COMP9242 Advanced OS

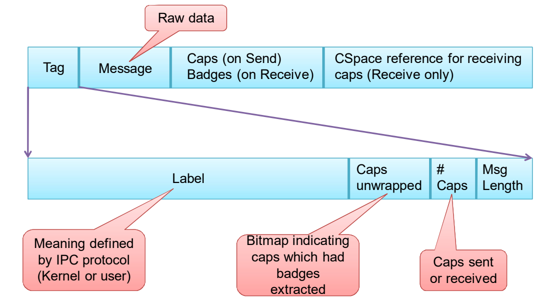
T2/2019 W01: **Introduction to seL4**

@GernotHeiser

Never Stand Still | Engineering | Computer Science and Engineering

Copyright Notice

IPC Message Format



Note: Don't need to deal with this explicitly for project

Client-Server IPC Example

Message length is 1

Client

Load into tag register

Set message register #0

```
seL4_MessageInfo_t tag = seL4_MessageInfo_new(0, 0, 0,1);
seL4_SetTag(tag);
seL4_SetMR(0,1);
seL4_Call(server_c, tag);
```

Server

Allocate slot & retype to EP

```
ut_t* ut = ut_alloc(seL4_EndpointBits, &cspace);
seL4_CPtr ep = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, ut->cap, ep,
                  seL4_EndpointObject, seL4_EndpointBits);
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(cptr, &badge);
…
seL4_MessageInfo_t reply = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_Reply(reply);
```

Mint cap with badge `0xff`

Implicit use of reply cap

Server Saving Reply Cap

Server

```
ut_t *ut = ut_alloc(seL4_EndpointBits, &cspace);
seL4_CPtr ep = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, ut->cap, ep,
                  seL4_EndpointObject, seL4_EndpointBits);
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(cptr, &badge);
seL4_CPtr reply_cap = cspace_alloc_slot(&cspace);
cspace_save_reply_cap(&cspace, reply_cap);
…
seL4_MessageInfo_t reply = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_Send(reply_cap, reply);
cspace_free_slot(&cspace, reply);
```

Save reply cap in CSpace

Explicit use of reply cap

Reply cap no longer valid

Derived Capabilities

- Badging is an example of *capability derivation*
- The `Mint` operation creates a new, less powerful cap
  - Can add a badge
    - `Mint` ( ) →
  - Can strip access rights
    - eg WR→R/O
- *Granting* transfers caps over an Endpoint
  - Delivers copy of sender's cap(s) to receiver
    - reply caps are a special case of this
  - Sender needs Endpoint cap with `Grant` permission
  - Receiver needs Endpoint cap with `Write` permission
    - else `Write` permission is stripped from new cap
- *Retyping*
  - Fundamental operation of seL4 memory management
  - Details later…

Remember, caps are kernel objects!

seL4 System Calls

- Notionally, seL4 has 6 syscalls:
  - `Yield`(): invokes scheduler
    - only syscall which doesn't require a cap!
  - `Send`(), `Recv`() and 3 variants/combinations thereof
    - `Signal`() is actually not a separate syscall but same as `Send`()
  - This is why I earlier said "approximately 3 syscalls" ☺
- All other kernel operations are invoked by "messaging"
  - Invoking `Call`() on an object cap
    - Logically sending a message to the kernel
  - Each object has a set of kernel protocols
    - operations encoded in message tag
    - parameters passed in message words
  - Mostly hidden behind "syscall" wrappers

Will change soon

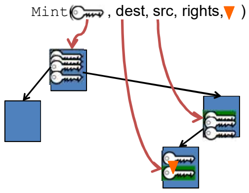
seL4 Memory-Management Principles

- 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

Capability Derivation

- `Copy`, `Mint`, `Mutate`, `Revoke` are invoked on CNodes

`Mint`(, dest, src, rights,)



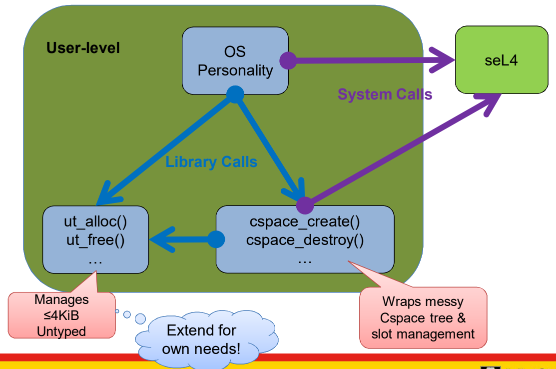
  - CNode cap must provide appropriate rights
- `Copy` takes a cap for destination
  - Allows copying of caps between Cspaces
  - Alternative to granting via IPC (if you have privilege to access Cspace!)

Cspace Operations

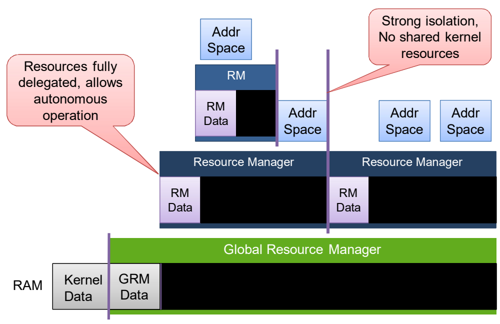
```
int cspace_create_two_level(cspace_t *bootstrap, cspace_t
*target, cspace_alloc_t cspace_alloc);
int cspace_create_one_level(cspace_t *bootstrap, cspace_t
*target);
void cspace_destroy(cspace_t *c);
seL4_CPtr cspace_alloc_slot(cspace_t *c);
void cspace_free_slot(cspace_t *c, seL4_CPtr slot);
```

```
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)
seL4_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);
```

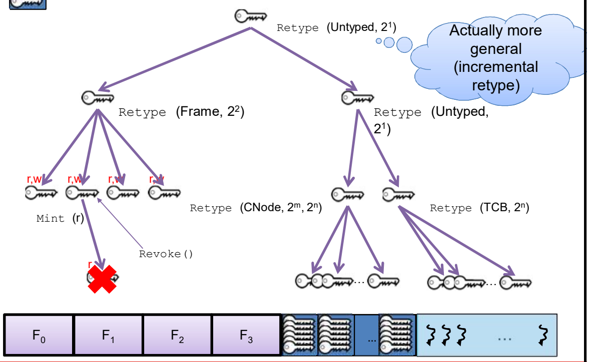

`cspace` and `ut` libraries



seL4 Memory Management Approach

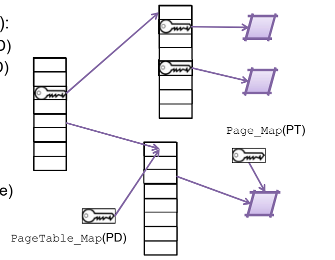


Memory Management Mechanics: `Retype`



seL4 Address Spaces (VSpaces)

- Very thin wrapper around hardware page tables
  - Architecture-dependent
  - 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)
- A VSpace is represented by a PGD object:
  - Creating a PGD (by Retype) creates the VSpace
  - Deleting the PGD deletes the 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, fram
        seL4_ARM_SmallPageObject, seL4_PageBits);
err = map_frame(&cspace, frame, pgd_cap, 0x
        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`)!

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


Multiple Frame Mappings: Shared Memory

```
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_VMAttributes);
```

```
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
- A simple list-based allocator, you need to understand how it works:
  - Freeing an object of size *n*: you can allocate new objects <= size *n*
  - Freeing 2 objects of size *n* ***does not mean*** that you can allocate an object of size *2n*.

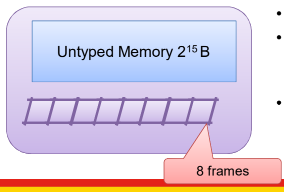
Object	Size (B), AARCH64	Alignment (B), AARCH64
Frame	2^{12}	2^{12}
PT/PD/PUD/PGD	2^{12}	2^{12}
Endpoint	2^4	2^4
Notification	2^5	2^5
Cslot	2^4	2^4
Cnode	$\geq 2^{12}$	2^{12}
TCB	2^{11}	2^{11}

Implementation choice!

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's view of Untyped allocation.
  - Major pain if kernel and user's view diverge
  - TIP: Keep objects address and CPtr together.

But debugging nightmare if you try!!



- Be careful with allocations!
- Don't try to allocate all of physical memory as frames, you need more memory for TCBs, endpoints etc.
- We provide a frametable that integrates with ut_alloc to manage the 4KiB untyped size.
  - You can modify as required

Threads

- Theads are represented by TCB objects
- They have a number of attributes (recorded in TCB):
  - VSpace: a virtual address space
    - page global directory (PGD) reference
    - multiple threads can belong to the same VSpace
  - CSpace: capability storage
    - CNode reference (CSpace root) plus a few other bits
  - *Fault endpoint*
    - Kernel sends message to this EP if the thread throws an exception
  - IPC buffer (backing storage for virtual registers)
  - stack pointer (SP), instruction pointer (IP), user-level registers
  - *Scheduling priority* and *maximum controlled priority (MCP)*
  - *Time slice length* (presently fixed)
- These must be explicitly managed
  - … we provide an example you can modify

Yes, this is broken!

Threads

Creating a thread
- Obtain a TCB object
- Set attributes: `Configure()`
  - associate with VSpace, CSpace, fault EP, prio, define IPC buffer
- Set scheduling parameters
  - priority (maybe MCP)
- Set SP, IP (and optionally other registers): `WriteRegisters()`
  - this results in a completely initialised thread
  - will be able to run if `resume_target` is set in call, else still inactive
- Activated (made schedulable): `Resume()`

Creating a Thread in Own AS and Cspace

```
static char stack[100];
int thread_fct() {
          while(1);
          return 0;
}
/* Allocate and map new frame for IPC buffer as before */
ut_t *ut = ut_alloc(seL4_TCBBits, &cspace);
/* alloc slot to retype into */
seL4_CPtr tcb = cspace_alloc_slot(&cspace);
/* retype */
err = cspace_untyped_retype(&cspace, ut->cap, tcb, seL4_TCBObject,
                                 seL4_TCBBits);
err = seL4_TCB_Configure(tcb, fault_ep, cspace.root_cnode, seL4_NilData,
                                   seL4_CapInitThreadVSpace, seL4NilData,
                                   PROCESS_IPC_BUFFER, ipc_buffer_cap);
err = seL4_TCB_SetPriority(tcb, seL4_CapInitThreadTCB, PRIORITY);
```

If you use threads, write a library to create and destroy them.

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!


```
f () {
   int
buf[10000];
    . . .
}
```


Creating a Thread in *New* AS and CSpace

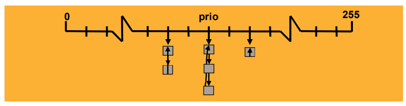
```
/* Allocate, retype and map new frame for IPC buffer as before
 * Allocate and map stack???
 * Allocate and retype a TCB  as before
 * Allocate and retype a  PageGlobalDirectoryObject of size seL4_PageDirBits
 * Mint a new badged cap to the syscall endpoint
 */
cspace_t * new_cpace = ut_alloc(seL4_TCBBits);

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, fault_ep, new_cspace.root_cnode, seL4_NilData,
                                new_pgd_cap, seL4NilData,
                                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

Better model in "MCS" branch – merge soon

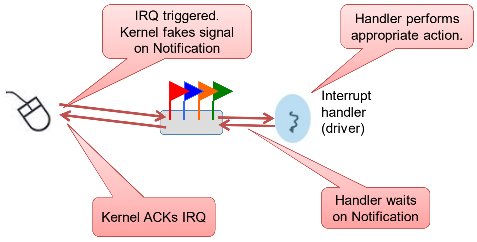
- Present seL4 scheduling model is fairly naïve
- 256 hard priorities (0–255)
  - Priorities are strictly observed
  - The scheduler will always pick the highest-prio runnable thread
  - Round-robin scheduling within prio level
- Aim is real-time performance, **not** fairness
  - Kernel itself will never change the prio of a thread
  - Achieving fairness (if desired) is the job of user-level servers



Exception Handling

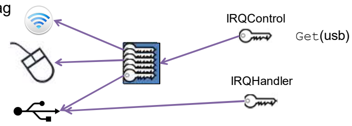
- A thread can trigger different kinds of exceptions:
  - invalid syscall
    - may require instruction emulation or result from virtualization
  - capability fault
    - cap lookup failed or operation is invalid on cap
  - page fault
    - attempt to access unmapped memory
    - may have to grow stack, grow heap, load dynamic library, …
  - architecture-defined exception
    - divide by zero, unaligned access, …
- Results in kernel sending message to fault endpoint
  - exception protocol defines state info that is sent in message
- Replying to this message restarts the thread
  - endless loop if you don't remove the cause for the fault first!

Interrupt Handling



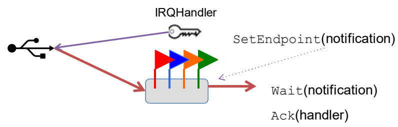
Interrupt Management

- seL4 models IRQs as messages sent to a Notification
  - Interrupt handler has Receive cap on that Notification
- 2 special objects used 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
- Afterwards:
  - Notification is used to receive interrupt
  - IRQHandler is used to acknowledge interrupt




```
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);
```

ACK to unmask IRQ

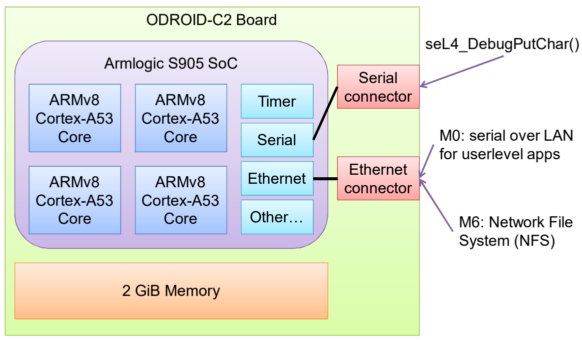
Device Drivers

- 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
  - Devices are memory-mapped on ARM
  - Have to find frame cap from bootinfo structure
  - Map the appropriate page in the driver's VSpace

```
device_vaddr = sos_map_device(&cspace, 0xA0000000, BIT(seL4_PageBits));
…
*((void *) device_vaddr= …;
```

Magic device register access

Project Platform: ODROID-C2



seL4 in the Real World (Courtesy Boeing, DARPA)