



COMP9242 Advanced OS

S2/2016 W01: Introduction to seL4

@GernotHeiser

Never Stand Still

Engineering

Computer Science and Engineering

Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

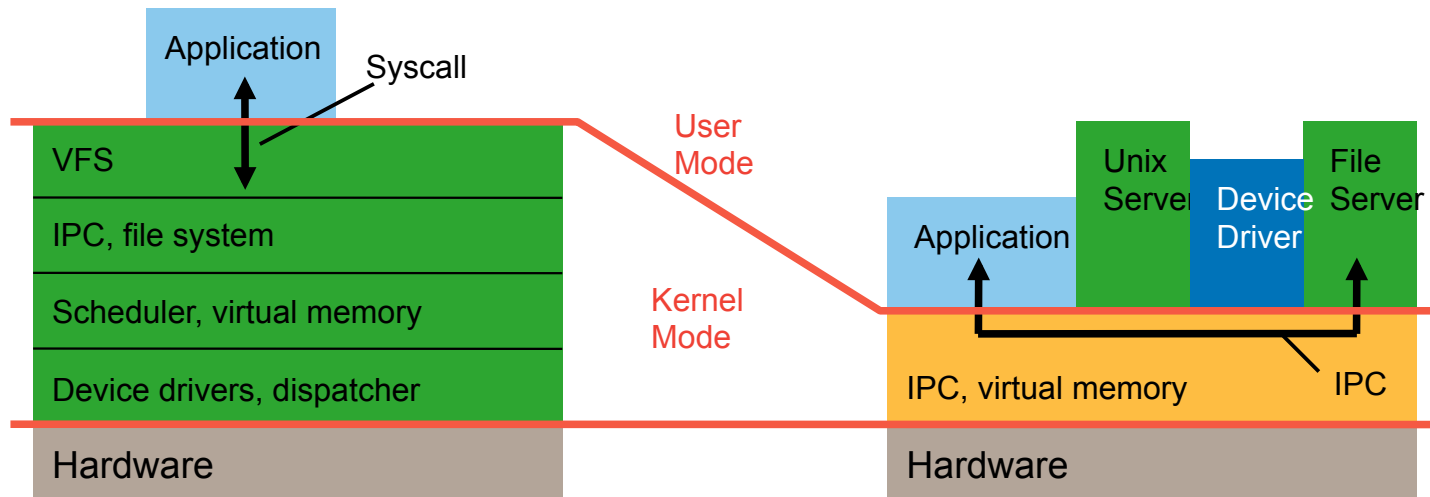
- You are free:
 - to share—to copy, distribute and transmit the work
 - to remix—to adapt the work
- under the following conditions:
 - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

“Courtesy of Gernot Heiser, UNSW Australia”

The complete license text can be found at
<http://creativecommons.org/licenses/by/3.0/legalcode>

Monolithic Kernels vs Microkernels

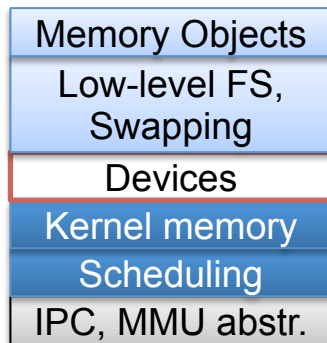
- Idea of microkernel:
 - Flexible, minimal platform
 - Mechanisms, not policies
 - Goes back to Nucleus [Brinch Hansen, CACM'70]



Microkernel Evolution

First generation

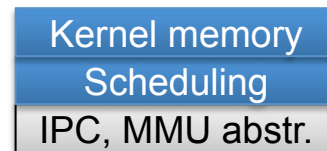
- Eg Mach ['87]



- 180 syscalls
- 100 kLOC
- 100 μ s IPC

Second generation

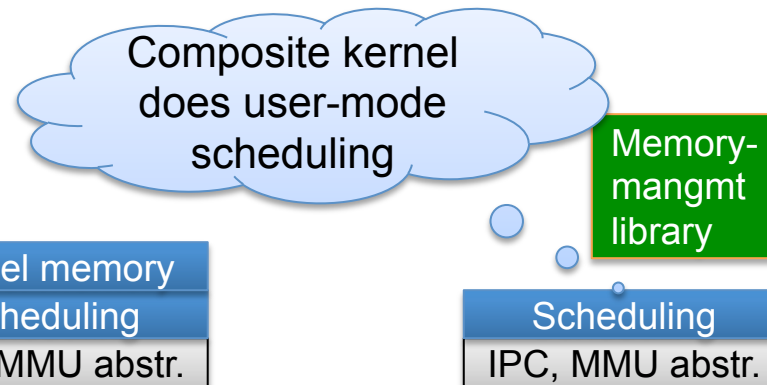
- Eg L4 ['95]



- ~7 syscalls
- ~10 kLOC
- ~ 1 μ s IPC

Third generation

- seL4 ['09]

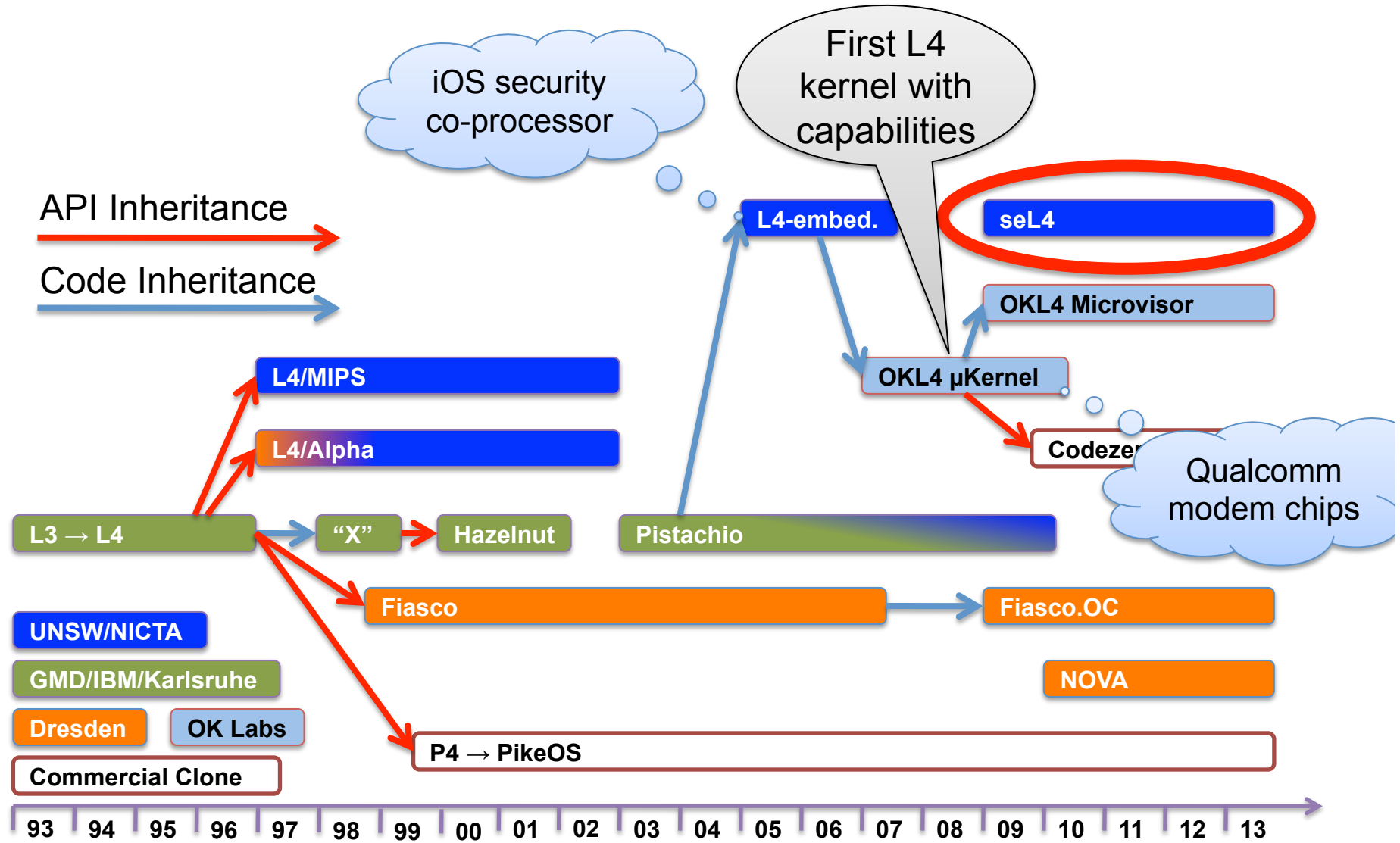


- ~3 syscalls
- 9 kLOC
- 0.1 μ s IPC
- *capabilities*
- *design for isolation*

2nd-Generation Microkernels

- 1st-generation kernels (Mach, Chorus) were a failure
 - Complex, inflexible, slow
- L4 was first 2G microkernel [Liedtke, SOSP'93, SOSP'95]
 - Radical simplification & manual micro-optimisation
 - *“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.”*
 - High IPC performance
- Family of L4 kernels:
 - Original Liedtke (GMD) assembler kernel ('95)
 - Family of kernels developed by Dresden, UNSW/NICTA, Karlsruhe
 - Commercial clones (PikeOS, P4, CodeZero, ...)
 - Influenced commercial QNX ('82), Green Hills Integrity ('90s)
 - Generated NICTA startup Open Kernel Labs (OK Labs)
 - large-scale commercial deployment (multiple billions shipped)

L4: A Family of High-Performance Microkernels

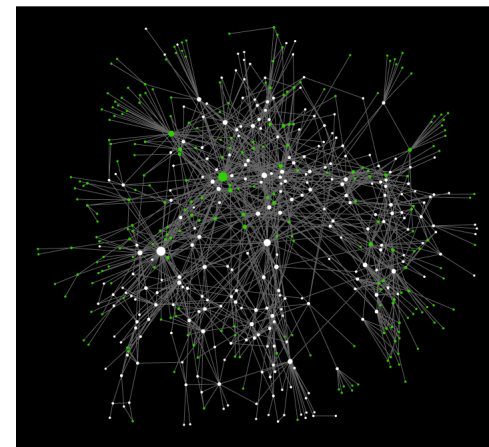


Issues of 2G Microkernels

- L4 solved performance issue [Härtig et al, SOSPP'97]
- Left a number of security issues unsolved
- Problem: ad-hoc approach to protection and resource management
 - Global thread name space \Rightarrow covert channels [Shapiro'03]
 - Threads as IPC targets \Rightarrow insufficient encapsulation
 - Single kernel memory pool \Rightarrow DoS attacks
 - Insufficient delegation of authority \Rightarrow limited flexibility, performance
 - Unprincipled management of time
- Addressed by seL4
 - Designed to support safety- and security-critical systems
 - Principled time management not yet mainline (RT branch)

seL4 Principles

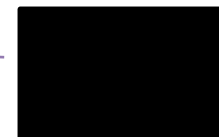
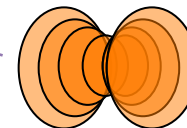
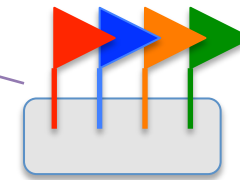
- Single protection mechanism: capabilities
 - Proper time management to be finished this year
- All resource-management policy at user level
 - Painful to use
 - Need to provide standard memory-management library
 - Results in L4-like programming model
- Suitable for formal verification (proof of implementation correctness)
 - Attempted since '70s
 - Finally achieved by L4.verified project at NICTA [Klein et al, SOSP'09]



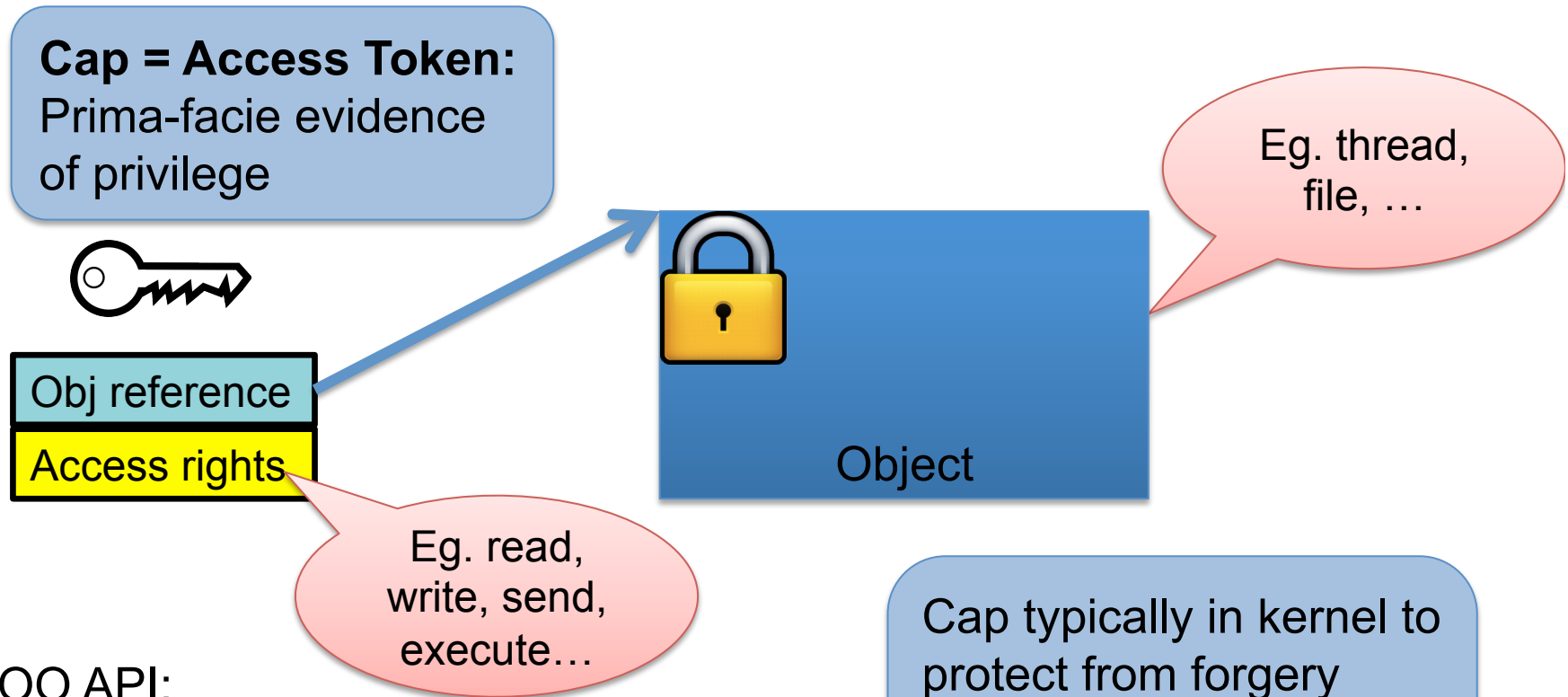
seL4 Concepts

Note: differences between AOS and mainline kernels!

- Capabilities (Caps)
 - mediate access
- Kernel objects:
 - Threads (thread-control blocks: TCBs)
 - Address spaces (page table objects: PDs, PTs)
 - Endpoints (IPC EPs, Notification AEPs)
 - Capability spaces (CNodes)
 - Frames
 - Interrupt objects
 - Untyped memory
- System calls
 - Send, Wait (and variants)
 - Yield



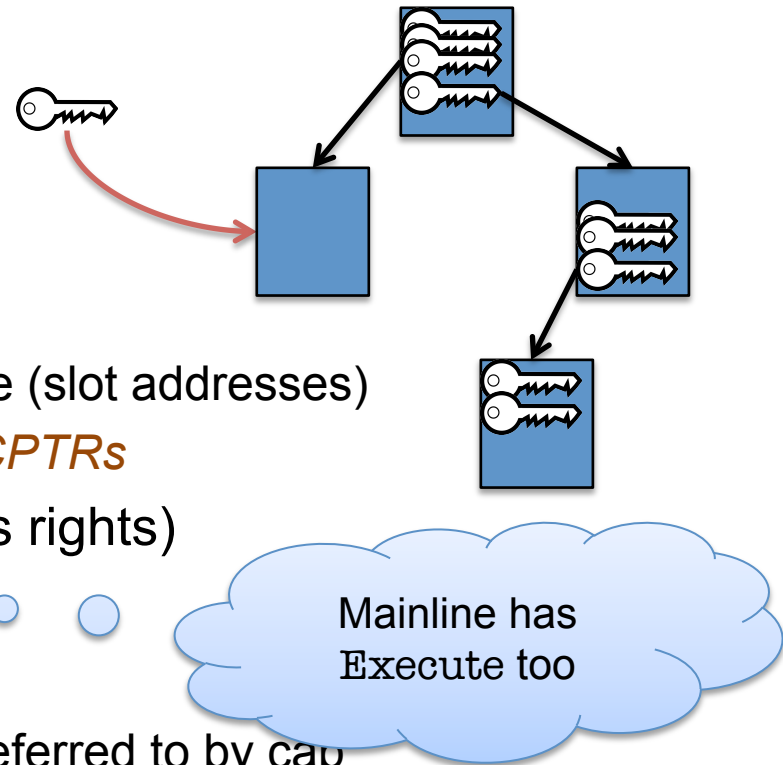
What are (Object) Capabilities?



- OO API:
 `err = method(cap, args);`
- Used in some earlier microkernels:
 - KeyKOS [‘85], Mach [‘87], EROS [‘99]

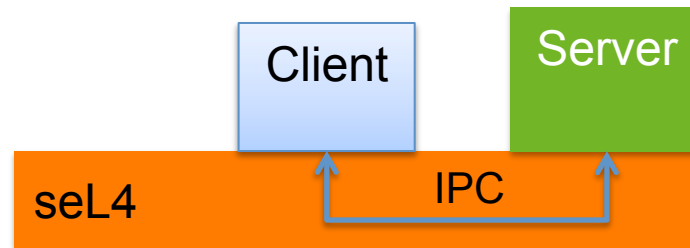
🔑 seL4 Capabilities

- 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, Grant (cap transfer)
- Main operations on caps:
 - *Invoke*: perform operation on object referred to by cap
 - Possible operations depend on object type
 - *Copy/Mint/Grant*: create copy of cap with *same/lesser* privilege
 - *Move/Mutate*: transfer to different address with same/lesser privilege
 - *Delete*: invalidate slot (cleans up object if this is the only cap to it)
 - *Revoke*: delete any derived (eg. copied or minted) caps

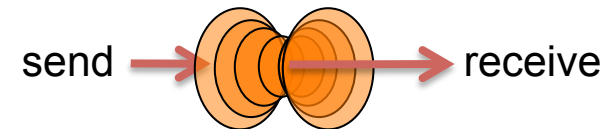


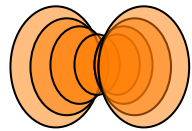
Inter-Process Communication (IPC)

- Fundamental microkernel operation
 - Kernel provides no services, only mechanisms
 - OS services provided by (protected) user-level server processes
 - invoked by IPC

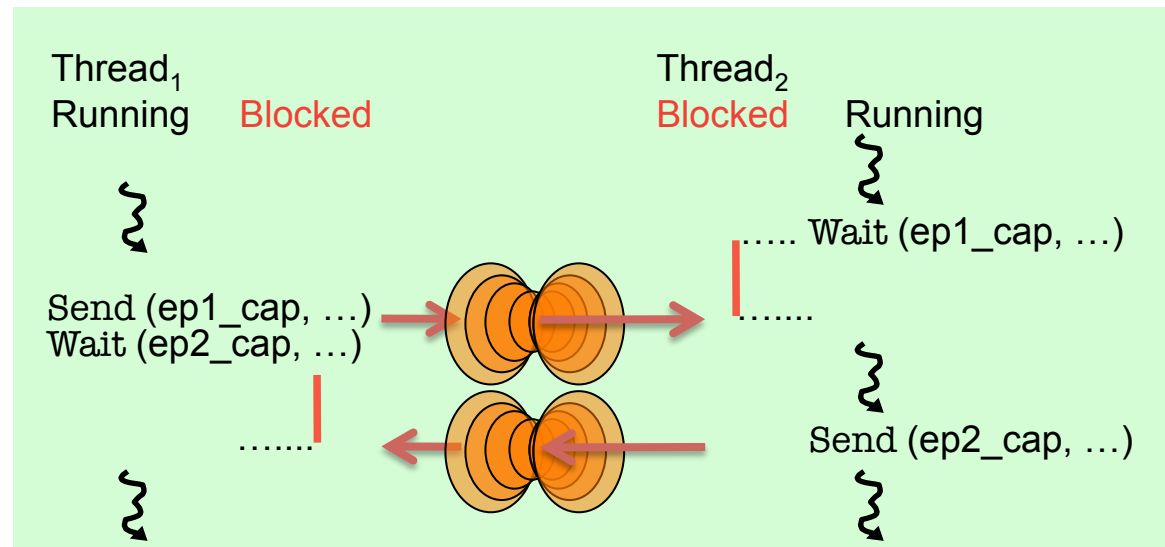


- seL4 IPC uses a handshake through *endpoints*:
 - Transfer points without storage capacity
 - Message must be transferred instantly
 - Single-copy user → user by kernel

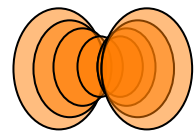




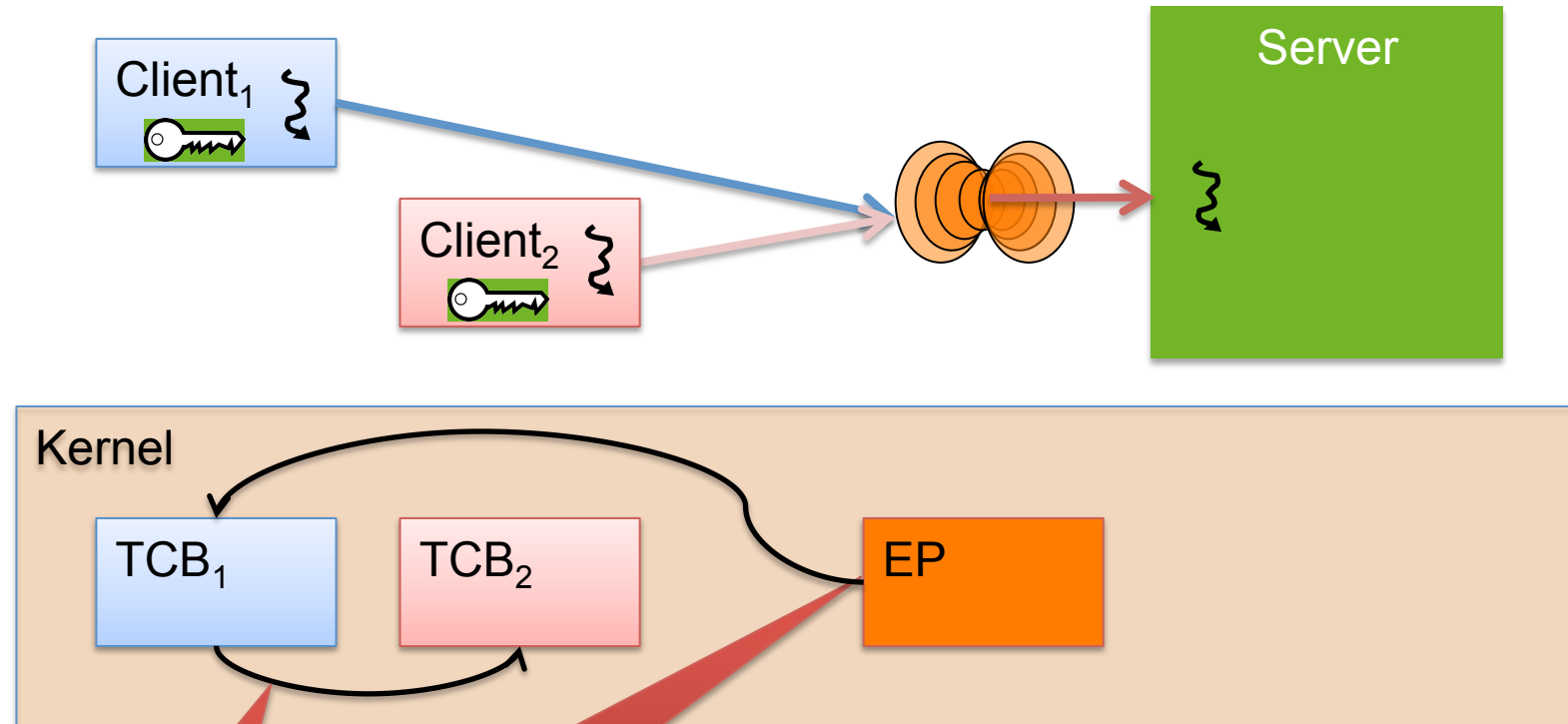
IPC: (Synchronous) Endpoints



- Threads must rendez-vous for message transfer
 - One side blocks until the other is ready
 - Implicit synchronisation
- Message copied from sender's to receiver's *message registers*
 - Message is combination of caps and data words
 - Presently max 121 words (484B, incl message “tag”)
 - Should never use anywhere near that much!



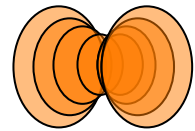
IPC Endpoints are Message Queues



Further callers of same direction queue behind

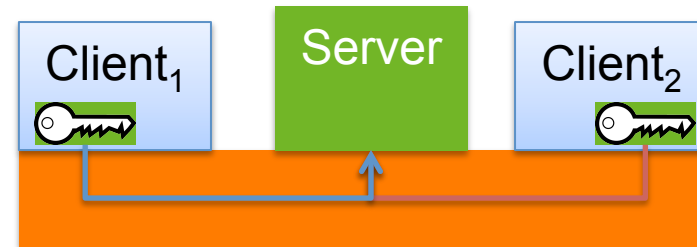
First invocation queues caller

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

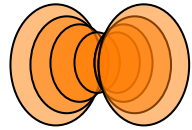


Client-Server Communication

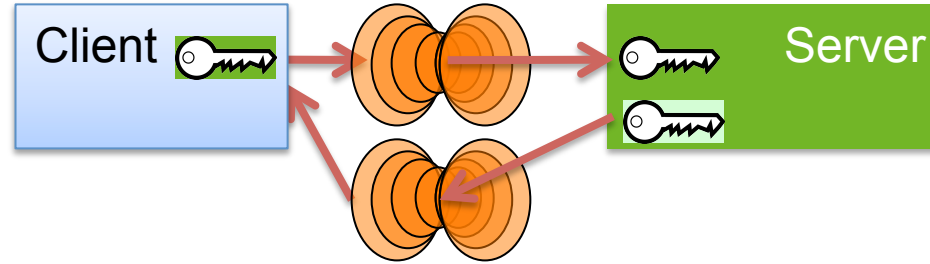
- Asymmetric relationship:
 - Server widely accessible, clients not
 - How can server reply back to client (distinguish between them)?



- Client can pass (session) reply cap in first request
 - server needs to maintain session state
 - forces stateful server design
- seL4 solution: Kernel provides single-use *reply cap*
 - only for Call operation (Send+Wait)
 - allows server to reply to client
 - cannot be copied/minted/re-used but can be moved
 - one-shot (automatically destroyed after first use)



Call RPC Semantics



Client

Kernel

Server

Call(ep,...)

Wait(ep,&rep)

mint rep
deliver to server

process
Send(rep,...)

deliver to client
destroy rep

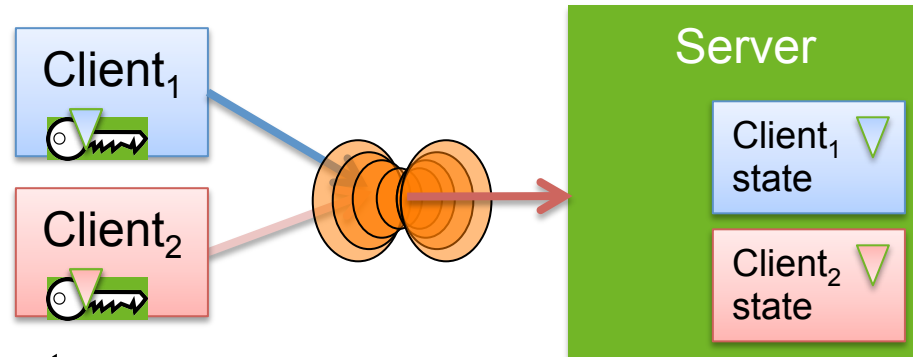
process

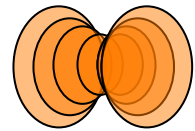
process

Identifying Clients

Stateful server serving multiple clients

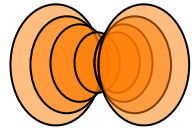
- Must respond to correct client
 - Ensured by reply cap
- Must associate request with correct state
- Could use separate EP per client
 - endpoints are lightweight (16 B)
 - but requires mechanism to wait on a set of EPs (like select)
- Instead, seL4 allows to individually mark (“badge”) caps to same EP
 - server provides individually badged caps to clients
 - server tags client state with badge (through `Mint()`)
 - kernel delivers badge to receiver on invocation of badged caps



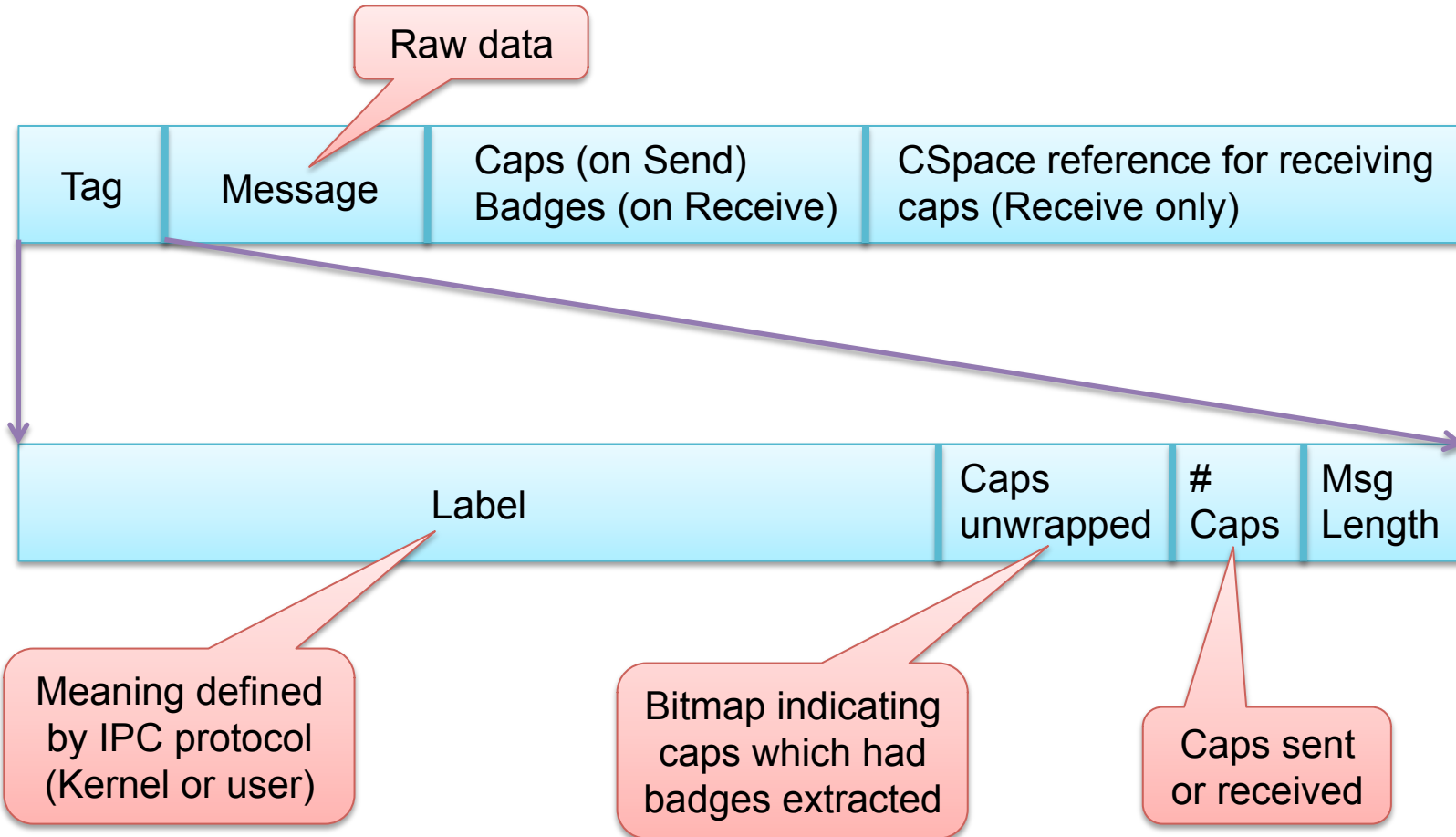


IPC Mechanics: Virtual Registers

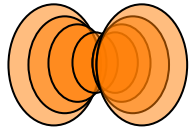
- Like physical registers, virtual registers are thread state
 - context-switched by kernel
 - implemented as physical registers or thread-local memory
- Message registers
 - contain message transferred in IPC
 - architecture-dependent subset mapped to physical registers
 - 5 on ARM, 3 on x86
 - library interface hides details
 - 1st transferred word is special, contains *message tag*
 - API MR[0] refers to next word (not the tag!)
- Reply cap
 - *overwritten by next receive!*
 - can move to CSpace with `cspace_save_reply_cap()`



IPC Message Format



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



Client-Server IPC Example

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

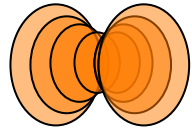
```
seL4_Word addr = ut_alloc(seL4_EndpointBits);
err = cspace_ut_retype_addr(tcb_addr, seL4_EndpointObject,
                           seL4_EndpointBits, cur_cspace, &ep_cap)
seL4_CPtr cap = cspace_mint_cap(dest, cur_cspace, ep_cap, seL4_all_rights,
                               seL4_CapData_Badge_new(0xff));
...
seL4_Word badge;
seL4_MessageInfo_t msg = seL4_Wait(ep, &badge);
...
seL4_MessageInfo_t reply = seL4_MessageInfo_new(0, 0, 0, 0);
seL4_Reply(reply);
```

Allocate EP and retype

Insert EP into CSpace

Cap is badged 0xff

Implicit use of reply cap



Server Saving Reply Cap

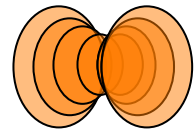
Server

```
seL4_Word addr = ut_alloc(seL4_EndpointBits);
err = cspace_ut_retype_addr(tcb_addr, seL4_EndpointObject,
                           seL4_EndpointBits, cur_cspace, &ep_cap)
seL4_CPtr cap = cspace_mint_cap(dest, cur_cspace, ep_cap, seL4_all_rights,
                               seL4_CapData_Badge_new(0xff));
...
seL4_Word badge;
seL4_MessageInfo_t msg = seL4_Wait(ep, &badge);
seL4_CPtr slot = cspace_save_reply_cap(cur_cspace);
...
seL4_MessageInfo_t reply = seL4_MessageInfo_new(0, 0, 0, 0);
seL4_Send(slot, reply);
cspace_free_slot(slot);
```

Save reply cap
in CSpace


Explicit use
of reply cap

Reply cap no
longer valid



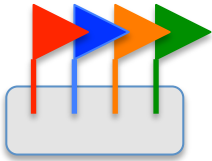
IPC Operations Summary

- Send (ep_cap, ...), Wait (ep_cap, ...)
 - blocking message passing
 - needs Write, Read permission, respectively
- NBSend (ep_cap, ...)
 - Polling send: *silently* discard message if receiver isn't ready
- Call (ep_cap, ...)
 - equivalent to Send (ep_cap,...) + reply-cap + Wait (ep_cap,...)
 - *Atomic*: guarantees caller is ready to receive reply
- Reply (...)
 - equivalent to Send (rep_cap, ...)
- ReplyWait (ep_cap, ...)
 - equivalent to Reply (...) + Wait (ep_cap, ...)
 - at present solely for efficiency of server operation



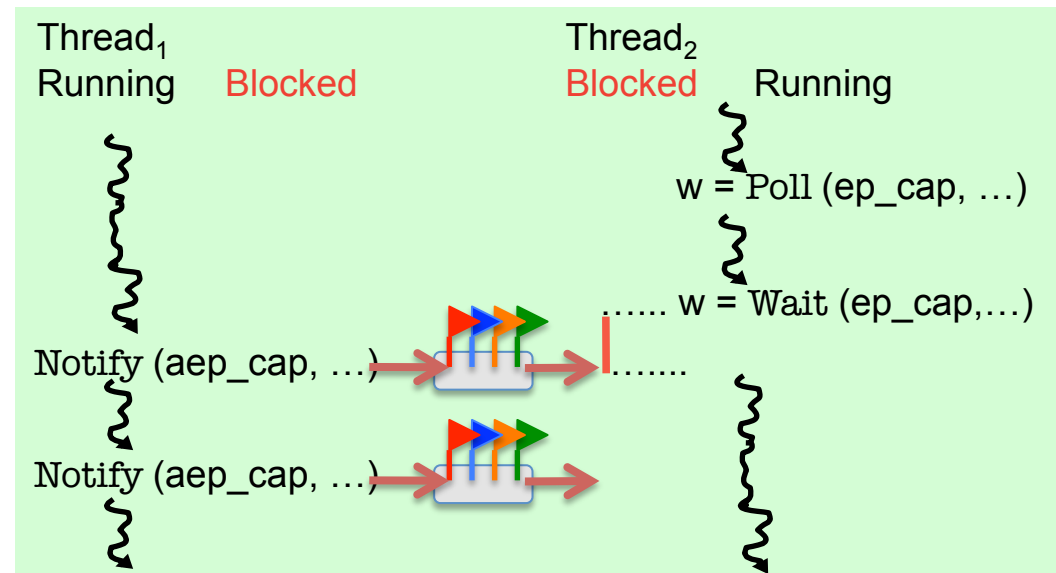
Need error handling protocol !

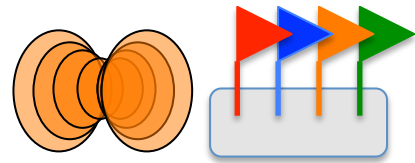
No failure notification where this reveals info on other entities!



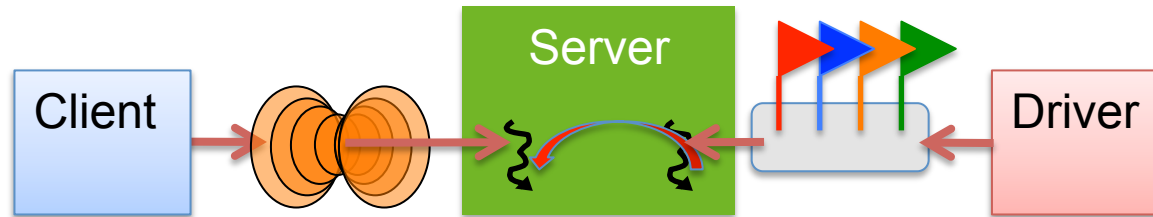
Notifications: Asynchronous Endpoints

- Logically, AEP is an array of binary semaphores
 - Multiple signalling, select-like wait
 - Not a message-passing IPC operation!
- Implemented by *data word* in AEP
 - 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



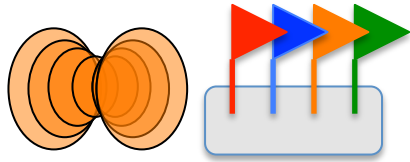


Receiving from EP *and* AEP



Server with synchronous and asynchronous interface

- Example: file system
 - synchronous (RPC-style) client protocol
 - asynchronous notifications from driver
- Could have separate threads waiting on endpoints
 - forces multi-threaded server, concurrency control
- Alternative: allow single thread to wait on both EP types
 - AEP is *bound* to thread with `BindAEP()` syscall
 - thread waits on synchronous endpoint
 - Notification delivered as if caller had been waiting on AEP



AOS vs Mainline Kernel Differences

- “Synchronous” vs “asynchronous” endpoint terminology is confusing
- seL4 really has only synchronous IPC, plus signal-like notifications
- Fixed in recent mainline kernels

AOS Kernel

- Sync EP, sync message
- AEP, async notification
- Send/Receive/Call/Reply&Wait
- NBSend (EP)
- AEP: NBSend, Wait

Mainline

- EP, message
- Notification obj, notification
- Send/Receive/Call/Reply&Wait
- NBSend, Poll, NBReply&Wait
- Signal, Poll, Wait


Derived Capabilities

- Badging is an example of *capability derivation*
- The *Mint* operation creates a new, less powerful cap
 - Can add a badge
 - $\text{Mint}(\text{key}, \blacktriangledown) \rightarrow \text{key} \blacktriangledown$
 - Can strip access rights
 - eg $\text{WR} \rightarrow \text{R/O}$
- *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...

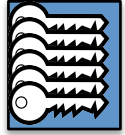
Remember,
caps are kernel
objects!

seL4 System Calls

- Notionally, seL4 has 6 syscalls:
 - Yield(): invokes scheduler
 - only syscall which doesn't require a cap!
 - Send(), Receive() and 3 variants/combinations thereof
 - Notify() 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

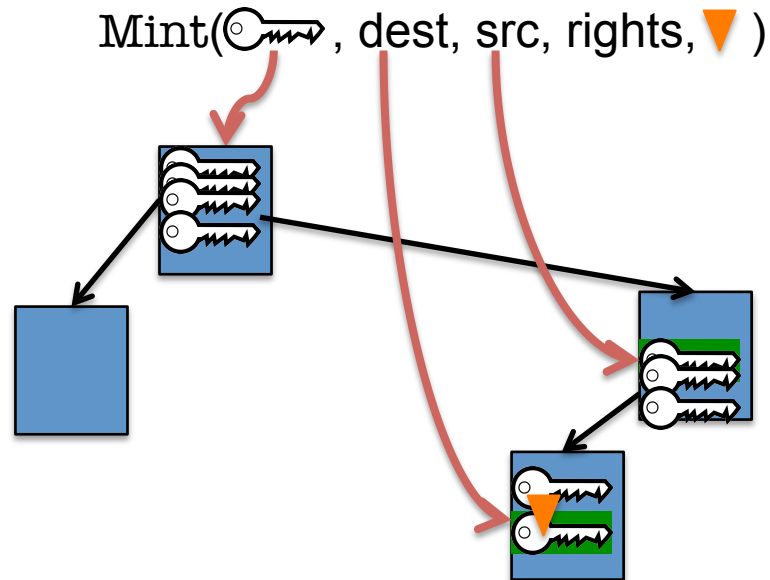


seL4 Memory-Management Principles

- 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



- 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

```
extern cspace_t * cspace_create(int levels); /* either 1 or 2 level */  
extern cspace_err_t cspace_destroy(cspace_t *c);
```

```
extern seL4_CPtr cspace_copy_cap(cspace_t *dest, cspace_t *src,  
                                seL4_CPtr src_cap, seL4_CapRights rights);
```

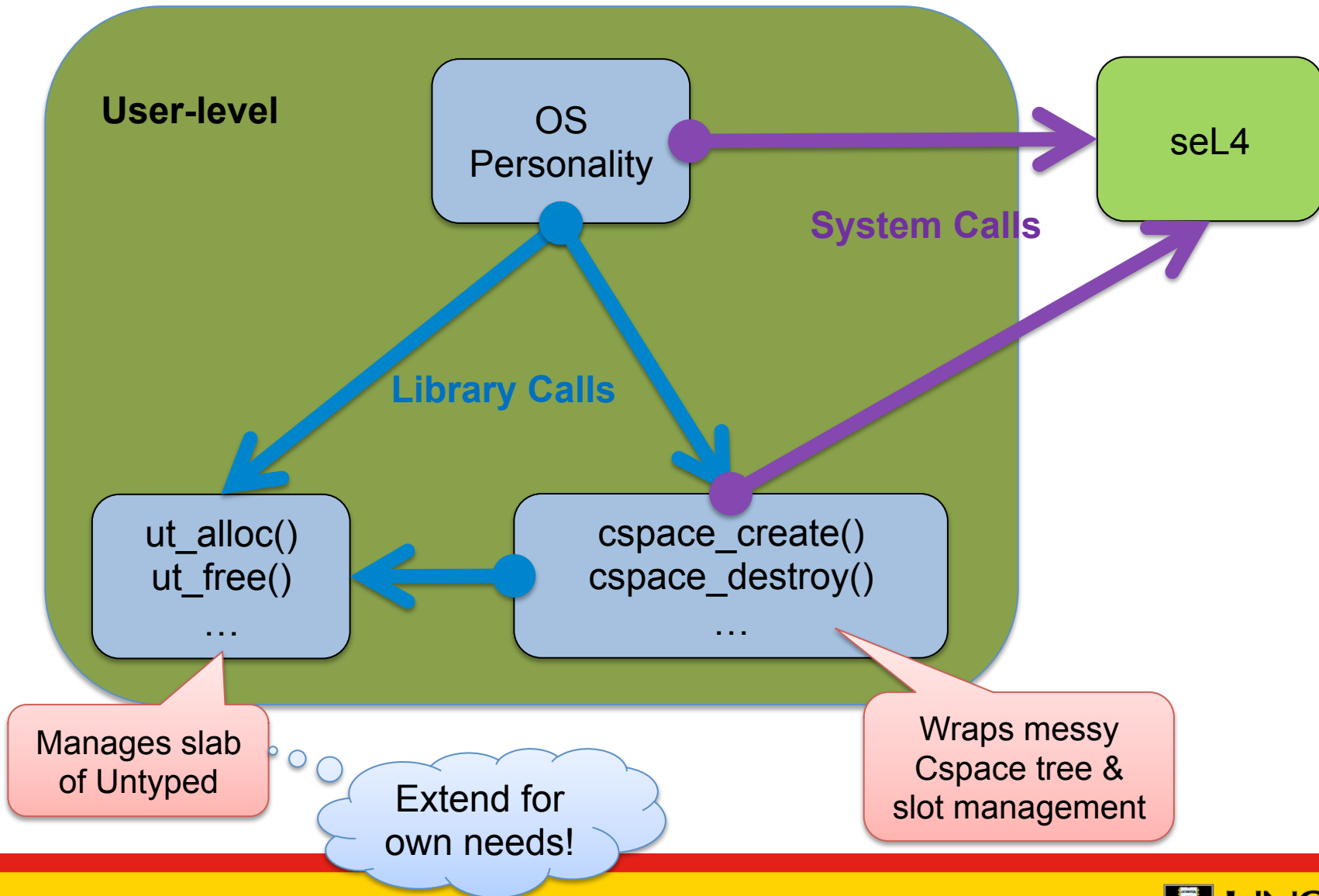
```
extern seL4_CPtr cspace_mint_cap(cspace_t *dest, cspace_t *src,  
                                seL4_CPtr src_cap, seL4_CapRights rights,  
                                seL4_CapData badge);
```

```
extern seL4_CPtr cspace_move_cap(cspace_t *dest, cspace_t *src,  
                                 seL4_CPtr src_cap);
```

```
extern cspace_err_t cspace_delete_cap(cspace_t *c, seL4_CPtr cap);
```

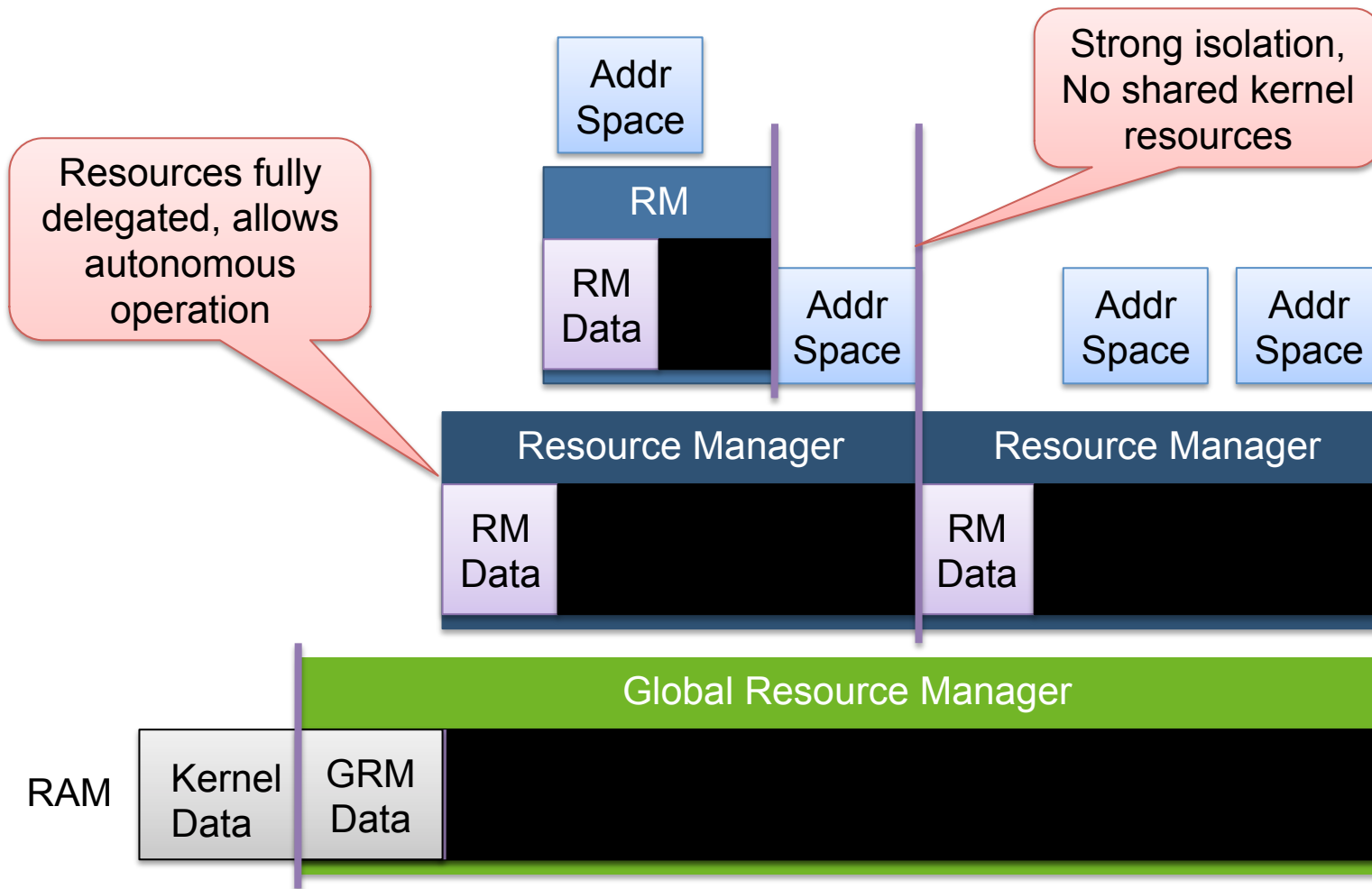
```
extern cspace_err_t cspace_revoke_cap(cspace_t *c, seL4_CPtr cap);
```

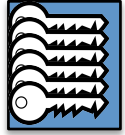
cspace and ut libraries



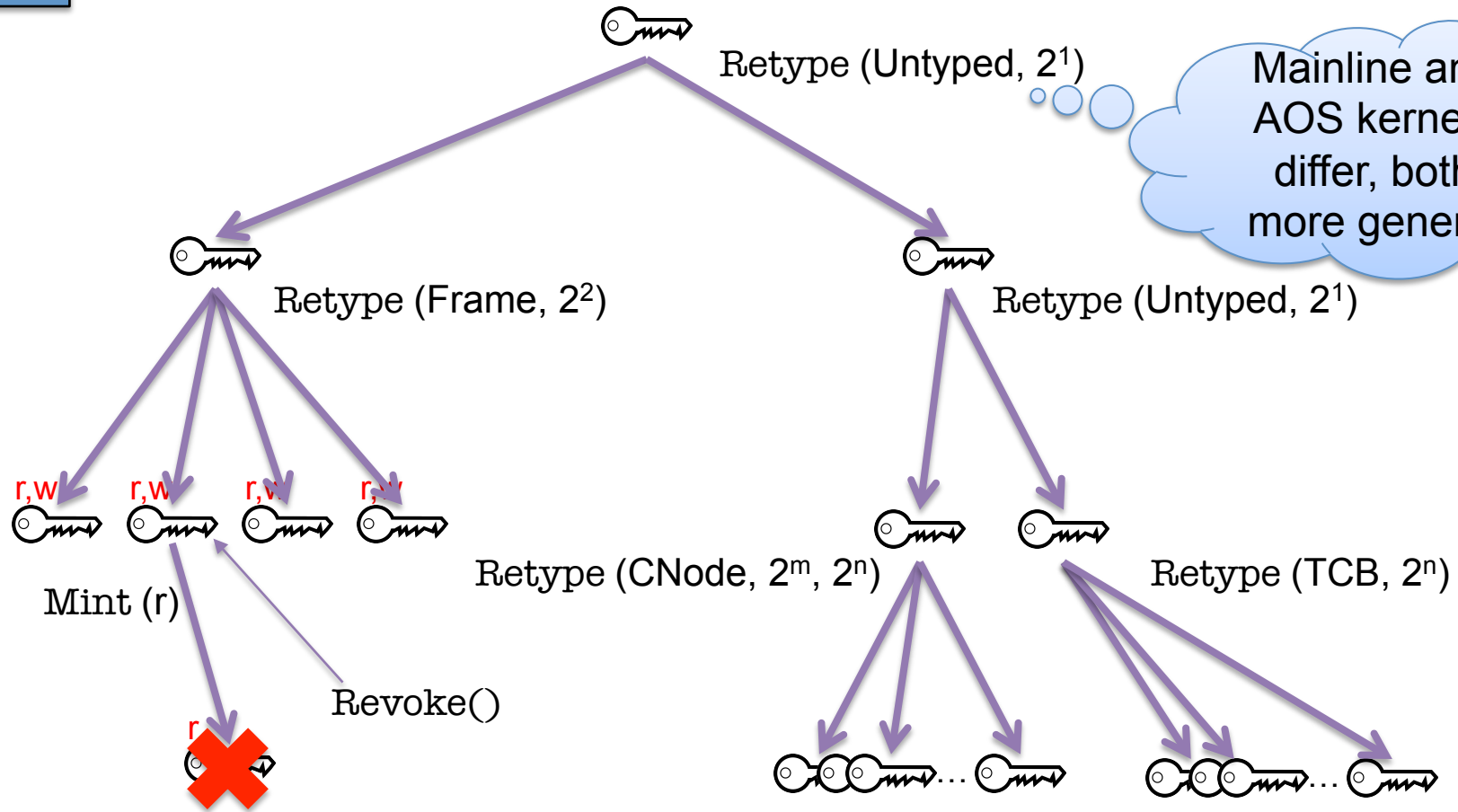


seL4 Memory Management Approach





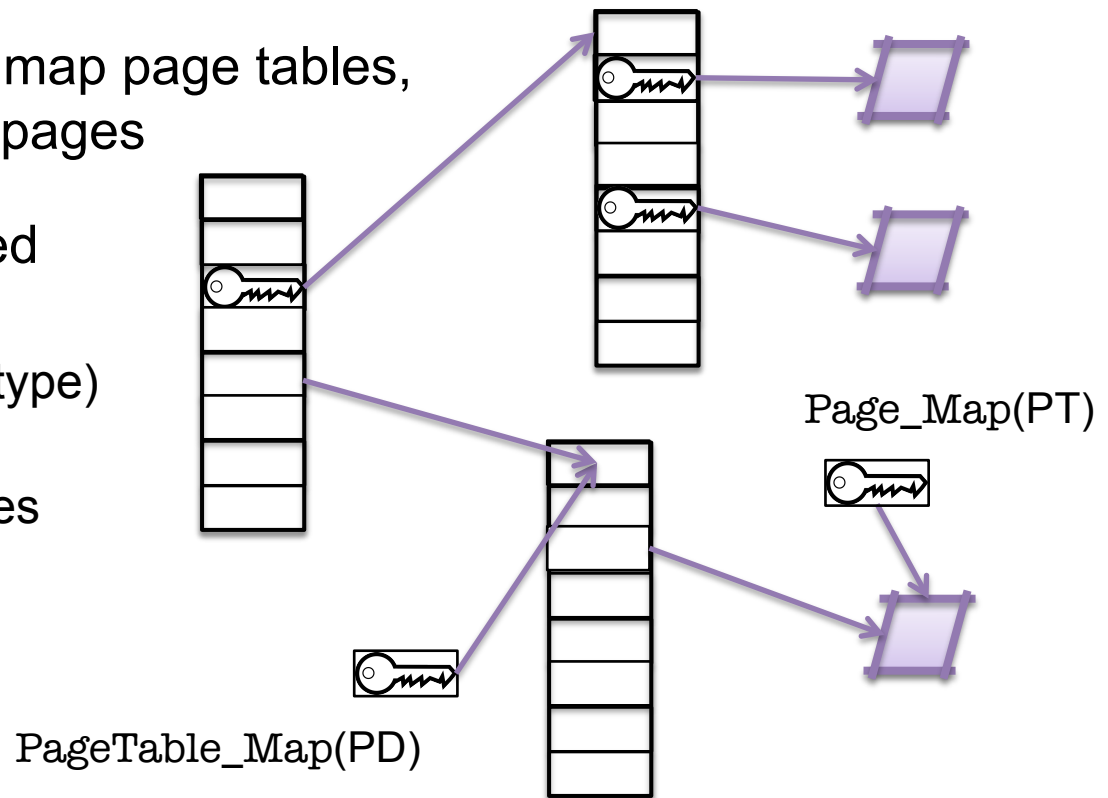
Memory Management Mechanics: Retype





seL4 Address Spaces (VSpaces)

- Very thin wrapper around hardware page tables
 - Architecture-dependent
 - ARM and (32-bit) x86 are very similar
- Page directories (PDs) map page tables, page tables (PTs) map pages
- A VSpace is represented by a PD object:
 - Creating a PD (by Retype) creates the VSpace
 - Deleting the PD deletes the VSpace





Address Space Operations

cap to level 1
page table

```
seL4_Word frame_addr = ut_alloc(seL4_PageBits, ...);
err = cspace_ut_retype_addr(frame_addr, seL4_ARM_Page,
                             seL4_ARM_PageBits, cur_cspace, &frame_cap);

map_page(frame_cap, pd_cap, 0xA0000000, seL4_AllRights,
          seL4_ARM_Default_VMAttributes);
bzero((void *)0xA0000000, PAGE_SIZE);
```

- 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_addr, p_addr)!

```
seL4_ARM_Page_Unmap(frame_cap);
cspace_delete_cap(frame_cap);
ut_free(frame_addr, seL4_PageBits);
```

Poor API
choice!



Multiple Frame Mappings: Shared Memory

```
seL4_CPtr new_frame_cap = cspace_copy_cap(cur_cspace, cur_cspace,  
                                           existing_frame_cap,  
                                           seL4_AllRights);  
  
map_page(new_frame_cap, pd_cap, 0xA0000000, seL4_AllRights,  
         seL4_ARM_Default_VMAttributes);  
bzero((void *)0xA0000000, PAGE_SIZE);
```

```
seL4_ARM_Page_Unmap(existing_frame_cap);  
cspace_delete_cap(existing_frame_cap)  
seL4_ARM_Page_Unmap(new_frame_cap);  
cspace_delete_cap(new_frame_cap)  
ut_free(frame_addr, seL4_PageBits);
```

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



Memory Management Caveats

- The object manager handles allocation for you
- Very simple buddy-allocator, you need to understand how it works:
 - Freeing an object of size n : you can allocate new objects \leq size n
 - Freeing 2 objects of size n **does not mean** that you can allocate an object of size $2n$.

Object	Size (B), ARM	Alignment (B), ARM
Frame	2^{12}	2^{12}
Page directory	2^{14}	2^{14}
Endpoint	2^4	2^4
Cslot	2^4	2^4
Cnode	2^{14}	2^{14}
TCB	2^9	2^9
Page table	2^{10}	2^{10}

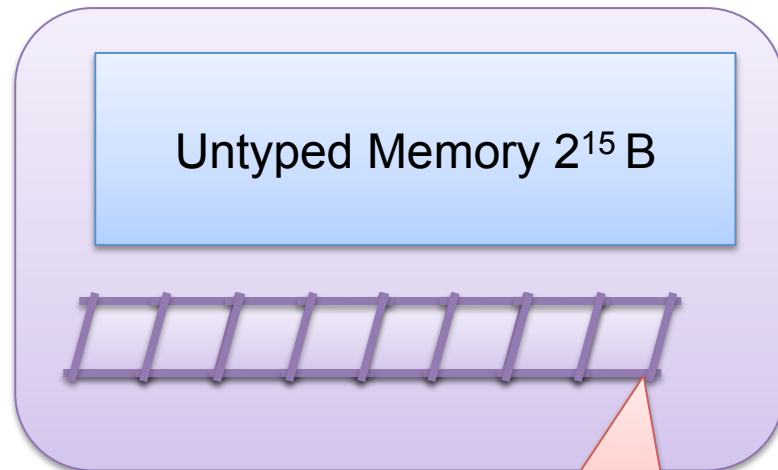
Implementation choice!



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!!



8 frames

- 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.



Threads

- Threads are represented by TCB objects
- They have a number of attributes (recorded in TCB):
 - VSpace: a virtual address space
 - page directory 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*
 - *Time slice length* (presently a system-wide constant)
- These must be explicitly managed
 - ... we provide an example you can modify

Yes, this is broken!
Fixed in later
kernels



Threads

Creating a thread

- Obtain a TCB object
- Set attributes: `Configure()`
 - associate with VSpace, CSpace, fault EP, prio, define IPC buffer
- 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 */
seL4_Word tcb_addr = ut_alloc(seL4_TCBBits);

err = cspace_ut_retype_addr(tcb_addr, seL4_TCBOobject, seL4_TCBBits,
                           cur_cspace, &tcb_cap)
err = seL4_TCB_Configure(tcb_cap, FAULT_EP_CAP, PRIORITY,
                        curspace->root_cnode, seL4_NilData,
                        seL4_CapInitThreadPD, seL4_NilData,
                        PROCESS_IPC_BUFFER, ipc_buffer_cap);
seL4_UserContext context = { .pc = &thread, .sp = &stack };
seL4_TCB_WriteRegisters(tcb_cap, 1, 0, 2, &context);
```

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 seL4_ARM_PageDirectoryObject of size seL4_PageDirBits
 * Mint a new badged cap to the syscall endpoint
 */
ospace_t * new_cspace = ut_alloc(seL4_TCBBits);

char *elf_base = cpio_get_file(_cpio_archive, "test")->p_base;
err = elf_load(new_pagedirectory_cap, elf_base);
unsigned int entry = elf_getEntryPoint(elf_base);

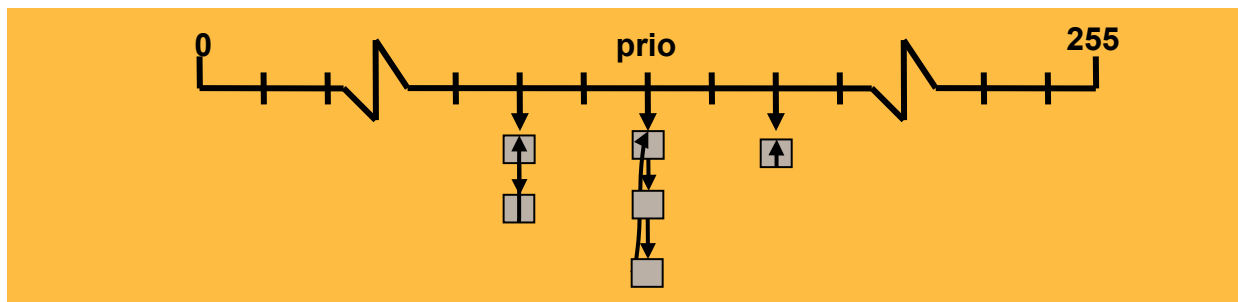
err = seL4_TCB_Configure(tcb_cap, FAULT_EP_CAP, PRIORITY,
                        new_cspace->root_cnode, seL4_NilData,
                        new_pagedirectory_cap, seL4_NilData,
                        PROCESS_IPC_BUFFER, ipc_buffer_cap);
seL4_UserContext context = {.pc = entry, .sp = &stack};
seL4_TCB_WriteRegisters(tcb_cap, 1, 0, 2, &context);
```



seL4 Scheduling

Better model in
“RT” branch –
merge soon

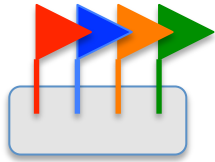
- Present seL4 scheduling model is fairly naïve
- 256 hard priorities (0–255)
 - Priorities are strictly observed
 - The scheduler will always pick the highest-prio runnable thread
 - Round-robin scheduling within prio level
- Aim is real-time performance, **not** fairness
 - Kernel itself will never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



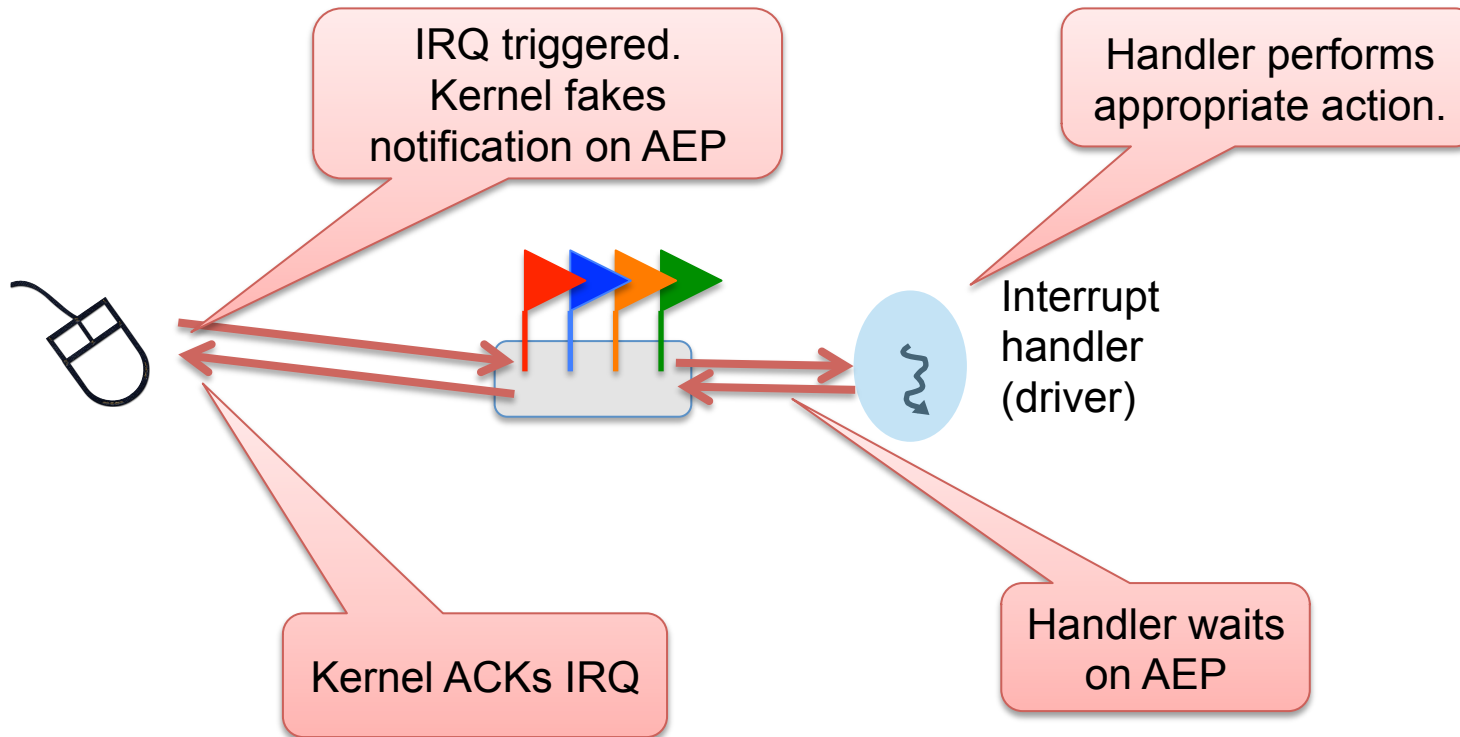


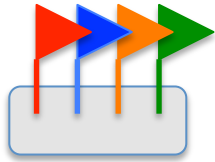
Exception Handling

- 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!



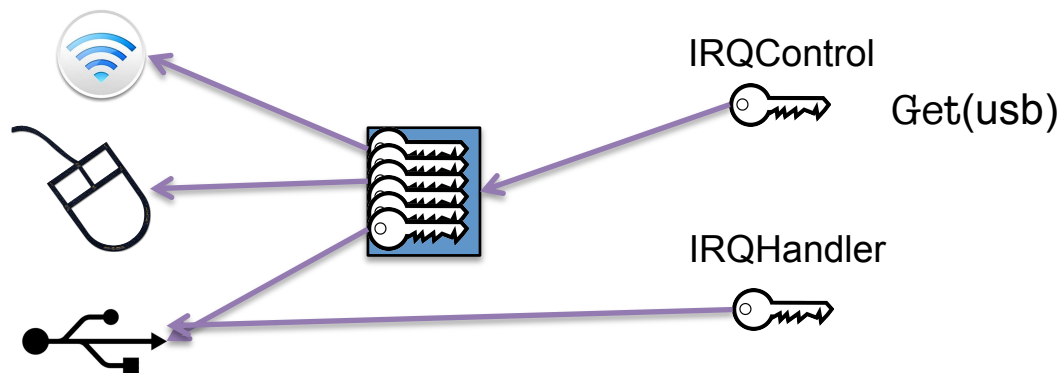
Interrupt Handling

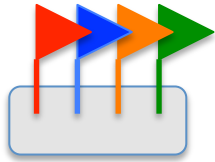




Interrupt Management

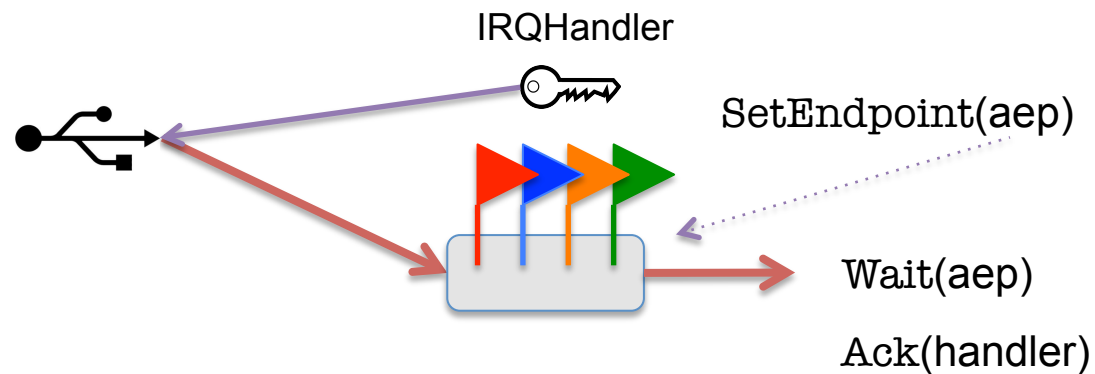
- seL4 models IRQs as messages sent to an AEP
 - Interrupt handler has Receive cap on that AEP
- 2 special objects used for managing and acknowledging interrupts:
 - Single IRQControl object
 - single IRQControl cap provided by kernel to initial VSpace
 - only purpose is to create IRQHandler caps
 - Per-IRQ-source IRQHandler object
 - interrupt association and dissociation
 - interrupt acknowledgment





Interrupt Handling

- IRQHandler cap allows driver to bind AEP to interrupt
- Afterwards:
 - AEP is used to receive interrupt
 - IRQHandler is used to acknowledge interrupt



```
seL4_IRQHandler interrupt = cspace_irq_control_get_cap(cur_cspace,  
                                                       seL4_CapIRQControl, irq_number);  
seL4_IRQHandler_SetEndpoint(interrupt, async_ep_cap);  
seL4_IRQHandler_ack(interrupt);
```

ACK first to
unmask IRQ



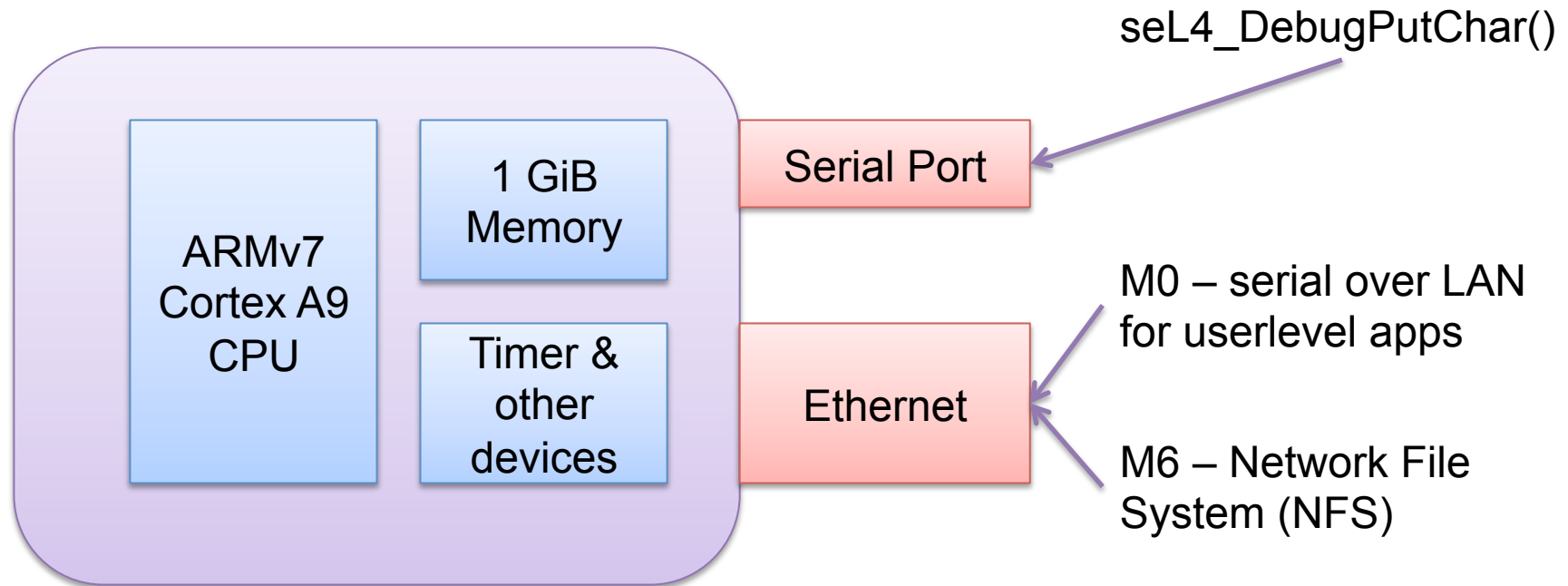
Device Drivers

- 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 = map_device(0xA0000000, (1 << seL4_PageBits));  
...  
*((void *) device_vaddr = ...;
```

Magic device
register access

Project Platform: i.MX6 Sabre Lite



Case 4 in the Real World (Courtesy Boeing, DARPA)

