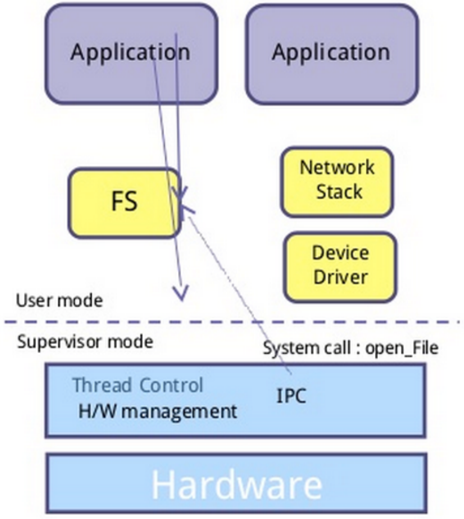
What is L4?



L4-embed.
seL4
OKL4 Microvisor
L4/MIPS
OKL4 µKernel
L4/Alpha
Codezero
L3 → L4
"X"
Hazelnut
Pistachio
Fiasco
Fiasco.OC
UNSW/NICTA
GMD/IBM/Karlsruhe
NOVA
Dresden
OK Labs
P4 → PikeOS
Commercial Clone
93 94 95 96 97 98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13

What is L4?

- System design
  - The kernel is “micro”
  - Device drivers, network stack are in userspace

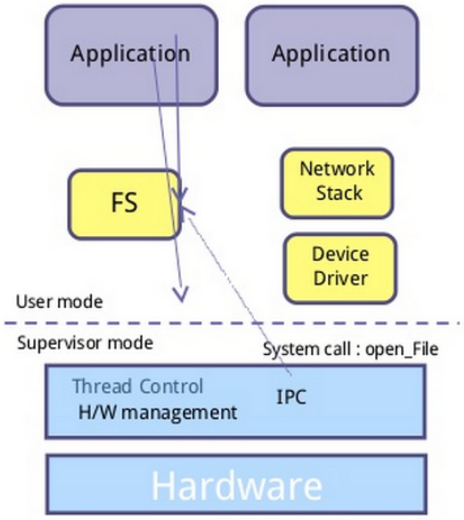


What is L4?

- System design
  - The kernel is “micro”
  - Device drivers, network stack are in userspace

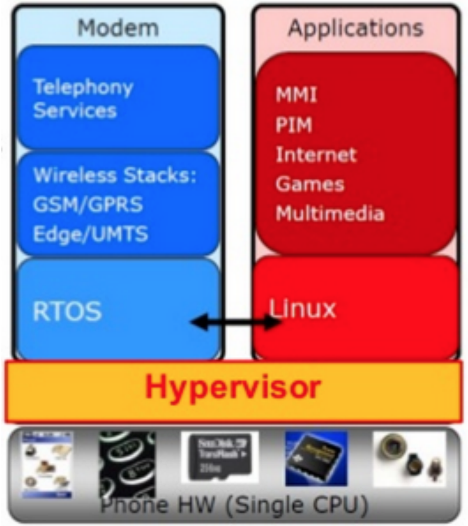
Minimality

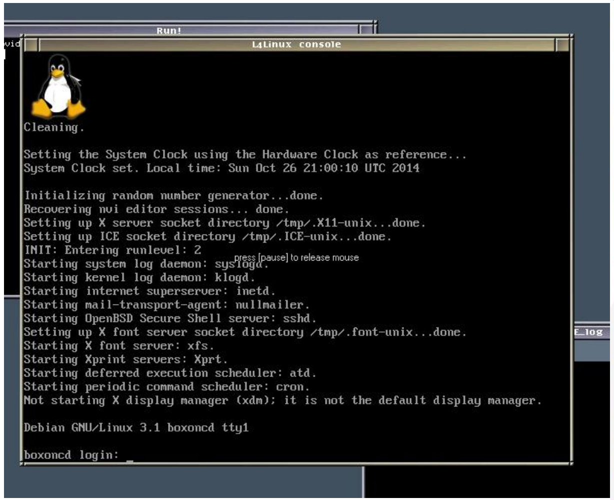
High performance IPC

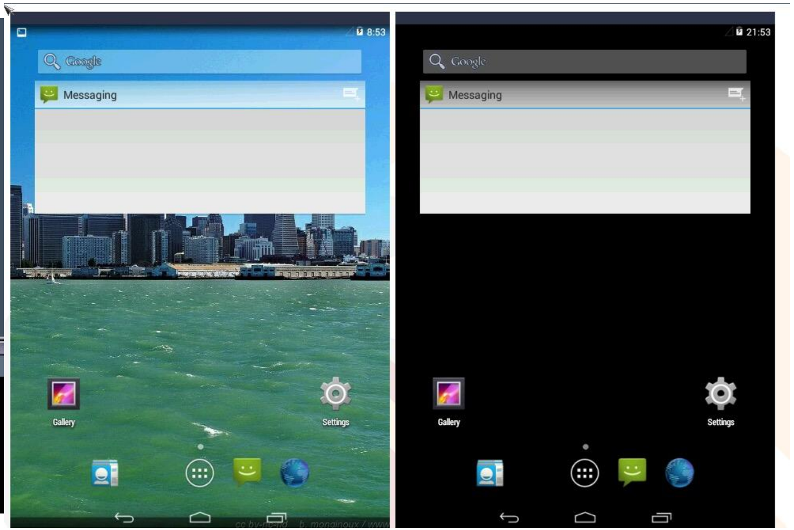


What is L4?

- Beyond the kernel
    - OS layer as userspace process







The problem?

- IPC design
- Hardware resource management
- Process management
- Programmability

IPC design

Synchronous IPC

- Synchronous IPC
  - Essential for L4 implementation
  - Not flexible for handling interrupts
  - Not scalable
- Synchronous + Asynchronous IPC
  - Asynchronous endpoints
  - Violate minimality!
- Pure asynchronous

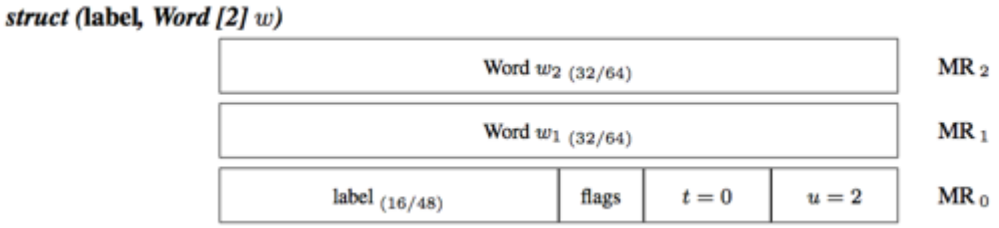
Synchronous IPC

- Synchronous IPC
  - Essential for L4 implementation
  - Not flexible for handling interrupts
  - Not scalable
- Synchronous + Asynchronous IPC
  - Asynchronous endpoints
  - Violate minimality!
- Pure asynchronous

From synchronous to asynchronous

IPC message structure

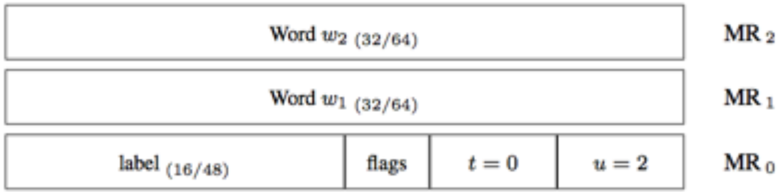
- In register messages(short message)
  - Physical register based messages
    - Limited by architecture
  - Virtual message registers
    - Fixed size
    - Flexible



IPC message structure

- In register messages(short message)
  - Physical register based messages
    - Limited by architecture
  - Virtual message registers
    - Fixed size
    - Flexible

struct (label, *Word [2] w*)



From physical to virtual

IPC message structure

- Long IPC
  - Triggers massive page faults
  - Rarely used (mainly used by legacy POSIX interface)
  - Hard to do verification
  - Violate minimality!

Abandon Long IPC

IPC destination

- Thread ID as destination
  - Expose one thread's internal to another
  - Unflexible
- IPC endpoint as destination
  - Asynchronous Endpoints
  - Synchronous Endpoints
  - Better management

Object	Object Size
n-bit Untyped	2^n bytes (where $n \geq 4$)
n-slot CNode	$16n$ bytes (where $n \geq 2$)
Synchronous Endpoint	16 bytes
Asynchronous Endpoint	16 bytes
IRQ Control	—
IRQ Handler	—

From Thread ID to IPC endpoint

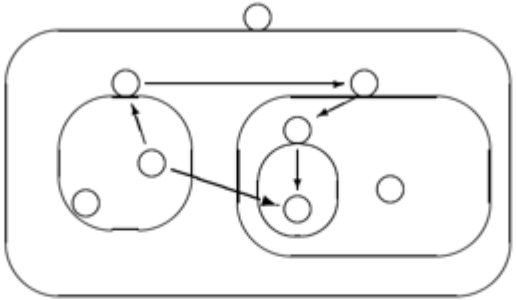
IPC timeout

- Blocking IPC
  - Suffers from DOS attack
- IPC timeout
  - Doesn't help at all
- No timeout at all!
  - A flag to indicate using polling or blocking

Abandon timeout

Communication Control

- “Chief and clans”
  - Provides access control
  - Overhead in inter-clan communication
- Capability control
  - Access control based on kernel objects

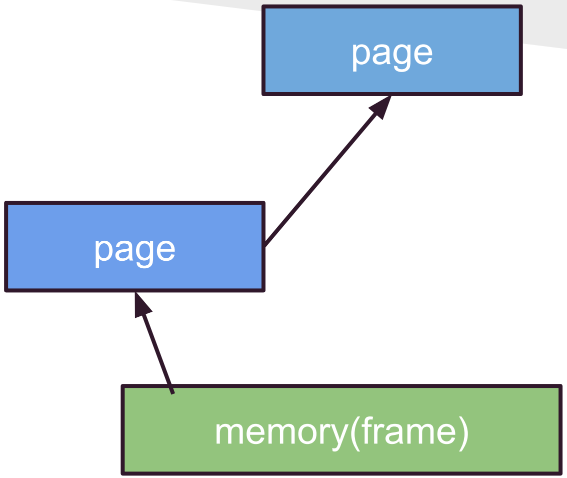


Abandon chief and clans

Hardware resource management

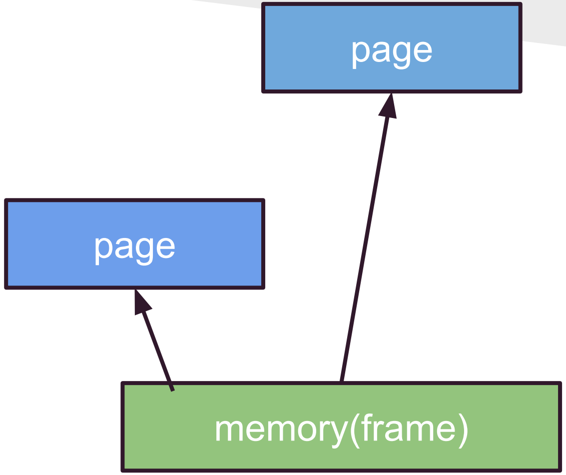
Resource management

- Recursive page mappings
  - Flexible page mapping between threads
  - Map from virtual pages
  - Map from physical frames



Page mapping

- Recursive page mappings
  - Flexible page mapping between threads
  - Map from virtual pages
  - Map from physical frames



Retain the mapping from pages

Map from physical frames

Kernel memory

- Allocate objects directly from free memory
  - Not safe
  - Hidden from userspace
- Allocate objects from untyped objects
  - Untyped objects are well controlled
  - All objects are controlled by capabilities

User-level memory control

Object	Description
TCB	Thread control block
Cnode	Capability storage
Synchronous Endpoint	Port-like rendezvous object for synchronous IPC
Asynchronous Endpoint	Port-like object for asynchronous notification.
Page Directory	Top-level page table for ARM and IA-32 virtual memory
Page Table	Leaf page table for ARM and IA-32 virtual memory
Frame	4 KiB, 64 KiB, 1 MiB and 16 MiB objects that can be mapped by page tables to form virtual memory
Untyped Memory	Power-of-2 region of physical memory from which other kernel objects can be allocated

Table 3: seL4 kernel objects.

Time (clock source)

- Time multiplexing
  - The key of scheduling
  - Has to be done in kernel
  - Violate minimality!

Unsolved (may be removed from kernel)

Multicore

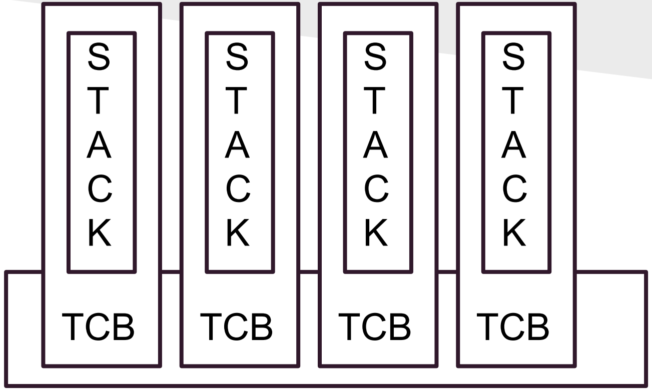
- Biglock
  - Bad scalability
- Multikernel
  - One kernel one core

Unsolved (concurrency is hard to verify)

Process management

TCB management

- Virtual TCB array
  - Indexed by thread id
  - Each thread(TCB) has a kernel stack
  - Easy to find the stack from TCB
  - Large memory overhead
  - Large cache footprint




```
INLINE word_t * tcb_t::get_stack_top ()
{
    return (word_t*)addr_offset(this, KTCB_SIZE);
}
```

TCB management

- Virtual TCB array
  - Indexed by thread id
  - Each thread(TCB) has a kernel stack
  - Easy to find the stack from TCB
  - Large memory overhead
  - Large cache footprint
- Single physically-allocated stack
  - Few IPC performance overhead

```
/* The stack is the very last page of virtual memory. */
#define PPTR_KERNEL_STACK 0xfffff000
```

Abandon Virtual TCB array

Scheduling

- Lazy scheduling

```
tcb_t chooseThread(void) {
  foreach prio ∈ prios
    foreach thread ∈ runQueue[prio]
      if runnable(thread)
        return thread
      else
        schedDequeue(thread)
}
```

Scheduling

- Lazy scheduling
  - Just put the blocking thread back into runnable queue
  - Performance is bad on real-time systems
- Benno scheduling
  - Every thread on the queue is runnable

```
tcb_t chooseThread(void) {
  foreach prio ∈ prios
    thread = runQueue[prio].head
    if thread != NULL
      return thread
}
```

From lazy scheduling to Benno scheduling

Programmability

Programmability

- Language
  - Assembler
    - Hard to maintain
  - C++
    - No good compiler
    - Can't be verified
- Calling convention
  - Hard to port or verify without good calling convention

Abandon assembler and C++

Abandon non-standard calling conventions

Programmability

- No portability!?
  - L4 was coded to directly talk to hardware
- Portability
  - Glue layer for different architecture

Introduce glue layer for portability

Thanks!