The seL4 Capability System
The seL4 Capability System
(A user’s perspective)
The seL4 Capability System

(A user’s perspective)
(Circa 2013)
The seL4 Capability
System
(A user’s perspective)
(Circa 2013)

Matthew P. Grosvenor
A little history
A little (user) history
A little (user) history

• 2008 - Advanced Operating Systems (Heiser et.al)
  • Built an OS personality on OKL4 (in 12 weeks)
• Nov 2008 - Joined NICTA
  • Summer intern - 12 weeks
  • Which became 18 months PT for course credit
500 days of seL4

1. Wrote the first draft of what would become the seL4 User Manual

2. Tried to solve this problem:
   - How do you write capalloc()
   - How do you write capfree()
500 days of seL4

1. Wrote the first draft of what would become the seL4 User Manual

2. Tried to solve this problem:
   - How do you write `capalloc()`
   - How do you write `capfree()`

Why was this so hard???
Part I: The really good ideas in the seL4 Capability System
The seL4 Kernel

- L4 family micro-kernel, realtime OS**
- About 10,000 lines of C, and a few hundred lines of Asm
- First “general purpose” “kernel” to be fully verified
- Machine checked refinement proof that
  - The (ARM) C code implements an executable Haskell model
  - The Haskell model implements a high level specification
- Capability based
seL4 Protection

• Capabilities using MMU / rings for protection.

• All dynamic allocation in the kernel handled via capability system.

• All system calls are capability “invocations”
seL4 Protection

- Capabilities using MMU / rings for protection.

- All dynamic allocation in the kernel handled via capability system. **No dynamic memory allocation in the kernel!**

- All system calls are capability “invocations”
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

Mem

Physical memory
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>VGA</td>
<td>Mem</td>
<td>NIC</td>
<td>Mem</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

Some memory is “special” (devices), so we carve it out into special ranges
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x10</th>
<th>0x11</th>
<th>0x55A</th>
<th>0x66A</th>
<th>0xFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>VGA</td>
<td>Mem</td>
<td>NIC</td>
<td>Mem</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

Allocate some memory to the kernel
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

Physical memory

Everything else becomes a memory object
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

Physical memory

Nothing special about devices, just memory
seL4 System Objects

Ideal world for seL4: everything is physically memory mapped

Physical memory
Nothing special the devices, : just memory
(corner case: but, physical addresses matter for these, discussed later)
seL4 System Objects

Real world for seL4: everything is physically memory mapped + plus a couple of extras

Physical mem

0x00 0x10 0x11 0x55A 0x66A 0xFFF

Kernel
Mem
Mem
Mem
Mem

I/O ports
IRQs
...

0x01 0x02 0x03 0x04

0x05 0x06 0x07
Real world for seL4: everything is physically memory mapped + plus a couple of extras

Physical mem 0x01 0x02 0x03 0x04

Kernel

Mem Mem Mem Mem

I/O ports IRQs ...

0x05 0x06 0x07

Ignore these
More on seL4 Objects

- All objects have a “type” (more in a moment)
- All objects are power of 2 sized
- All objects are power of 2 aligned
- Have some kind of physical address (mostly mem mapped)
- Reside in the “object space” or physical memory space
More on seL4 Objects

• All objects have a “type” (more in a moment)

• All objects are power of 2 sized

• All objects are power of 2 aligned

• Have some kind of physical address (mostly mem mapped)

• Reside in the “object space” or physical memory space
Object Types

- Untyped objects - default type
  - “memory objects”, in seL4 called “untyped”

- Typed Objects:
  - TCB - Thread control block (object)
  - Page Directory / Page Table (objects)
  - IPC end point / AsyncIPC endpoint (objects)
  - Capability Objects (called “Cap Nodes” or “CNodes”)
seL4 Capabilities

• Access to every object is mediated through a capability.

• seL4 “syscalls” are capability “invocations” on objects
  • e.g., Map a page, start a thread etc.

• seL4 Caps:
  • 16B - Stored in a “CNode” object
  • Store object type and physical address
  • Access control (R/W/E/M) flags
  • Include the capability derivation tree (more later)
Capabilities and Objects

Physical mem

0x00  0x10  0x11  0x55A  0x66A  0xFF

Kernel  CNode  Mem  Mem  Mem

Capability Node (object) contains a list of capabilities which are (roughly) fat pointers to other objects in the system
Capabilities and Objects

Capability Node (object) contains a list of capabilities which are (roughly) fat pointers to other objects in the system
Capability Addressing

Capability Node Object
Capability Addressing

Capability Node Object

```plaintext
Cap  Cap  Cap
```
Capability Addressing

Capability Node Object

Diagram showing a Capability Node Object with various Capabilities (Cap) and a CNode Cap.
Capability Addressing

Capability Node Object

0x00

0x0

Capability Node Object

CNode Cap

Cap

Cap

Cap

0xFF

0xF
Capability Addressing

- Each address is a 32bit number or index into a CNode
- Each CNode has a number of slots
- Each CNode has a “guard” value and length.
- Sometimes you need a depth as well
Capability Address Space

0x00

CNode Cap

Cap  Cap  Cap

0x0

0xFF

0xF
Capability Address Space

256 slots

0x00

CNode Cap

Cap

Cap

Cap

16 slots

0x0

0xF
Capability Address Space

(Guarded Page Table)

0x0000 0x00 256 slots 0xFF

16b guard  CNode Cap  Cap  Cap

Capability Node Object

0x0 0x0 16 slots 0xF

4bit guard
Capability Address Space

(Guarded Page Table)

16b guard | CNode Cap | 256 slots | Cap | Cap | Cap |

Capability Node Object

4bit guard | 16 slots | 0xF

Diagram shows the structure of the capability address space with guard bits and capability objects.
Capability Address Space

(Guarded Page Table)

- 0x0000 to 0x00: 16b guard
- 0x0000 to 0xFF: 256 slots
- 0x00: CNode Cap
- 0x00 to 0xF: 16 slots
- 0x0000 to 0x0F: 4bit guard

Capability Node Object
Capability Address Space

Address = 0x00000103

16b guard | CNode Cap | 256 slots | Cap | Cap | Cap

Capability Node Object

4bit guard | Cap | 16 slots
Capability Address Space

Address = 0x00000103

0x0000 0x00 256 slots 0xFF

16b guard CNode Cap Cap Cap

Capability Node Object

0x0 0x0 16 slots 0xF

4bit guard Cap
Capability Address Space

Address = 0x00000103

Capability Node Object

16b guard

256 slots

0x0000 0x00 256 slots 0xFF

16 slots

0x00000103

4bit guard

16 slots

0x0 0x0 16 slots 0xF

Cap

0x00000103

Cap
Capability Address Space

Address = 0x00000103

16b guard

CNode Cap

Cap

Cap

Cap

Capability Node Object

4bit guard

Cap
Capability Address Space

Address = 0x00000103

16b guard
CNode Cap

0x0000  0x00  256 slots  0xFF

Capability Node Object

0x0  0x0  16 slots  0xF

4bit guard
Cap
More on seL4 Objects

- All objects have a “type” (more in a moment)
- All objects are power of 2 sized
- All objects are power of 2 aligned
- Have some kind of physical address (mostly mem mapped)

- Reside in the “object space” or physical memory space
seL4 Spaces

- “object space” or physical memory space
- “capability space” (cspace)
- Capability derivation space
Capability Derivation Space

Untyped

Untyped

Untyped

Untyped

CNode

TCB

Page Dir

Page Tble
seL4 Spaces

- “object space” or physical memory space
- “capability space” (cspace)
- Capability derivation space
- Virtual memory space (vspace)
Virtual Memory Space
seL4 Spaces

- “object space” or physical memory space
  physical memory addresses

- “capability space” (cspace) capability addresses

- Capability derivation space (derivation tree)

- Virtual memory space (vspace) virtual memory addresses
Part II: Why the ideas in Part I were bad ideas
seL4 Spaces

- “object space” or physical memory space
  physical memory addresses

- “capability space” (cspace) capability addresses

- Capability derivation space (derivation tree)

- Virtual memory space (vspace) virtual memory addresses
seL4 Spaces

- “object space” or physical memory space
  physical memory addresses

- “capability space” (cspace) capability
  addresses

- Capability derivation space (derivation tree)

- Virtual memory space (vspace) virtual memory
  addresses

As a seL4 programmer, you have to think in 4 different address spaces at the same time.
seL4 Spaces

- “object space” or physical memory space

- “capability space” (cspace) capability addresses

- Capability derivation space (derivation tree)

- Virtual memory space (vspace) virtual memory addresses

**None of these are available (queriable) by userspace**
The root process

- Hard coded into the OS image
- At boot time, is given a description of the boot capability state "boot info".
- No virtual memory (stack only)
- A few (16-256) “free” capability node slots
Bootinfo

• A “machine parseable” description of the boot time state of the CSpace

• Includes some hints about special “untyped” memory regions for devices (eg VGA)
And now the problem

- Given:
  - an initial set of capabilities (bootinfo)
  - a running root process
- How do you arbitrarily allocate/deallocate capabilities?
Book Keeping

• If you make changes to the default CSpace, you need to bookkeep them.
Book Keeping

• If you make changes to the default CSpace, you need to bookkeep them.

• **Where do you put the book-keeping?**
Book Keeping

• If you make changes to the default CSpace, you need to bookkeep them.

• Where do you put the book-keeping? \textbf{Virtual memory}. 
Book Keeping

• If you make changes to the default CSpace, you need to bookkeep them.

• Where do you put the book-keeping? Virtual memory.

• But where do you get/put the capabilities for the virtual memory?
Book Keeping

• If you make changes to the default CSpace, you need to bookkeep them.

• Where do you put the book-keeping? Virtual memory.

• But where do you get/put the capabilities for the virtual memory? Make changes to the capability space.
Book Keeping

- If you make changes to the default CSpace, you need to bookkeep them.
- But where do you get/put the capabilities for the virtual memory? Make changes to the capability space.
Capability Allocation
Problem

- Given the current state of the system, can I derive:
  - The capability I want, which may involve generating many extra capabilities (eg. 128MB cap -> 8M x 16B IPC cap.)
  - Enough CNode caps to store all of the above.
  - Enough Page table capabilities, to allocate memory to store the book-keeping changes above.
  - A plan to execute the changes to the CSpace and VSpace
Capability Deallocation Problem

• If I revoke a capability
  • Will a CNode become empty?
    • With the exception of the CNode cap itself.
  • Will the book-keeping become empty
    • With the exception of the CNode and VSpace caps
  • Can I make a plan for executing these changes
Capability Alloc / Dealloc

• I spent 18 months on this problem

• Had to write a constraint solver that would run in static memory (stack only).

• “Aurora” project. More info available.
Conclusion

• seL4 makes some really sensible decisions about how to make and manage caps.

• These turn out to be very hard to program against

• Please write programs against your API before you build your OS.
For more info

- https://github.com/seL4/seL4
- https://sel4.systems/
Backups
IPC in seL4

• Dirty (clever) secret, there are no “syscalls” in seL4, they are all IPCs.

• IPCs are sent to an endpoint. Which is a capability that may be received in the kernel or by another application

• IPCs are done with a “rendezvous” style, one process must call send the other must call read.

• Async IPC is a 32 bit register with bits that may be flipped.

• Async IPC is used for delivering interrupts.