2022 T2 Week 01 Part 1

Introduction: Microkernels and seL4

@GernotHeiser
Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:
    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/4.0/legalcode
Why Advanced Operating Systems?

• Understand OS (especially microkernels) in real depth
• Understand how to design an OS
• Learn to build a sizable system with great deal of independence
• Learn to cope with the complexity of systems code
• Tackle a real challenge
• Get a glimpse of OS research, and preparation for it
• Obtain skills highly sought-after in industry
• Have fun while working hard!
Today’s Lecture

• Whirlwind intro to microkernels and the context of seL4
• seL4 principles and concepts
• seL4 Mechanisms
  • IPC and Notifications

Aim: Get you ready for the project quickly
Microkernels
Microkernels: Reducing the Trusted Computing Base

• Idea of microkernel:
  • Flexible, minimal platform
  • Mechanisms, not policies
  • OS functionality provided by usermode servers
  • Servers invoked by kernel-provided message-passing mechanism (IPC)
  • Goes back to Nucleus [Brinch Hansen’70]

IPC performance is critical!
Monolithic vs Microkernel OS Evolution

**Monolithic OS**

- New features add code kernel
- New policies add code kernel
- Kernel complexity grows

**Microkernel OS**

- Features add usermode code
- Policies replace usermode code
- Kernel complexity is stable

**COMP9242 2022 T2 W01 Part 1: Introduction to seL4**
Microkernel Principle: Minimality

- Small trusted computing base
  - Easier to get right
  - Small attack surface
- Challenges:
  - API design: generality despite small code base
  - Kernel design and implementation for high performance

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. [Lietdke SOSP’95]
L4: 30 Years High-Performance Microkernels

First L4 kernel with capabilities

API Inheritance
Code Inheritance

L3 → L4
L4/MIPS
L4/Alpha

“X”
Hazelnut

Pistachio

L4-embed.
OKL4 μKernel

seL4
OKL4 Microvisor

UNSW/NICTA
GMD/IBM/Karlsruhe
Dresden
OK Labs
Commercial Clone

Fiasco

Fiasco.OC

NOVA

UNSW/IBM/Karlsruhe
Dresden
Commercial Clone

P4 → PikeOS

iOS secure enclave

Qualcomm modem chips

8
COMP9242 2022 T2 W01 Part 1: Introduction to seL4
© Gernot Heiser 2019 – CC BY 4.0
The seL4 Microkernel
Principles

• Single protection mechanism: capabilities
  • Now also for time: MCS configuration [Lyons et al, EuroSys’18]
• All resource-management policy at user level
  • Painful to use
  • Need to provide standard memory-management library
    • Results in L4-like programming model
• Suitable for formal verification
  • Proof of implementation correctness
  • Attempted since ‘70s
  • Finally achieved by L4.verified project at NICTA [Klein et al, SOSP’09]

More on principles in my blog: https://bit.ly/34uI8FI
Concepts in a Slide

- Capabilities (Caps): reference kernel objects
- 10 kernel object types:
  - Threads (thread-control blocks: TCBs)
  - Scheduling contexts (SCs)
  - Address spaces (page table objects: PDs, PTs)
  - Endpoints (IPC)
  - Reply objects (ROs)
  - Notifications
  - Capability spaces (CNodes)
  - Frames
  - Interrupt objects (architecture specific)
  - Untyped memory
- System calls:
  - Call(), ReplyRecv() (and one-way variants)
  - Yield()
Not a Concept: Hardware Abstraction

Why?
• Hardware abstraction violates minimality
• Hardware abstraction introduces policy

True microkernel:
• Minimal wrapper of hardware, just enough to safely multiplex
• “CPU driver” [Charles Gray]
• Similarities with Exokernels [Engeler ’95]
What Are (Object) Capabilities?

Capability = Access Token: Prima-facie evidence of privilege

Obj reference
Access rights

Eg. thread, address space

Object

Capabilities provide:
• Fine-grained access control
• Reasoning about information flow

Any system call is invoking a capability:
err = cap.method( args );

Eg. read, write, send, execute…
seL4 Capabilities

- Stored in cap space (CSpace)
  - Kernel object made up of CNodes
  - each an array of cap “slots”
- Inaccessible to userland
  - But referred to by pointers into CSpace (slot addresses)
  - These CSpace addresses are called CPTRs
- Caps convey specific privilege (access rights)
  - Read, Write, Execute, GrantReply (Call), Grant (cap transfer)
- Can invoke a cap or derive cap of less or equal strength
  - Details later
seL4 Mechanisms

IPC & Notifications
Protected Procedure Calls (IPC)

Fundamental microkernel operation
- Kernel provides no services, only mechanisms
- OS services provided by (protected) user-level server processes
- Invoked by protected procedure call (called “IPC” for historical reasons)

seL4 IPC uses a handshake through Endpoints:
- Transfer points without storage capacity
- Message must be transferred instantly
  - Single-copy user ➔ user by kernel
seL4 IPC: Cross-Domain Invocation

seL4 IPC is **not**:
- A mechanism for shipping data
- A synchronisation mechanism
  - side effect, not purpose

seL4 IPC is **is**:
- A user-controlled context switch “with benefits”:
  - change protection context
  - pass arguments / result

```
... err = server.f( args );
...

Server
f( args ) {
...
}
```
IPC: Endpoints

- Involves 2 threads, but always one blocked
- Logically, thread moves between address spaces

- Threads must rendez-vous
  - One side blocks until the other is ready
  - Implicit synchronisation

- Arguments copied from sender’s to receiver’s message registers
  - Combination of caps (by reference arguments) and data words (by value)
    - Presently max 121 words (484B, incl message “tag”)
    - Should never use anywhere near that much!
Endpoints are Message Queues

EP has no sense of direction
May queue senders or receivers
never both at the same time!
Communication needs 2 EPs!

Note: On single core should not get queues – server should be highest priority!

But: Reasonable for single-threaded (“passive”) server on multicore!
Server Invocation & Return

- Asymmetric relationship:
  - Server widely accessible, clients not
  - How can server reply back to client (distinguish between them)?

- Client can pass session cap in first request
  - server needs to maintain session state
  - forces stateful server design

- seL4 solution: Kernel creates channel in *reply object* (RO)
  - server provides RO in `ReplyRecv()` operation
  - kernel blocks client on RO when executing receive phase
  - server invokes RO for send phase (only one send until refreshed)
  - only works when client invokes with `Call()`
Call Semantics

Priorities:
- Call to high
- Receive from low!

One per client for blocking calls!

Client

Call(ep, args)

deliver to server
block client on RO

Kernel

process

deliver to client

Server

ReplyRecv(ro, ep, &args)

ReplyRecv(ro, ep, &args)
Stateful Servers: Identifying Clients

- Server must respond to correct client
  - Ensured by reply cap
- Must associate request with correct state
- Could use separate EP per client
  - Endpoints are lightweight (16 B)
  - But would require mechanism to wait on a set of EPs (like Unix select())
- Instead, seL4 allows to individually mark (“badge”) caps to same EP
  - Server provides individually badged (session) caps to clients
    - Separate endpoints for opening session, further invocations
    - Server tags client state with badge
    - Kernel delivers badge to receiver on invocation of badged caps
IPC Mechanics: Virtual Registers

• Like physical registers, virtual registers are thread state
  • context-switched by kernel
  • map to physical registers or thread-local memory ("IPC buffer")
• Message registers
  • contain message transferred in IPC
  • architecture-dependent subset mapped to physical registers
    • presently 1 on x86, 4 on x64, Arm, RISC-V
    • library interface hides details
  • 1\textsuperscript{st} transferred word is special, contains message tag
• API: MR[0] refers to next word (not the tag!)
IPC Operations Summary

- **Call** \((ep\_cap, ...)\)
  - *Atomic*: guarantees caller is ready to receive reply
  - Sets up server’s reply object
- **ReplyRecv** \((ep\_cap, ...)\)
  - Invokes RO (non-blocking), waits on EP, re-inits RO
- **Recv** \((ep\_cap, ...), Reply(...), Send (ep\_cap, ...)\)
  - For initialisation and exception handling
  - needs Read, Write, Write permission, respectively
- **NBSend** \((ep\_cap, ...)\)
  - Polling send, message lost if receiver not ready

No failure notification where this reveals info on other entities!
Notifications – Synchronisation Objects

- Logically, a Notification is an array of binary semaphores
  - Multiple signalling, select-like wait
  - Not a message-passing IPC operation!

- Implemented by `data word` in Notification
  - Send OR`s sender’s `cap badge` to data word
  - Receiver can poll or wait
    - waiting returns and clears data word
    - polling just returns data word

```
Thread1
Running  Blocked

w = Poll(not_cap, ...)

Signal(not_cap, ...)

Thread2
Blocked  Running

...... w = Wait(not_cap,...)

Signal(not_cap, ...)
```
Notification Queues

Process₁

Process₂

Kernel
TCB₁
TCB₂
Notification

0 0 0 0 0 ... 0

Further waiters queued by priority
First invocation queues waiter
Receiving from EP and Notification

Server with synchronous and asynchronous interface

- Synchronous RPC protocol
- Asynchronous completion signals

- Client
- File Server
- Device Driver

Better: single thread for both interfaces
- Notification “bound” to TCB
- Signal delivered as “IPC” from EP

Concurrency control, complexity!

Separate thread per interface?

Must partition badge space to distinguish!
IPC Message Format

- Raw data
- Semantics defined by IPC protocol (Kernel or user)
- Bitmap indicating caps which had badges extracted
- Caps sent or received

<table>
<thead>
<tr>
<th>Tag</th>
<th>Message</th>
<th>Caps (on Send)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Badges (on Receive)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Label</th>
<th>Caps unwrapped</th>
<th># Caps</th>
<th>Msg Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Client-Server IPC Example

Client

```c
seL4_MessageInfo_t tag = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_SetMR(0, value);
seL4_Call(server_c, tag);
```

Server

```c
ut_t* reply_ut = ut_alloc(seL4_ReplyBits, &cspace);
seL4_CPtr reply = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, reply_ut->cap, reply,
                              seL4_ReplyObject, seL4_ReplyBits);
seL4_CPtr badged_ep = cspace_alloc_slot(&cspace);
cspace_mint(&cspace, badged_ep, &cspace, ep, seL4_AllRights, 0xff);
...
seL4_Word badge;
seL4_MessageInfo_t msg = seL4_Recv(ep, &badge, reply);
...
seL4_MessageInfo_t response = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_NBSend(reply, response);
```

Note: this is for clarity, in reality should use ReplyRecv!
Proper Server Loop

```c
... while (1) {
    seL4_MessageInfo_t msg = seL4_ReplyRecv(ep, response, &badge, reply);
    ...
    seL4_MessageInfo_t response = seL4_MessageInfo_new(0, 0, 0, 1);
} 
```