Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 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/3.0/legalcode
seL4 Mechanisms

Capabilities
Derived Capabilities

- Badging is an example of *capability derivation*
- The *Mint* operation creates a new, less powerful cap
  - Can add a badge: Mint (_capability, ▼) ➞ *
  - Can strip access rights, eg RW→R/O
- *Granting* transfers caps over an Endpoint
  - Delivers copy of sender’s cap(s) to receiver
  - Sender needs Endpoint cap with Grant permission
  - Receiver needs Endpoint cap with Write permission
    - else Write permission is stripped from new cap
- *Retyping*: fundamental memory management operation
  - Details later…

Remember: Caps are kernel objects!
Capability Derivation

Copy, Mint, Mutate, Revoke are invoked on CNodes

Mint(        , dest, src, rights, ▼ )

Copy takes a CNode cap as destination
- Allows copying between CSpaces
- Alternative to IPC cap transfer

CNode cap must allow modification
seL4 System Calls

- seL4 has 11 syscalls:
  - `Yield()`: invokes scheduler
    - doesn’t require a capability!
  - `Send()`, `Recv()` and variants/combinations thereof
    - `Call()`, `ReplyRecv()`
    - `Send()`, `NBSend()`
    - `Recv()`, `NBRecv()`, `NBSendRecv()`
    - `Wait()`, `NBWait()`, `NBSendWait()`
  - `Call()` is atomic `Send()` + reply-object setup + `Wait()`
    - cannot be simulated with one-way operations!
  - `ReplyRecv()` is `NBSend()` + `Recv()`

That’s why I earlier said “approximately 3” 😁
seL4 System Calls

- Endpoints support all 10 IPC variants
- Notifications support:
  - NBSend() – aliased as Signal()
  - Wait()
  - NBWait() – aliased as Poll()
- Other objects only support Call()
  - Appear as (kernel-implemented) servers
  - Each has a kernel-defined protocol
    - operations encoded in message tag
    - parameters passed in message words
  - Mostly hidden behind “syscall” wrappers
seL4 Memory-Management Principles

• Memory (and caps referring to it) is typed:
  • Untyped memory:
    • unused, free to Retype into something else
  • Frames:
    • (can be) mapped to address spaces, no kernel semantics
  • Rest: TCBs, address spaces, CNodes, EPs, …
    • used for specific kernel data structures

• After startup, kernel never allocates memory!
  • All remaining memory made Untyped, handed to initial address space

• Space for kernel objects must be explicitly provided to kernel
  • Ensures strong resource isolation

• Extremely powerful tool for shooting oneself in the foot!
  • We hide much of this behind the cspace and ut allocation libraries
## CSpace Operations

```c
int cspace_create_two_level(cspace_t *bootstrap, cspace_t *target, cspace_alloc_t cspace_alloc);
int cspace_create_one_level(cspace_t *bootstrap, cspace_t *target);
void cspace_destroy(cspace_t *c);
seL4_CPtr cspace_alloc_slot(cspace_t *c);
void cspace_free_slot(cspace_t *c, seL4_CPtr slot);

seL4_Error cspace_copy(cspace_t *dest, seL4_CPtr dest_cptr, cspace_t *src,
                        seL4_CPtr src_cptr, seL4_CapRights_t rights)

cspace_delete(cspace_t *cspace, seL4_CPtr cptr)
seL4_Error cspace_mint(cspace_t *dest, seL4_CPtr dest_cptr, cspace_t *src,
                       seL4_CPtr src_cptr, seL4_CapRights_t rights, seL4_Word badge)

cspace_move(cspace_t *dest, seL4_CPtr dest_cptr, cspace_t *src, seL4_CPtr src_cptr)
seL4_Error cspace_mutate(cspace_t *dest, seL4_CPtr dest_cptr, cspace_t *src,
                         seL4_CPtr src_cap, seL4_Word badge)

cseL4_Error cspace_revoke(cspace_t *cspace, seL4_CPtr cptr)
seL4_Error cspace_save_reply_cap(cspace_t *cspace, seL4_CPtr cptr)
seL4_Error cspace_irq_control_get(cspace_t *dest, seL4_CPtr cptr, seL4_IRQControl irq_cap, int irq, int level)
seL4_Error cspace_untyped_retype(cspace_t *cspace, seL4_CPtr ut, seL4_CPtr target,
                                  seL4_Word type, size_t size_bits);
```
Project: cspace and ut libraries

- OS Personality
  - Library Calls
    - ut_alloc()
    - ut_free()
    - cspace_create()
    - cspace_destroy()
  - System Calls
  - sel4
  - sel4

Manages ≤4KiB Untyped
Extend for own needs!
Wraps messy Cspace tree & slot management

COMP9242 2020T2 W01b
© Gernot Heiser 2019 – CC Attribution License
seL4 Mechanisms
Address Spaces and Memory Management
seL4 Memory Management Approach

- Resources fully delegated, allows autonomous operation
- Strong isolation, No shared kernel resources

init Task = Global Resource Manager
Memory Management Mechanics: Retype
seL4 Address Spaces (VSpaces)

• Very thin (arch-dependent) wrapper of hardware page tables
  • Arm & x86 similar (32-bit 2-level, 64-bit 4–5 level)

• Arm 64-bit ISA (AARCH64):
  • page global directory (PGD)
  • page upper directory (PUD)
  • page directory (PD)
  • page table (PT)

• PGD object represents VSpace:
  • Creating a PGD (by Retype) creates the VSpace
  • Deleting PGD deletes VSpace
Address Space Operations

```
seL4_Word paddr = 0;
ut_t *ut = ut_alloc_4k_untyped(&paddr);
seL4_CPtr frame = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, ut->cap, frame,
  seL4_ARM_SmallPageObject, seL4_PageBits);
err = map_frame(&cspace, frame, pgd_cap, 0xA0000000,
  seL4_AllRights, seL4_Default_VMAttributes);
```

Cap to top-level page table

Poor API choice!

Each mapping has:
- `virtual_address`, `phys_address`, `address_space` and `frame_cap`
- `address_space` struct identifies the level 1 page directory cap
- you need to keep track of `(frame_cap, PD_cap, v_adr, p_adr)`!
Multiple Frame Mappings: Shared Memory

```c
seL4_CPtr new_frame_cap = cspace_alloc_slot(&cspace);
seL4_Error err = cspace_copy(&cspace, new_frame_cap,
                        &cspace, frame, seL4_AllRights);
err = map_frame(&cspace, new_frame_cap, pgd_cap, 0xA0000000,
                seL4_AllRights, seL4_Default_VMAtributes);

seL4_ARCH_Page_Unmap(frame);
cspace_delete(&cspace, frame);
cspace_free_slot(&cspace, frame);
seL4_ARCH_Page_Unmap(new_frame_cap);
cspace_delete(&cspace, new_frame_cap);
cspace_free_slot(&cspace, new_frame_cap);
ut_free(&ut, seL4_PageBits);
```

Each mapping requires its own frame cap even for the same frame!
Memory Management Caveats

• The UT table handles allocation for you

• But: very simple buddy-allocator:
  • Freeing an object of size \( n \) ⇒ can allocate new objects ≤ size \( n \)
  • Freeing 2 objects of size \( n \) ≠ can allocate an object of size \( 2n \).

<table>
<thead>
<tr>
<th>Object</th>
<th>Size (B)</th>
<th>Align (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>( 2^{12} )</td>
<td>( 2^{12} )</td>
</tr>
<tr>
<td>PT/PD/PUD/PGD</td>
<td>( 2^{12} )</td>
<td>( 2^{12} )</td>
</tr>
<tr>
<td>Endpoint</td>
<td>( 2^4 )</td>
<td>( 2^4 )</td>
</tr>
<tr>
<td>Notification</td>
<td>( 2^5 )</td>
<td>( 2^5 )</td>
</tr>
<tr>
<td>Scheduling Context</td>
<td>≥ ( 2^8 )</td>
<td>( 2^8 )</td>
</tr>
<tr>
<td>Cslot</td>
<td>( 2^4 )</td>
<td>( 2^4 )</td>
</tr>
<tr>
<td>Cnode</td>
<td>≥ ( 2^{12} )</td>
<td>( 2^{12} )</td>
</tr>
<tr>
<td>TCB</td>
<td>( 2^{11} )</td>
<td>( 2^{11} )</td>
</tr>
</tbody>
</table>

Values for AARCH64
Memory-Management Caveats

- Objects are allocated by Retype() of Untyped memory.
- The kernel will not allow you to overlap objects.
- ut_alloc and ut_free() manage user-level view of allocation.
  - Major pain if kernel and user view diverge.
  - TIP: Keep objects address and CPtr together!

- Be careful with allocations!
- Don’t try to allocate all of physical memory as frames, you need more memory for TCBs, endpoints etc.
- Your frametable will eventually integrate with ut_alloc to manage the 4KiB untyped size.

Untyped Memory 2^{15} B

But debugging nightmare if you try!!
seL4 Mechanisms

Threads
Threads

- Threads are represented by TCB objects
- They have a number of attributes (recorded in TCB):
  - VSpace: a virtual address space, can be shared by multiple threads
  - CSpace: capability storage
  - Fault endpoint and timeout endpoint
  - IPC buffer (backing storage for virtual registers)
  - stack pointer (SP), instruction pointer (IP), general-purpose registers
  - Scheduling priority and maximum controlled priority (MCP)
  - Scheduling context: right to use CPU time
- These must be explicitly managed – we provide examples
Threads

Creating a thread:

- Obtain a TCB object
- Set attributes: `Configure()`
  - associate with VSpace, CSpace, fault EP, define IPC buffer
- Set scheduling parameters
  - priority, scheduling context, timeout EP (maybe MCP)
- Set SP, IP (and optionally other registers): `WriteRegisters()`

Thread is now initialised

- if `resume_target` was set in call thread is runnable
- else activate with `Resume()`
Creating a Thread in Own AS and Cspace

```c
static char stack[100];
int thread_fct() {
    while(1);
    return 0;
}

ut_t *ut = ut_alloc(seL4_TCBBits, &cspace);
seL4_CPtr tcb = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, ut->cap, tcb, seL4_TCBObject, seL4_TCBBits);

err = seL4_TCB_Configure(tcb, cspace.root_cnode, seL4_NilData, seL4_CapInitThreadVSpace,
                          seL4_NilData, PROCESS_IPC_BUFFER, ipc_buffer_cap);
if (err != seL4_NoError) return err;

err = seL4_TCB_SetSchedParams(tcb, seL4_CapInitThreadTCB, seL4_MinPrio,
                               TTY_PRIORITY, sched_context, fault_ep);
```

Tip: If you use threads, write a library for create/destroy!
Threads and Stacks

• Stacks are completely user-managed, kernel doesn’t care!
  • Kernel only preserves SP, IP on context switch
• Stack location, allocation, size must be managed by userland
• Beware of stack overflow!
  • Easy to grow stack into other data
    • Pain to debug!
  • Take special care with automatic arrays!

```c
f() {
  int buf[10000];
  ...
}
```
Creating a Thread in New AS and CSpace

/* Allocate, retype and map new frame for IPC buffer as before
 * Allocate and map stack??
 * Allocate and retype a TCB as before
 * Allocate and retype a PageGlobalDirectoryObject of size seL4_PageDirBits
 * Mint a new badged cap to the syscall endpoint
 */
cspace_t * new_cspace = ut_alloc(seL4_TCBBits);

char *elf_base = cpio_get_file(_cpio_archive, app_name, &elf_size);
seL4_Word sp = init_process_stack(&cspace, new_pgd_cap, elf_base);
err = elf_load(&cspace, seL4_CapInitThreadVSpace, tty_test_process.vspace, elf_base);
err = seL4_TCB_Configure(tcb, new_cspace.root_cnode, seL4_NilData, new_pgd_cap
                      seL4_NilData, PROCESS_IPC_BUFFER, ipc_buffer_cap);

seL4_UserContext context = {
    .pc = elf_getEntryPoint(elf_base),
    .sp = sp,
};
err = seL4_TCB_WriteRegisters(tty_test_process.tcb, 1, 0, 2, &context);
seL4 Scheduling

• 256 hard priorities (0–255), strictly observed
  • The scheduler will always pick the highest-prio runnable thread
  • Round-robin within priority level
  • Kernel will never change priority (but can do with syscall)

• Thread with no scheduling context or no budget is not runnable
  • SC contains *budget*: when exhausted, thread removed from run queue
  • SC contains *period*: specifies when budget is replenished
  • Budget = period: Operates as a time slice

Aim is real-time performance, not fairness!
  • Can implement fair policy at user level
seL4 Mechanisms

Interrupts and Exceptions
Exception Handling

Exception types:
• invalid syscall
  • eg for instruction emulation, virtualisation
• capability fault
  • cap lookup failed or found invalid cap
• page fault
  • address not mapped
  • maybe invalid address
  • maybe grow stack, heap, load library…
• architecture-defined
  • divide by zero, unaligned access, …
• timeout
  • scheduling context out of budget

On exception:
• message sent to fault endpoint
• pretends to be from faulter
• replying will restart thread

has its own fault endpoint
Interrupt Management

2 special objects for managing and acknowledging interrupts:

• Single IRQControl object
  • single IRQControl cap provided by kernel to initial VSpace
  • only purpose is to create IRQHandler caps

• Per-IRQ-source IRQHandler object
  • interrupt association and dissociation
  • interrupt acknowledgment
  • edge-triggered flag
Interrupt Handling

IRQHandler cap allows driver to bind Notification to interrupt
- Notification is used to receive interrupt
- IRQHandler is used to acknowledge interrupt

```c
seL4_CPtr irq = cspace_alloc_slot(&cspace);
seL4_Error err = cspace_irq_control_get(&cspace, irq, seL4_CapIRQControl, irq_number, true_if_edge_triggered);
seL4_IRQHandler_SetNotification(irq, ntfn);
seL4_IRQHandler_Ack(irq);
```
Device Drivers

• In seL4 (and all other L4 kernels) drivers are usermode processes

• Drivers do three things:
  • Handle interrupts (already explained)
  • Communicate with rest of OS (IPC + shared memory)
  • Access device registers

• Device register access (ARM uses memory-mapped IO)
  • Have to find frame cap from bootinfo structure
  • Map the appropriate page in the driver’s VSpace

```c
device_vaddr = sos_map_device(&cspace, 0x0A000000, BIT(seL4_PageBits));
...
*((void *) device_vaddr) = ...;
```
Project Platform: ODROID-C2

ODROID-C2 Board

Armlogic S905 SoC

- ARMv8 Cortex-A53
- ARMv8 Cortex-A53
- ARMv8 Cortex-A53
- ARMv8 Cortex-A53

Serial
Timer
Ethernet
Other...

Serial connector
Ethernet connector

2 GiB Memory

seL4_DebugPutChar()

M0: serial over LAN for userlevel apps
M6: Network File System (NFS)
in the Real World (Courtesy Boeing, DARPA)