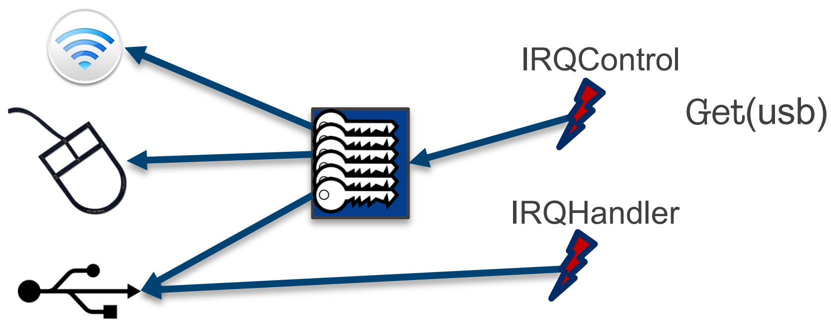
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2022 T2 Week 01 Part 2

Introduction: Using seL4

@GernotHeiser



Today's Lecture

- seL4 Mechanisms
  - Capabilities
  - Address spaces & memory management
  - Threads
  - Interrupts and Exceptions
- seL4 System Design Hints

Aim: You should then be ready to start the project

seL4 Mechanisms

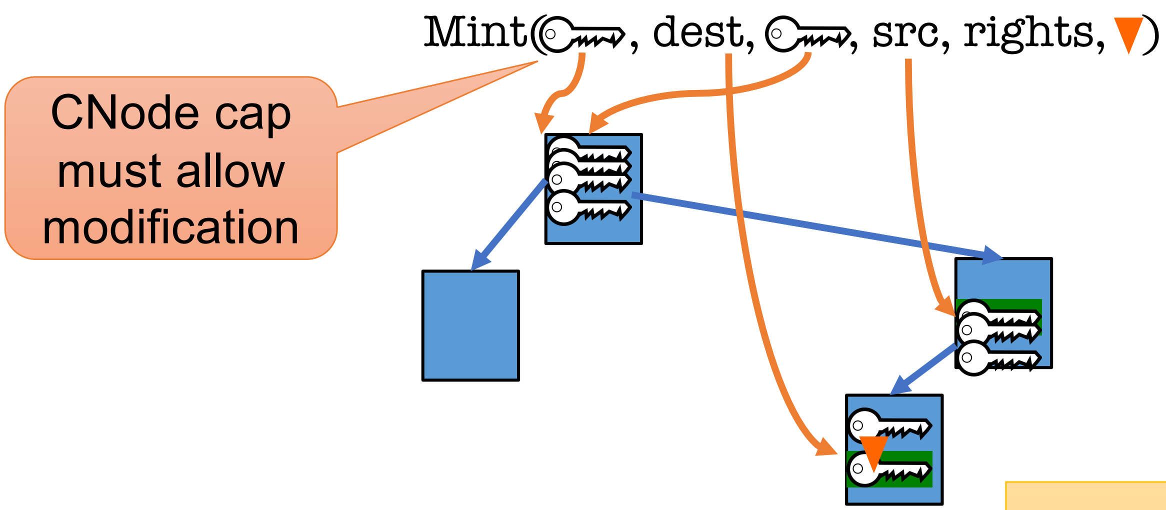
Capabilities

Derived Capabilities

- *Badging* is an example of *capability derivation*
- The *Mint* operation creates a new, less powerful cap
  - Can add a badge: `Mint` (,) ➞
  - 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

Copy takes a CNode cap as destination

- Allows copying between CSpaces
- Alternative to IPC cap transfer

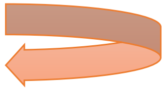
seL4 System Calls [1/3]

That's why I earlier said "approximately 3" 😀

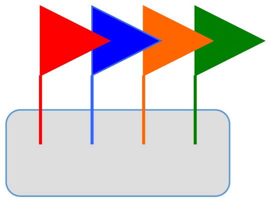
- seL4 has 11 syscalls:
  - `Yield()`: invokes scheduler
    - doesn't require a capability!
  - `Send()`, `Recv()` and variants/combinations thereof
    - `Call()`, `ReplyRecv()`
    - `Send()`, `NBSend()`
    - `Recv()`, `NBRecv()`, `NBSendRecv()`
    - `Wait()`, `NBWait()`, `NBSendWait()`
  - `Call()`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

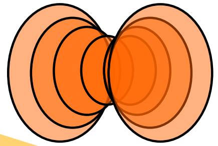
seL4 System Calls [2/3]

- Endpoints support all 10 Send/Receive variants
- ROs support:
  - NBSend ()
  - NBSendRecv()



- Notifications support:
  - NBSend() – aliased as Signal()
  - Wait()
  - NBWait() – aliased as Poll()

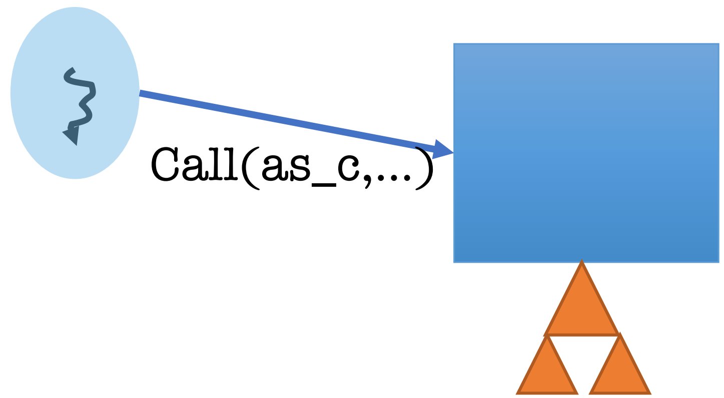




But remember, you should just use Call() and ReplyRecv()

seL4 System Calls [3/3]

- Endpoints support all 10 IPC variants
- ROs support `NBSend ()`, `NBSendRecv()`
- Notifications support `NBSend()`, `Wait()`, `NBWait`
- 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



Most of this is hidden behind "syscall" wrappers

seL4 Memory-Management Principles

- Memory (and caps referring to it) is *typed*:
  - *Untyped* memory:
    - unused, free to `Retype` into something useful
  - 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 gun for shooting yourself in the foot!
  - We hide much of this behind the *cspace* and *ut* allocation libraries

CSpace Operations

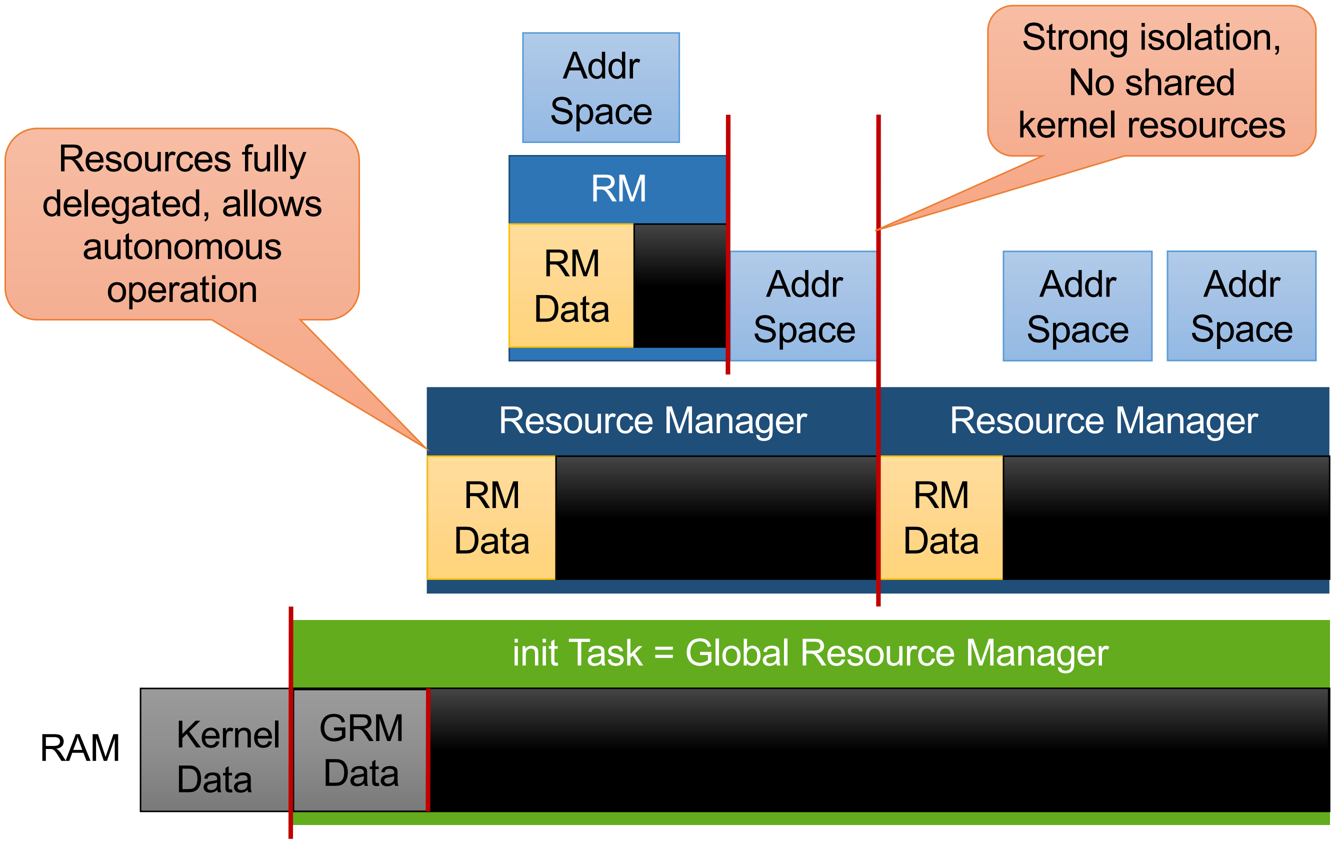
```
int cspace_create_two_level(cspace_t *bootstrap, cspace_t *target, cspace_alloc_t cspace_alloc);
int cspace_create_one_level(cspace_t *bootstrap, cspace_t *target);
void cspace_destroy(cspace_t *c);
seL4_CPtr cspace_alloc_slot(cspace_t *c);
void cspace_free_slot(cspace_t *c, seL4_CPtr slot);
```

```
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);
```

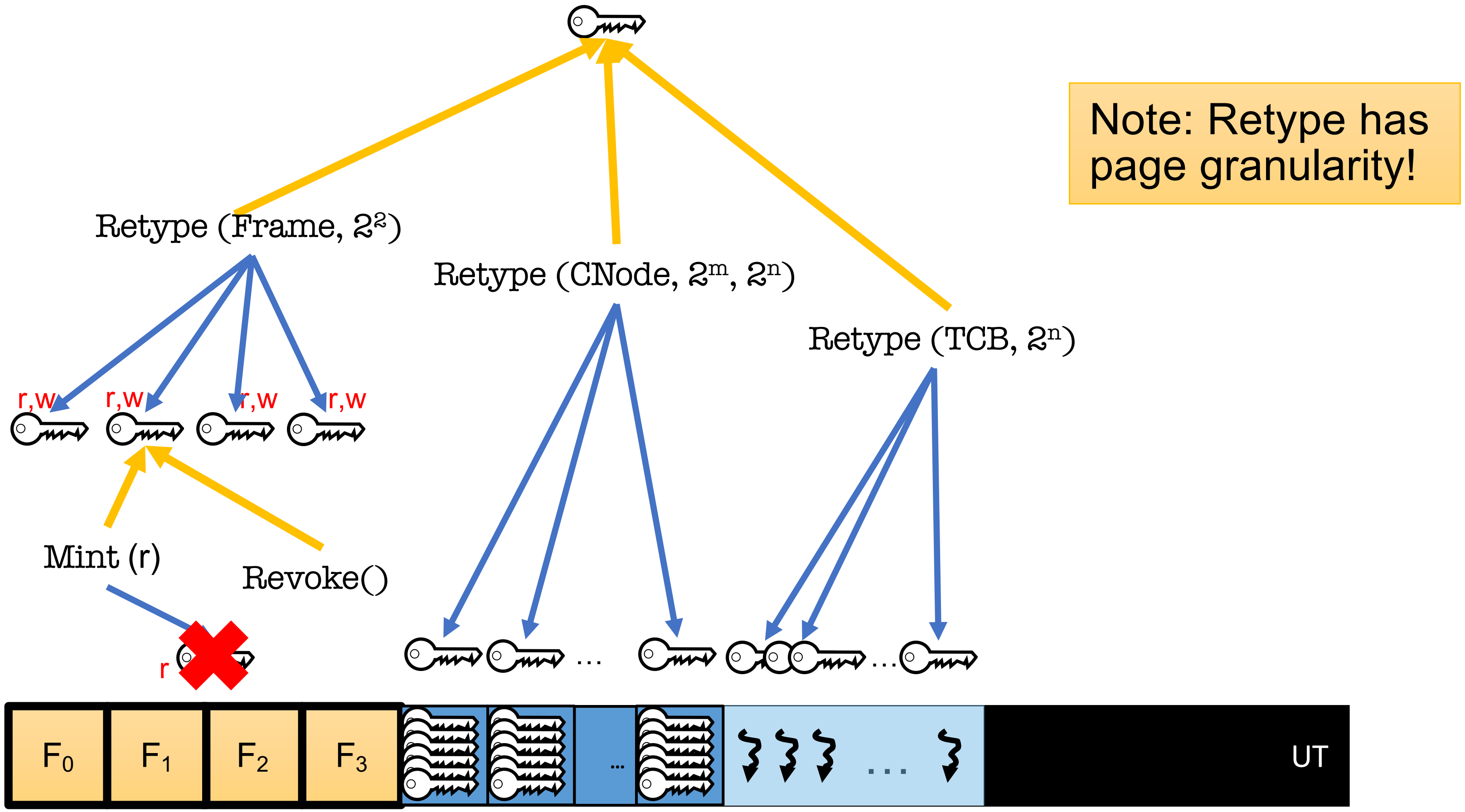
seL4 Mechanisms

Address Spaces and Memory Management

seL4 Memory Management Approach



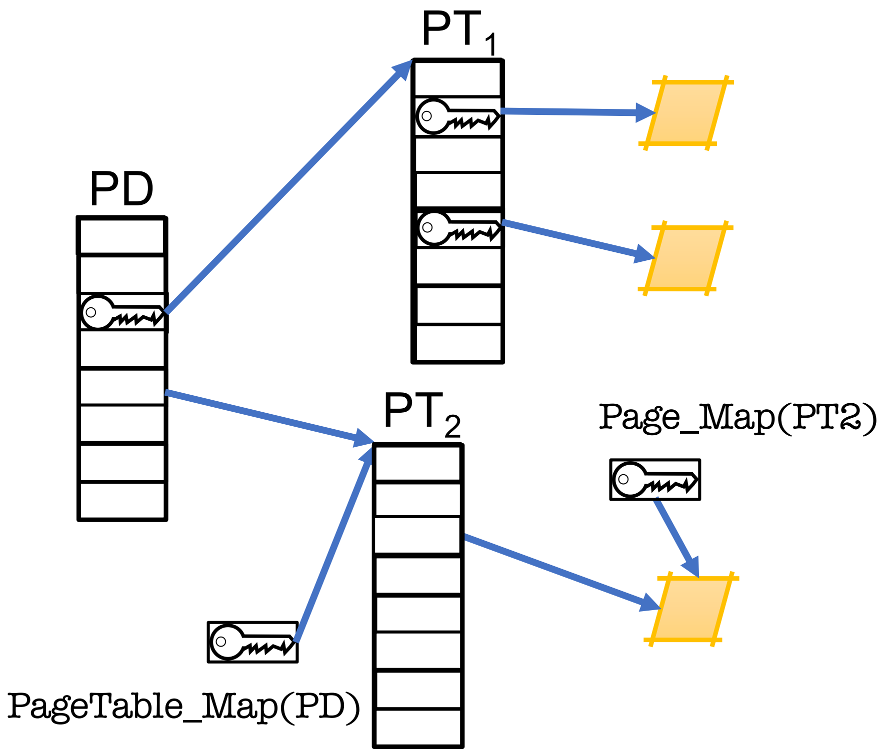
Memory Management Mechanics: Retype



UNSW SYDNEY

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

Poor API choice!

```
seL4_Word paddr = 0;
ut_t *ut = ut_alloc_4k_untyped(&p_addr);
seL4_CPtr frame = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, ut->cap, frame
                  seL4_ARM_SmallPageObject, seL4_
err = map_frame(&cspace, frame, pgd, v_addr,
                  seL4_AllRights, seL4_Default_VMAttributes);
```

Cap to top-level page table

Each frame 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, PD, v_addr, p_addr)!

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

Poor API choice!

Multiple Frame Mappings: Shared Memory

```
seL4_CPtr new_frame = cspace_alloc_slot(&cspace);
seL4_Error err = cspace_copy(&cspace, new_frame,
                             &cspace, frame, seL4_AllRights);
err = map_frame(&cspace, new_frame, pgd, new_v_addr,
                seL4_AllRights, seL4_Default_VMAttributes);
```

Allocate frame

Duplicate cap

Map frame

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

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

seL4 Mechanisms

Threads

Threads

- Theads 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, can be shared
  - *Fault endpoint* and *timeout endpoint*
  - IPC buffer (backing storage for virtual message registers)
  - stack pointer (SP), instruction pointer (IP), general-purpose registers
  - *Scheduling priority* and *maximum controlled priority (MCP)*
  - *Scheduling context*: right to use CPU time

PGD reference

CNode reference: root of CSpace

Invoked by kernel upon exception

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

```
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,
                         seL4NilData, PROCESS_IPC_BUFFER, ipc_buffer);
if (err != seL4_NoError) return err;

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

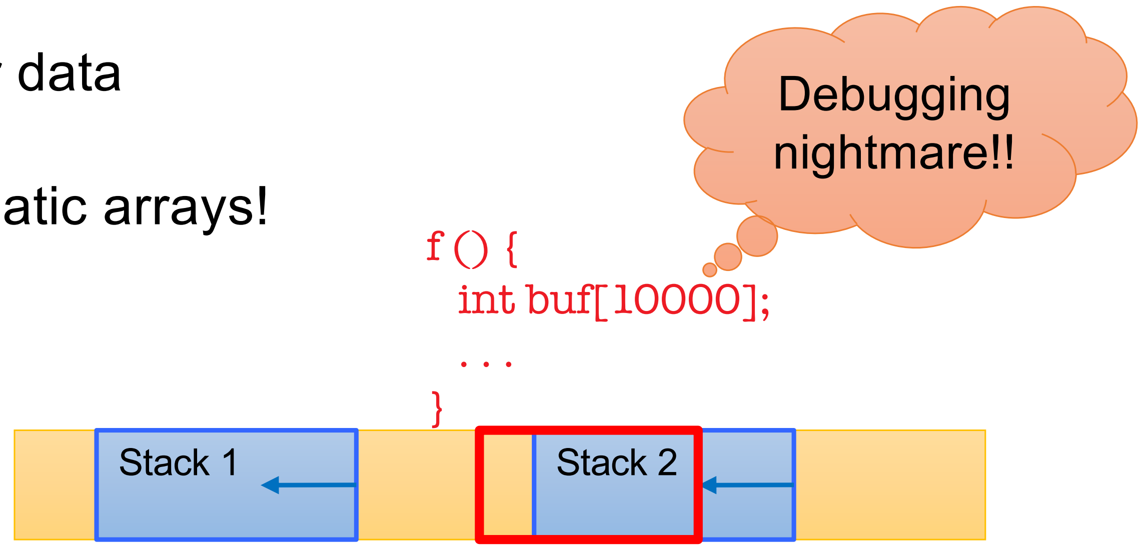
Alloc & map frame for IPC buffer

Alloc slot

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!



Recommend leaving page above top of stack unmapped!

Creating a Thread in *New* AS and CSpace

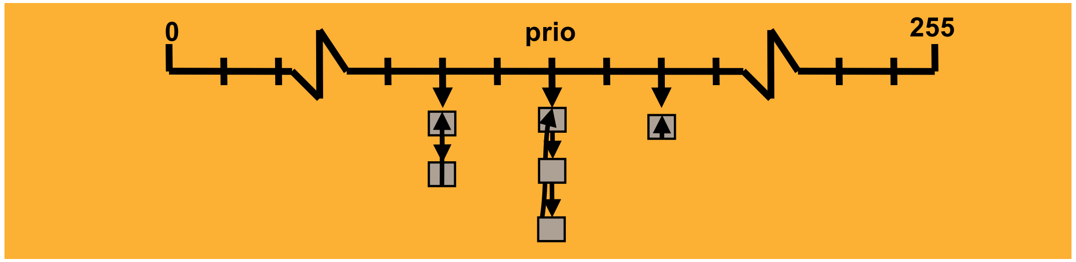
```
/* Allocate, retype and map new frame for IPC buffer as before
 * Allocate and map stack???
 * Allocate and retype a TCB  as before
 * Allocate and retype a  PageGlobalDirectoryObject of size seL4_PageDirBits
 * Mint a new badged cap to the syscall endpoint
 */
cspace_t * new_cpace = ut_alloc(seL4_TCBBits);

char *elf_base = cpio_get_file(_cpio_archive, app_name, &elf_size);
seL4_Word sp = init_process_stack(&cspace, new_pgd, 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,
                         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

- 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 user can do with syscall)
- Thread without scheduling context or 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 best-effort time slice (round robin)



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

seL4 Mechanisms

Interrupts and Exceptions

Exception Handling

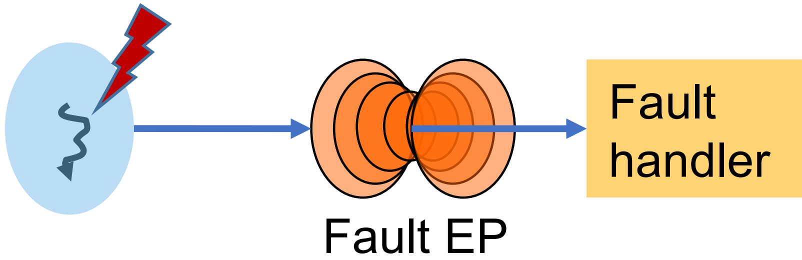
Exception types:

- 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

has its own fault endpoint

On exception:

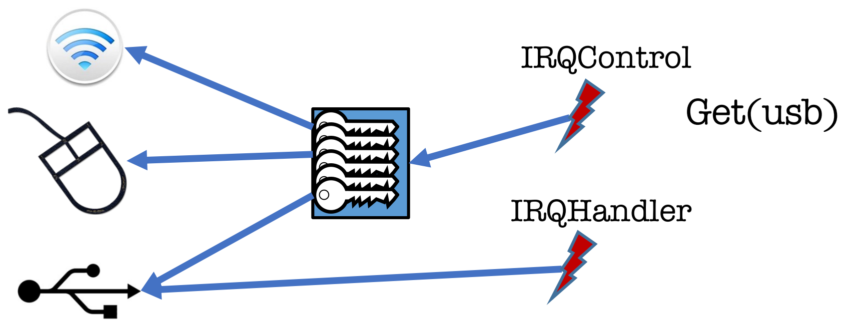
- kernel sends message to fault EP
- pretends to be from faulter
- replying will restart thread

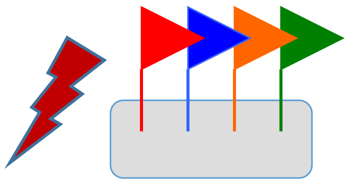


Interrupt Management

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

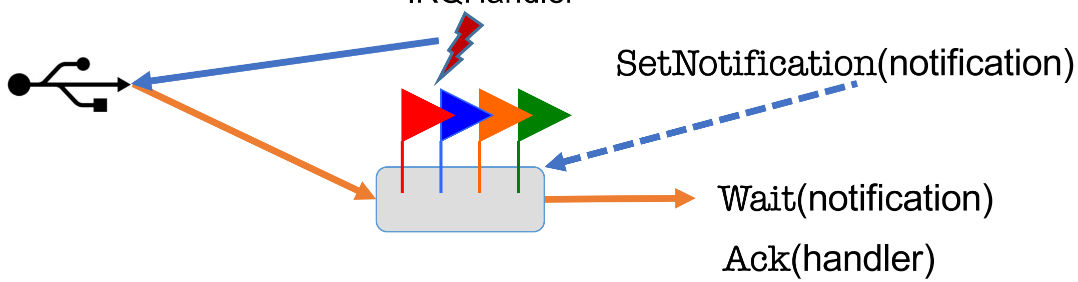




Interrupt Handling

IRQHandler cap allows driver to bind Notification to interrupt

- Notification is used to receive interrupt
- IRQHandler is used to acknowledge interrupt



Unmasks IRQ

```
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, notification);
seL4_IRQHandler_Ack(irq);
```

Device Drivers

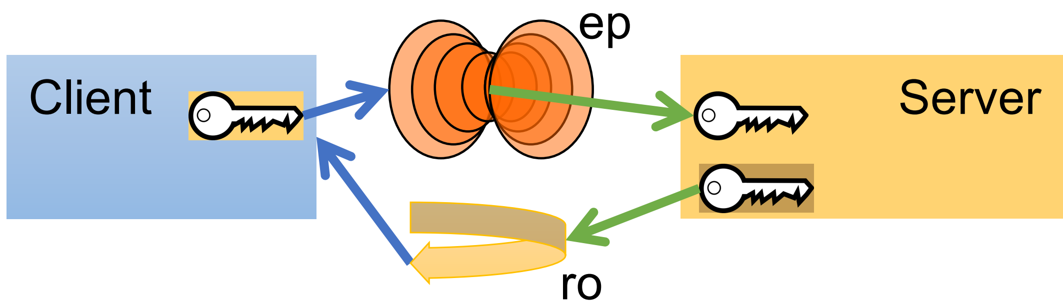
- In seL4 (and all other L4 kernels) drivers are usermode processes
- Drivers do three things:
  - Handle interrupts (already explained)
  - Communicate with rest of OS (IPC + shared memory)
  - Access device registers
- Device register access (Arm uses memory-mapped IO)
  - Have to find frame cap from bootinfo structure
  - Map the appropriate page in the driver's VSpace

Magic device register access

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

seL4 System Design Hints

PS on Reply Objects



Client

Kernel

Server

ReplyRecv(ro,ep,&args)

Call(ep, args)

deliver to server

block client on RO

process

ReplyRecv(ro,ep,&args)

deliver to client

process

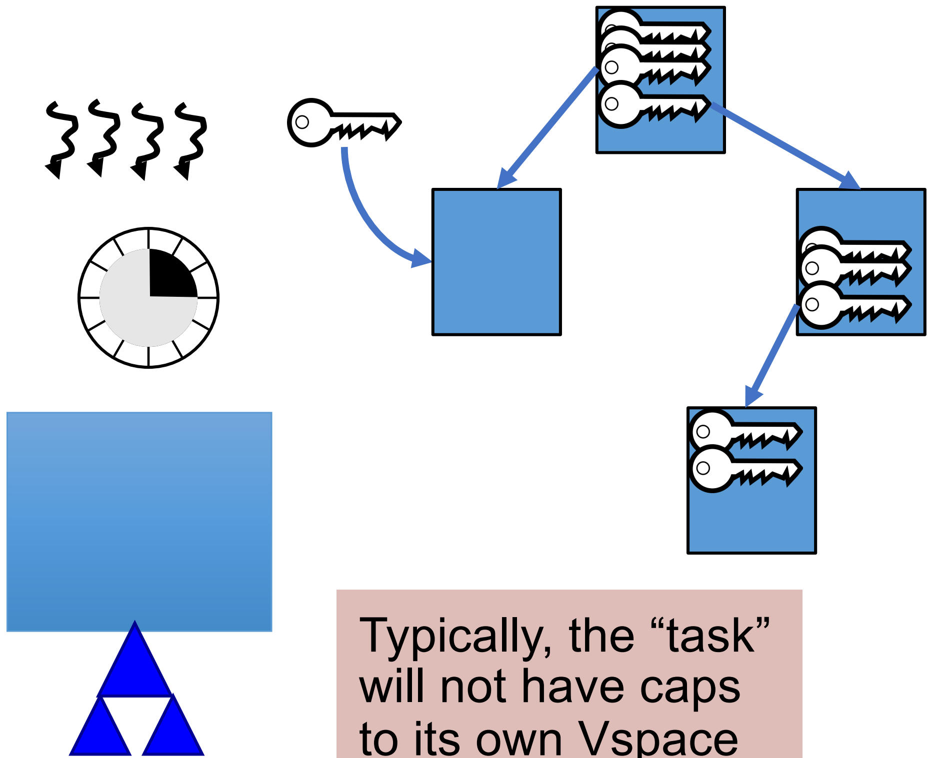
Kernel sets up reply channel in RO
- overwrites previous RO state
- ⇒ need to have multiple ROs to support concurrent long-running client requests!

Kernel has no notion of a process/task!

Informally, a "task" consists of:
- a virtual address space (Vspace)
- a capability space (Cspace)
- one or more threads
- zero or more scheduling contexts
- likely Endpoint(s) & Notification(s)

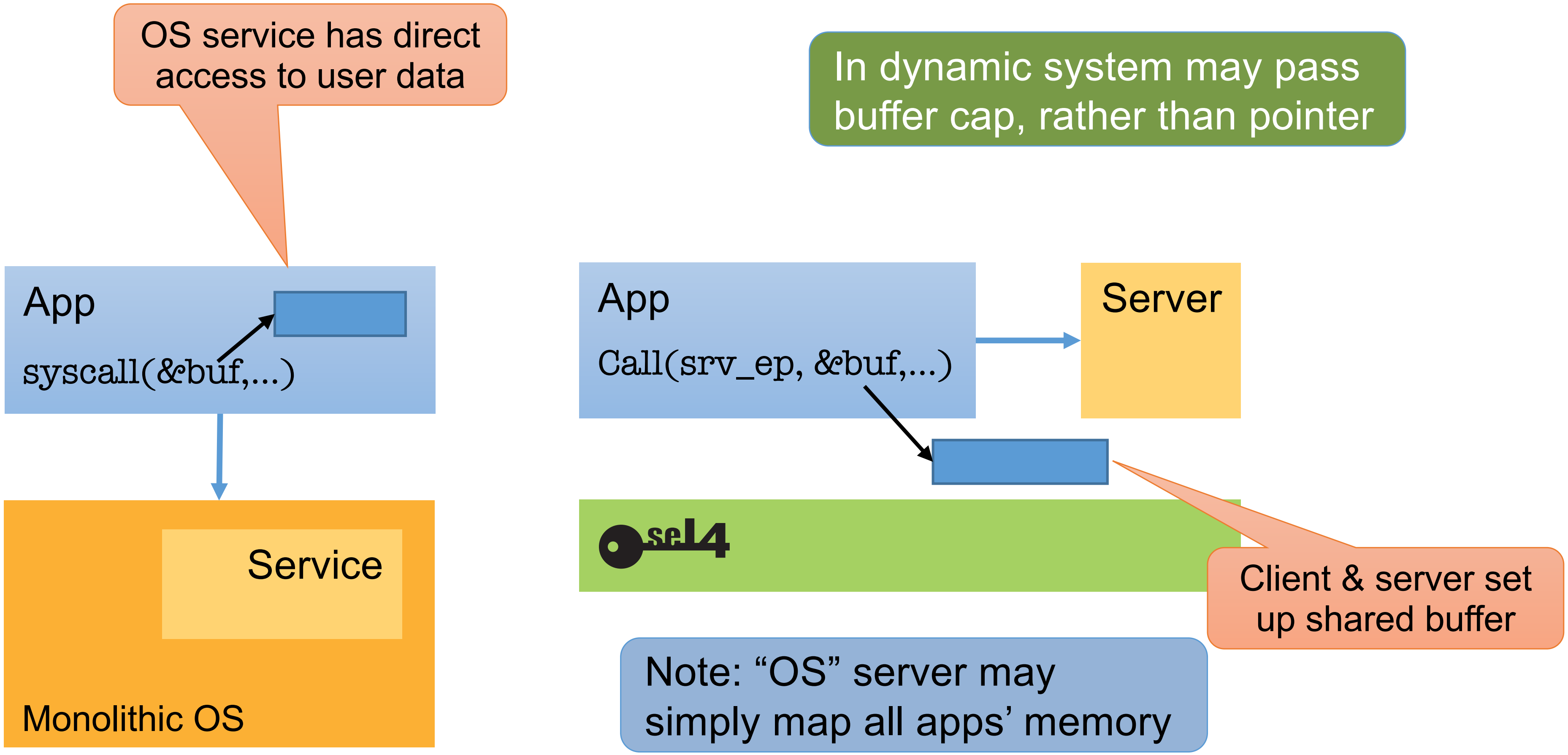
A server may not need an SC, runs on client's

Related tasks may share a Cspace

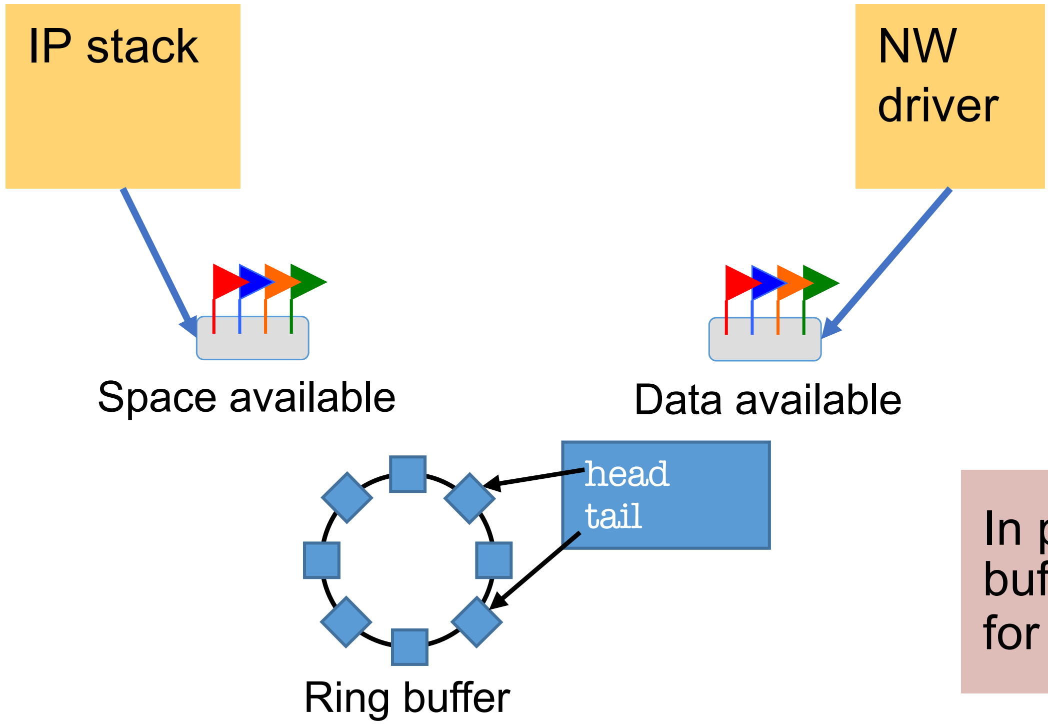


Typically, the "task" will not have caps to its own Vspace and Cspace!

Shared memory is usually required…

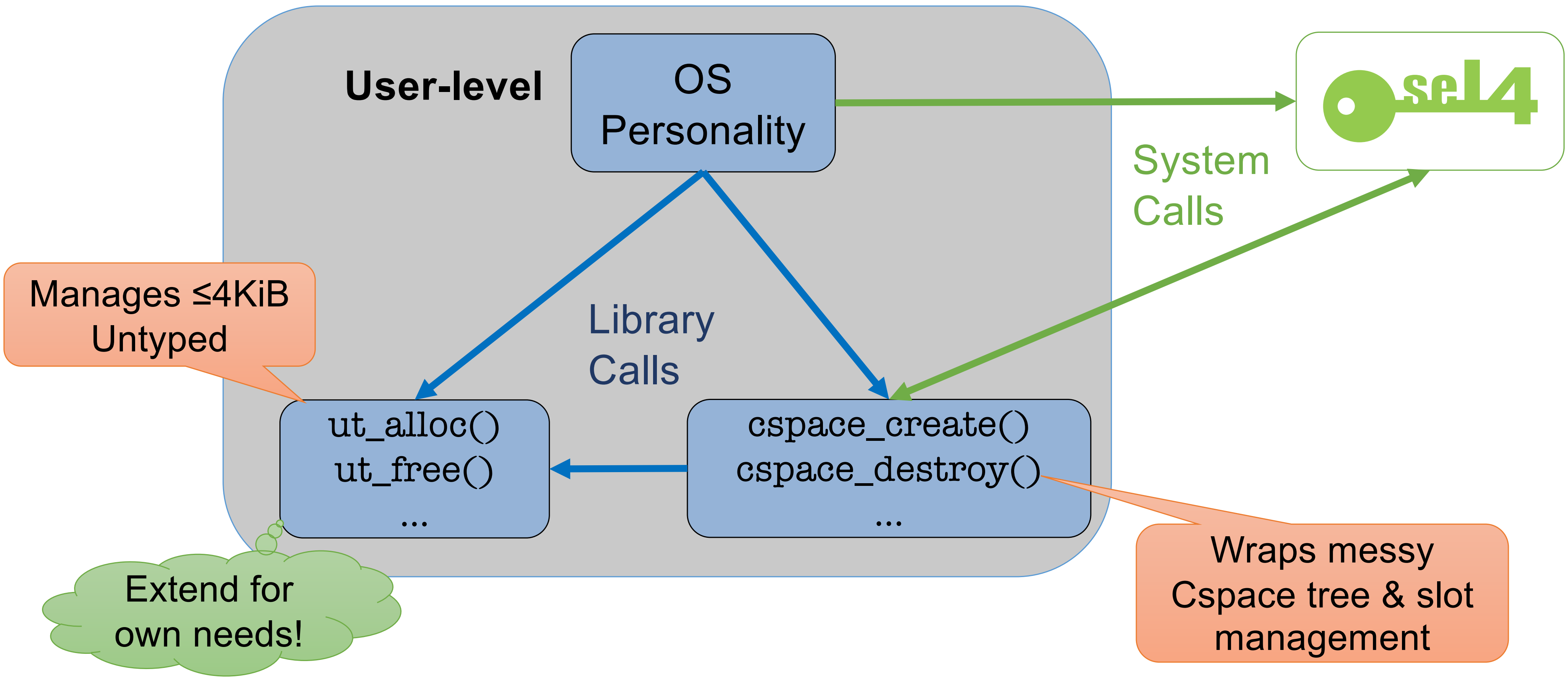


... especially for high-performance I/O



In practice separate buffers & Notifications for tx/rx

Project: cspace and ut libraries



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*.

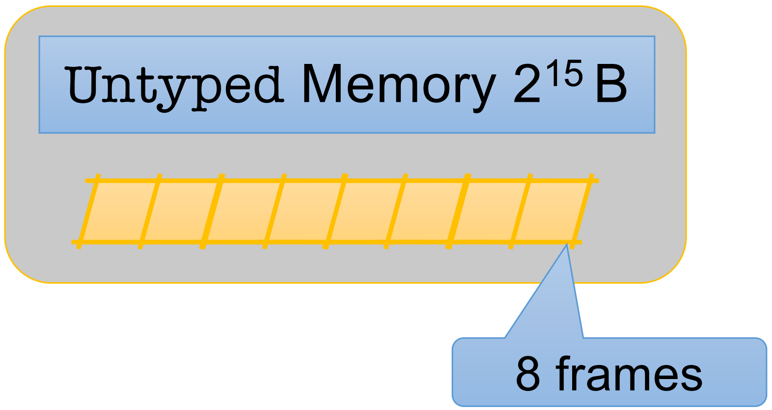
Object	Size (B)	Align (B)
Frame	2^{12}	2^{12}
PT/PD/PUD/PGD	2^{12}	2^{12}
Endpoint	2^{4}	2^{4}
Notification	2^{5}	2^{5}
Scheduling Context	$\geq 2^{8}$	2^{8}
Cslot	2^{4}	2^{4}
Cnode	$\geq 2^{12}$	2^{12}
TCB	2^{11}	2^{11}

Values for AArch64

Memory-Management Caveats

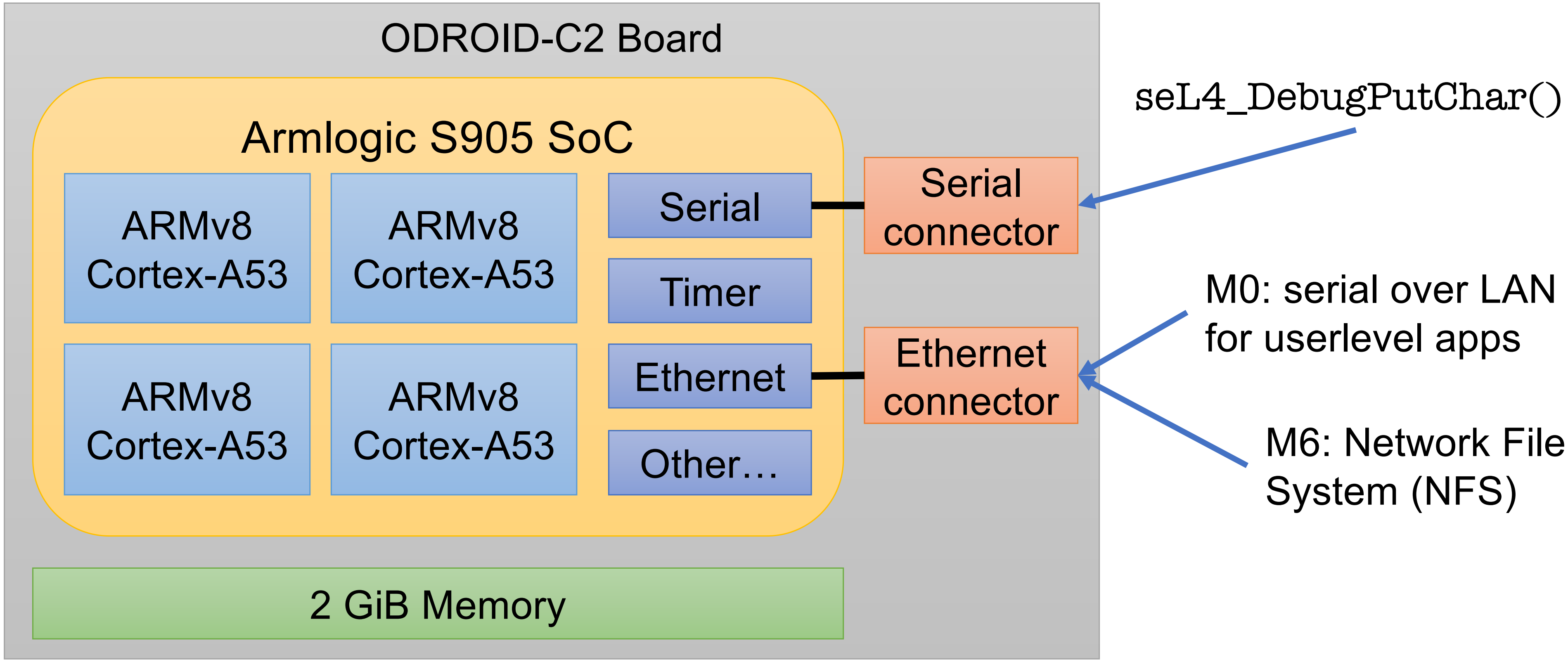
- Objects are allocated by `Retype()` of Untyped memory
- The kernel will not allow you to overlap objects
- `ut_alloc` and `ut_free()` manage user-level view of allocation.
  - Major pain if kernel and user view diverge
  - TIP: Keep objects address and CPtr together!

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.
- Your frametable will eventually integrate with `ut_alloc` to manage the 4KiB `untyped` size.

Project Platform: ODROID-C2



seL4 in the Real World (Courtesy Boeing, DARPA)