CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Spring 2016

Week 8: seL4 - capabilities in practice

Primary focus

- Review main features of the seL4 microkernel
  - With some implementation hints: not exactly what you'll find in the seL4 source code … but representative
- Based on publicly distributed descriptions (particularly seL4 documentation and code from http://sel4.systems, but also NICTA slides, Dhamika Elkaduwe's dissertation, …)
  - In particular, some of the following slides (the ones with NICTA and UNSW logos) are taken from a presentation called "Introduction to seL4" and are used here courtesy of Gernot Heiser, UNSW.

http://www.cse.unsw.edu.au/~cs9242/14/lectures/01-intro.pdf

UNSW AUSTRALIA

NICTA

COMP9242
Advanced Operating Systems

S2/2014 Week 1:
Introduction to seL4

Australian Government

NSW Trade & Investment

Victoria

Queensland Government

http://www.cse.unsw.edu.au/~cs9242/14/lectures/01-intro.pdf

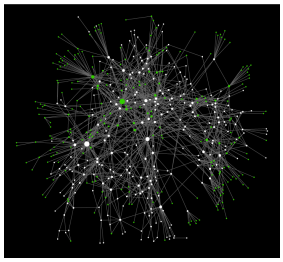
Copyright Notice

NICTA

UNSW AUSTRALIA

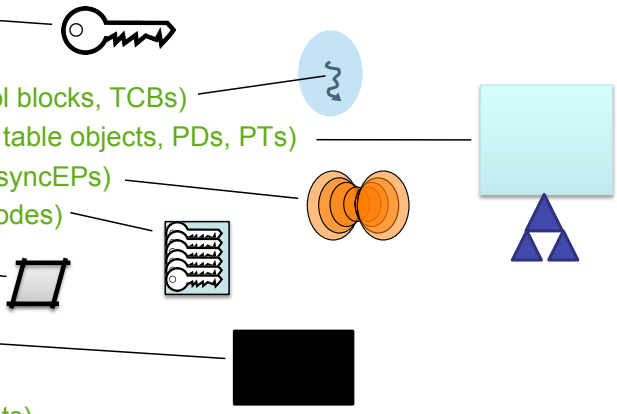
seL4 Principles

- Single protection mechanism: capabilities
  - Except for time ☹
- 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

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



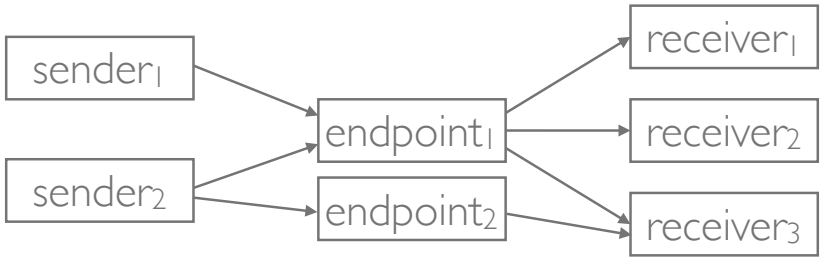
IPC and Endpoints

How to support capability-based IPC?

- How can interprocess communication (IPC) be controlled and protected using capabilities?
- One option would be to provide capabilities to TCB objects
  - This might be useful for other purposes anyway (e.g., reading/modifying thread status, starting, suspending, …)
  - Could use send / receive permissions on TCB capabilities to determine which IPC actions are allowed
- But this is also inflexible:
  - Single thread to single thread communication is limiting
  - Lacks fine-grained control: if you can contact a thread for *one* purpose, you can contact it for *any* purpose

IPC via endpoints

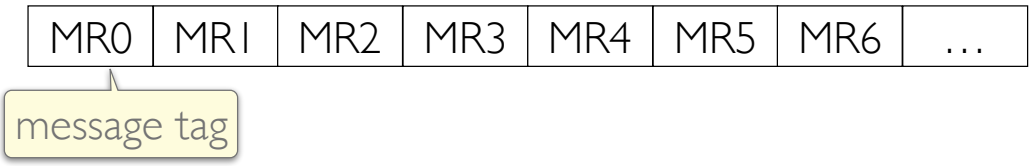
- Interprocess communication (IPC) in seL4 passes messages between threads using (capabilities to) an **endpoint** object:



- Allows flexible communication patterns
  - multiple senders and/or receivers on a single endpoint
  - multiple endpoints between communication partners
- Messages are transferred synchronously when both sender and receiver are ready
- Multiple senders or receivers can be queued at each endpoint

IPC messages

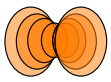
- Each thread can have a region of memory in its address space that is designated as its "IPC buffer"
- The IPC buffer holds "Message Registers" (MRs)



- Each thread can read or write values directly in its IPC buffer
- Each MR holds a single 32 bit word
- Some of the slots in the IPC buffer are reserved for sending or receiving capabilities via IPC

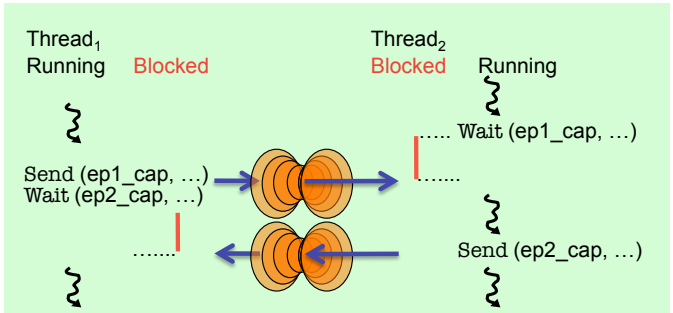
Typical IPC process

- Sending thread writes message into its IPC buffer and invokes a Send system call using a capability to an endpoint
- Receiving thread invokes a Receive/Wait system call using a capability to the same endpoint
- When both parties are ready, the kernel copies the message from the sender's MRs to the receiver's MRs
- A small number of MRs are passed in CPU registers, which is fast and avoids the need for an IPC buffer

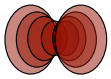


Synchronous Endpoint

NICTA

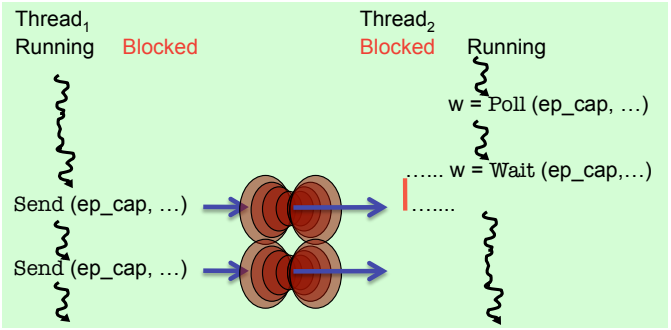


- Threads must rendez-vous for message transfer
  - One side blocks until the other is ready
  - Implicit synchronisation
- Message copied from sender's to receiver's *message registers*
  - Message is combination of caps and data words
    - presently max 121 words (484B, incl message "tag")



Asynchronous Endpoint

NICTA

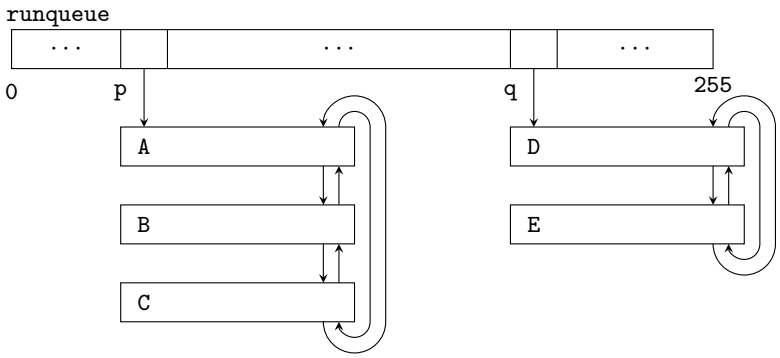


- Avoids blocking
  - send OR-s cap badge to AEP's *data word*
  - no caps can be sent
- Receiver can poll or wait
  - waiting returns and clears data word
  - polling just returns data word
- Similar to interrupt (with small payload, like interrupt mask)

UNSW AUSTRALIA

Linking TCBs

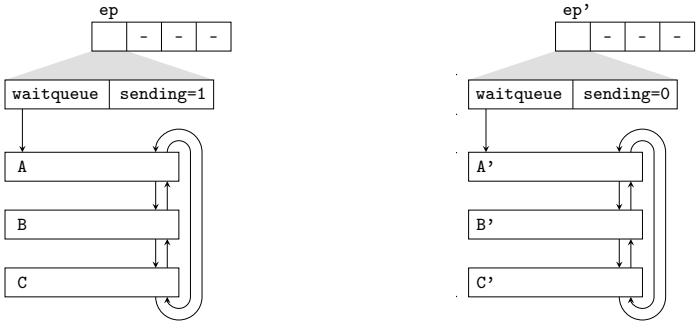
- Every TCB includes space for two pointers that can be used to include the TCB in a doubly linked list in the run queue



- What should we do if a thread blocks, waiting for an IPC message to be transferred?

Endpoints are thread queues

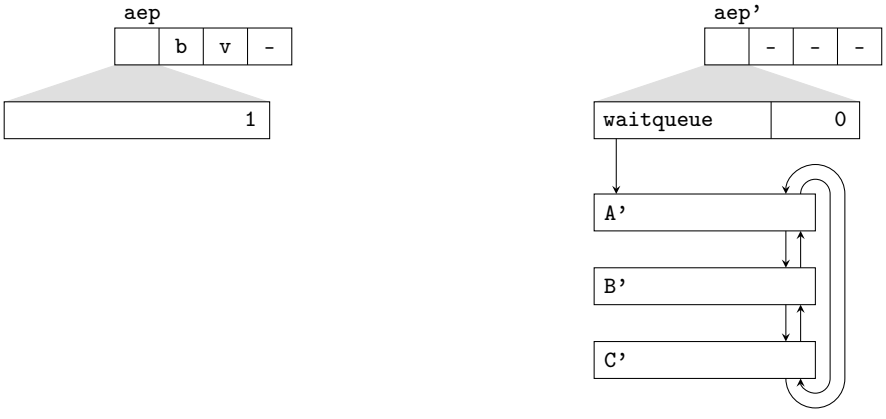
- An endpoint just provides a place to collect a queue of threads that are all waiting either to send or to receive



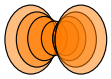
- No thread can be both runnable and blocked (waiting to send or receive a message), so one pair of TCB pointers suffices
- An endpoint doesn't require all 16 bytes of storage: that's just the smallest size allowed for any kernel object

Asynchronous endpoints

- An asynchronous endpoint provides a place to collect a queue of threads that are waiting to receive

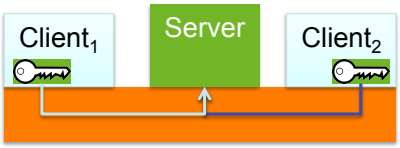


- No blocking on threads that send: the endpoint just collects the badge (b) and value (v) bits of any sender until a receiver collects them



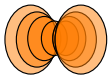
Client-Server Communication

NICTA



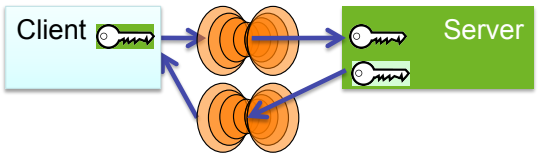
- Asymmetric relationship:
  - Server widely accessible, clients not
  - How can server reply back to client (distinguish between them)?
- Client can pass (session) reply cap in first request
  - server needs to maintain session state
  - forces stateful server design
- seL4 solution: Kernel provides 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)

UNSW AUSTRALIA



Call RPC Semantics

NICTA



Client **Kernel** **Server**

`Wait(ep,&rep)`

UNSW AUSTRALIA

System calls via IPC

- Conceptually, seL4 has just three system calls:
  - **Yield** the CPU to another thread
  - **Send** a message via a capability
  - **Receive** a message via a capability
- All other system calls are treated as special cases of Send:
  - Write parameters of system call into MRs
  - Execute a Send system call using the capability slot for the appropriate kernel object
- Allows kernel objects to be "virtualized":
  - Messages may be "received" directly by kernel object
  - … or passed to another thread via an endpoint
  - The sender will never know the difference!

Data Representation

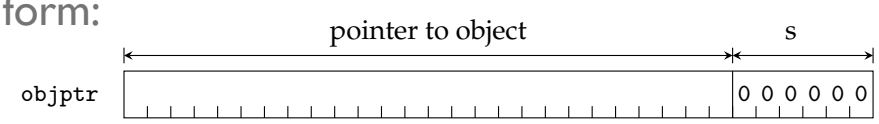
Kernel objects

The kernel deals with a range of different kernel objects:

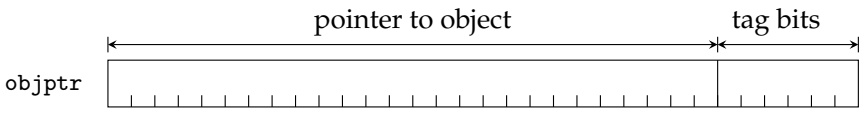
- Platform independent:
  - Untyped memory, TCBs, Endpoints (synchronous and asynchronous), CNodes, …
- Architecture specific:
  - Page table, Page directory, Page, Superpage
  - IOPort range
  - ASID (address space identifier) table
  - IRQ Handler and Control objects
  - …

Kernel object size and alignment

- Every kernel object takes 2^s bytes for some s
- All kernel objects must be size aligned:
  - If the kernel object has size 2^s, then its address must be some number of the form $2^s n$
- So every kernel address has a bit-level representation/layout of the form:

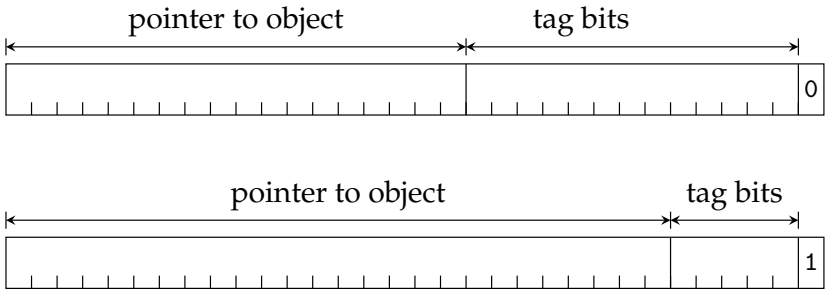


- In practice, we can use the least significant bits to store additional information:



Kernel object pointers

- The entries in each cspace table are object pointers
- We can use the low order bits to encode the type of the object that is pointed to by the high order bits
- An empty slot can be represented by a null pointer
- Different objects have different sizes; these can be integrated by using carefully designed bit-level encodings. Examples:



Overlap and disjointness properties

- If A and B are two kernel objects, then either A and B are disjoint or else one is wholly contained within the other
- A kernel object of size 4KB or less fits wholly in one page
- A kernel object of size 4MB or less fits wholly in one superpage
- New kernel objects are allocated from memory in an untyped memory object: no kernel object spans multiple untyped memory objects

Kernel object sizes

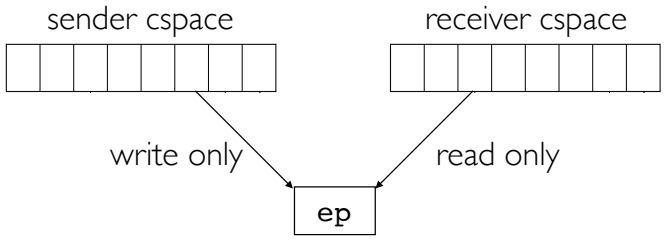
Object	Size
Untyped Memory	2^n bytes, $n \geq 2$
CNode	16×2^n bytes, $n \geq 1$
Endpoint	16 bytes
IRQ Handler	-
Thread Control Block (TCB)	1KB
IA32 4K Frame (page)	4KB
IA32 4M Frame (superpage)	4MB
IA32 Page Directory	4KB
IA32 Page Table	4KB
IA32 ASID Table	-
IA32 Port	-

- No variable size objects
- Reserve extra fields in data structures to avoid the need for "dynamic" allocation
- No room for metadata … where can it be stored?

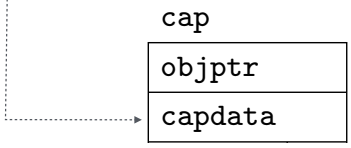
Capability Metadata

Storing metadata in capabilities

- The same endpoint may be accessed via multiple capability entries, with different access permissions

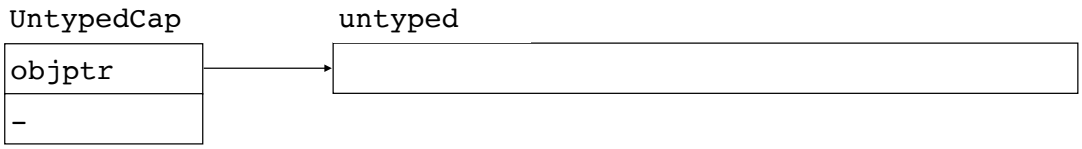


- The obvious place to store the permission settings is in the individual capability objects

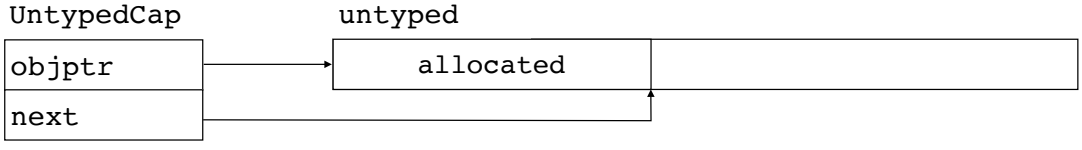


Metadata for untyped memory

- In early designs, there was no metadata for untyped memory

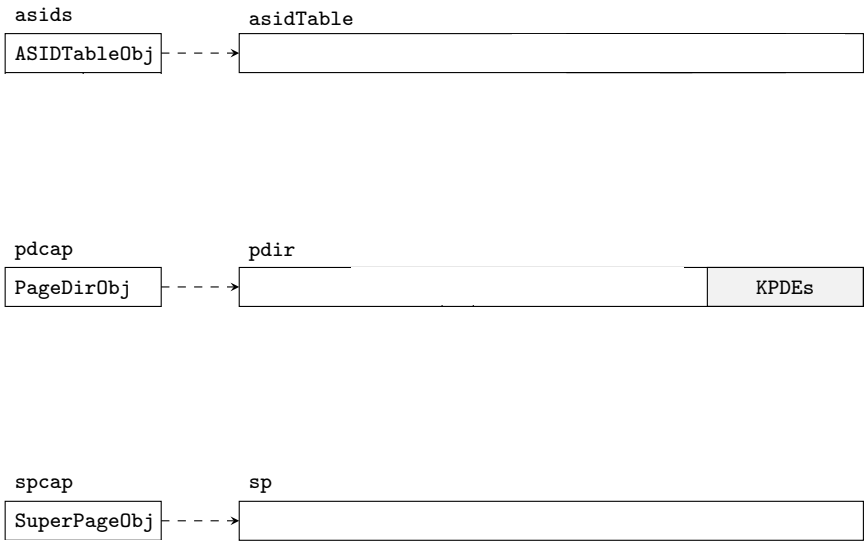


- At some point, somebody realized that the metadata could be used to store a `next` pointer

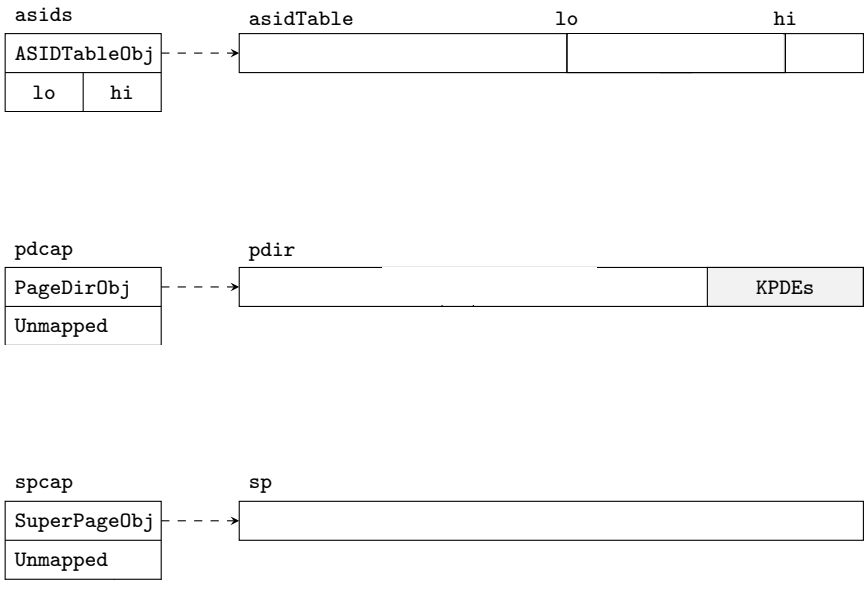


- Complication: we cannot have multiple capability objects pointing to the same untyped memory with different `next` pointers

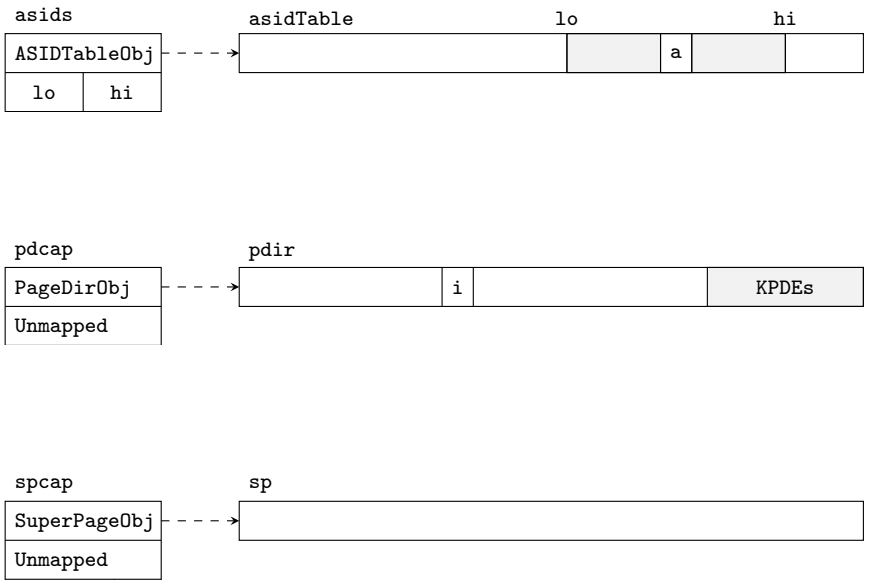
Paging structures



Paging structures, with metadata

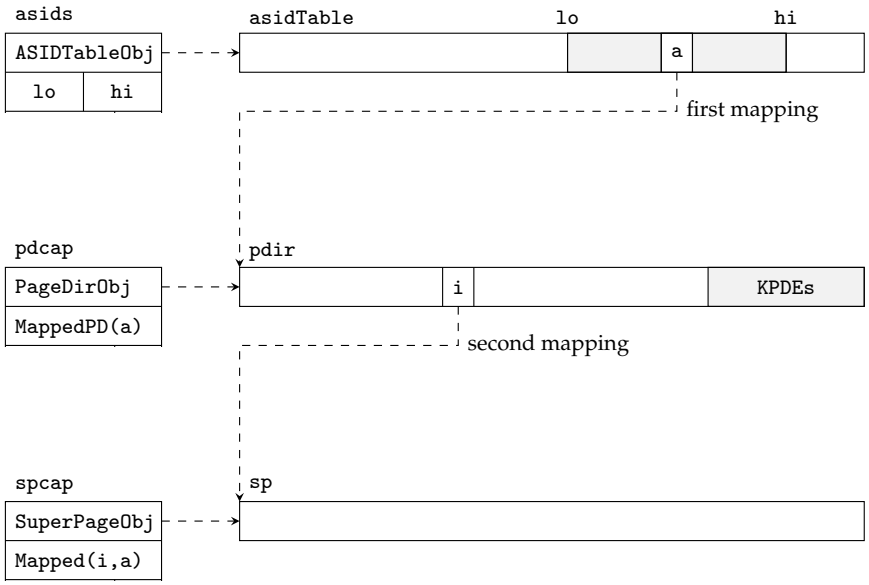


Paging structures, with metadata



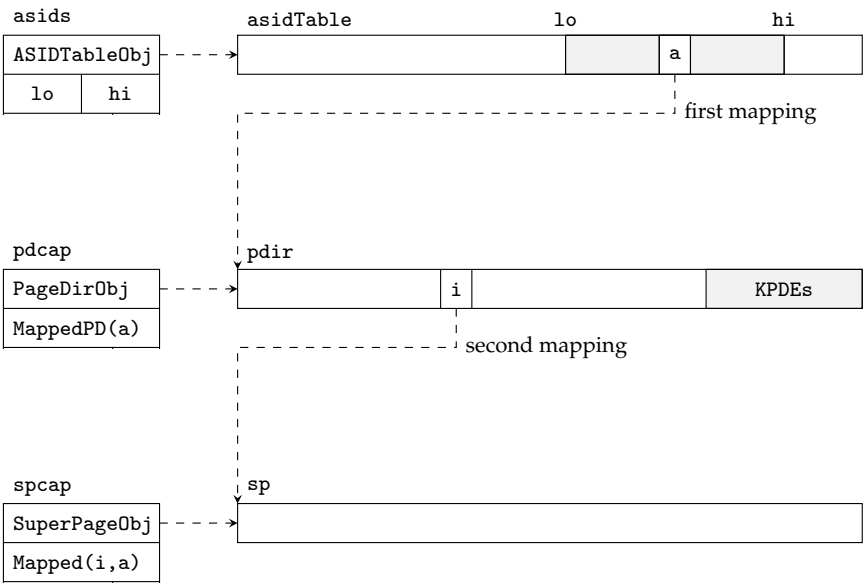
Suppose we want to associate `pdir` with address space `a`, and then map `sp` at index `i` in `pdir`

Paging structures



The metadata in `spcap` can be used to locate the appropriate page directory if the user subsequently unmaps `spcap`

Paging structures



Multiple copies of `spcap` needed to map `sp` in multiple places: unbounded storage at the expense of user level complexity

Metadata summary

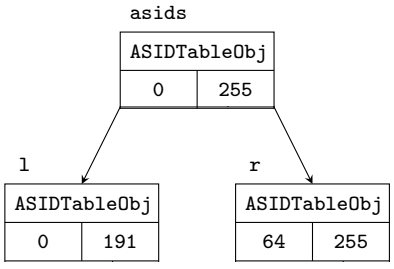
Object	Size	Metadata
Untyped Memory	2^n bytes, n≥2	high water pointer
CNode	16 x 2^n bytes, n≥1	guard
Endpoint	16 bytes	permissions, badge
IRQ Handler	-	IRQ number
Thread Control Block	1KB	permissions
IA32 4K Frame (page)	4KB	ASID and virtual address for where this object is mapped, if any
IA32 4M Frame	4MB	
IA32 Page Directory	4KB	
IA32 Page Table	4KB	
IA32 ASID Table	-	lo and hi range
IA32 Port	-	port number

- A single word of metadata goes a long way …

Derived Capabilities

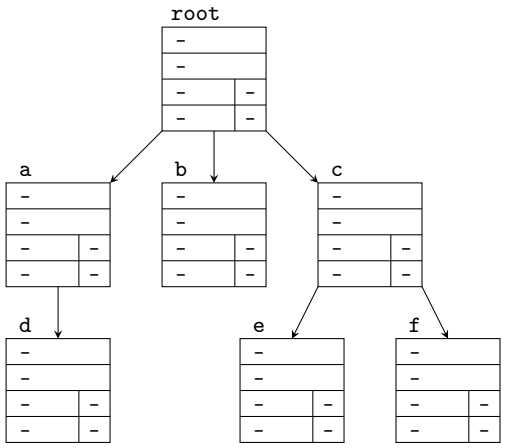
Derived capabilities

- In some situations, we might want to create derived versions of a capability with restricted permissions



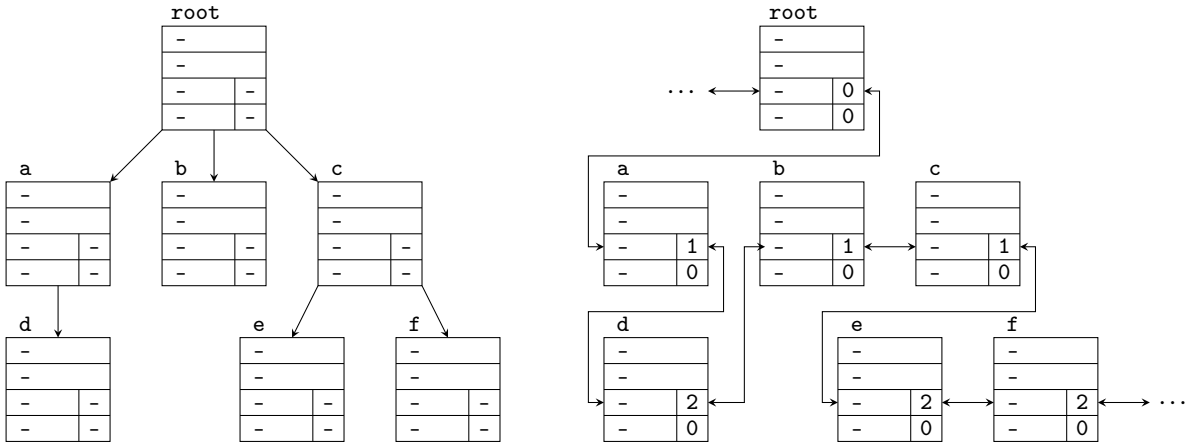
- Another example: root task creates a new endpoint and then hands out two copies of that capability to child threads, one with write permission and one with read permission, to implement a form of "pipe"
- The resulting structure is called the capability derivation tree or CDT

Representing the capability derivation tree



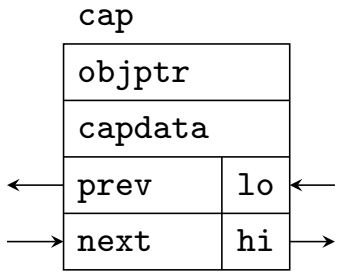
- CDT nodes can have arbitrarily many children
- A conventional implementation would require:
  - unbounded storage per node
  - unbounded recursion (stack) to traverse all children

Representing the capability derivation tree



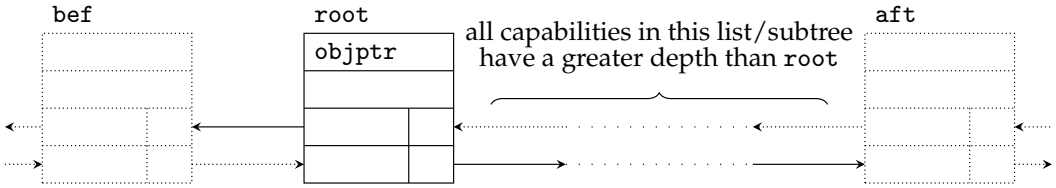
- A clever implementation represents the tree as a doubly linked list with "depth" information at each node
- Fixed storage (two pointers + depth) per node
- (Limited) traversal of tree structure without recursion

General form



- Every capability holds:
  - a pointer to a kernel object + bits giving the object type
  - some metadata
  - doubly linked list pointers
  - depth information (hi and lo bits)
- Total size: 4 words, 16 bytes
- This is why a CNode with 2^n entries requires 16 x 2^n bytes

Visiting a subtree



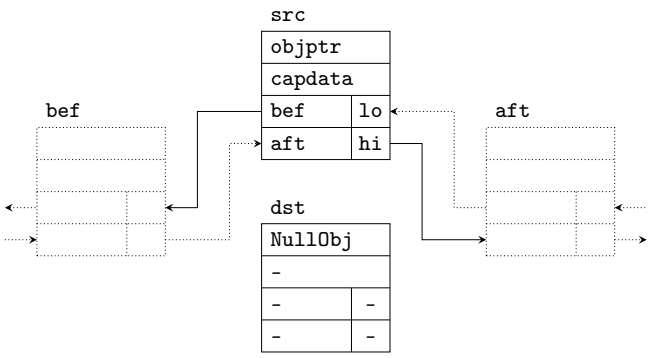
- Pattern for traversing the descendants of a capability:

```
visitChildren(root) {
    curr = root.next;
    while (curr≠null && curr.depth>root.depth) {
        … curr is a child of root …
        curr = curr.next;
    }
}
```

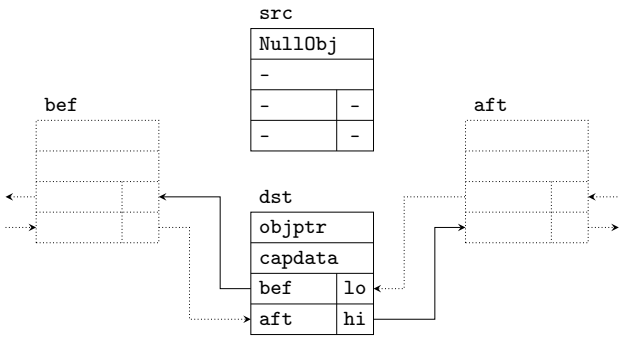
- Typical uses: revoking or deleting a capability

Moving capabilities

Before

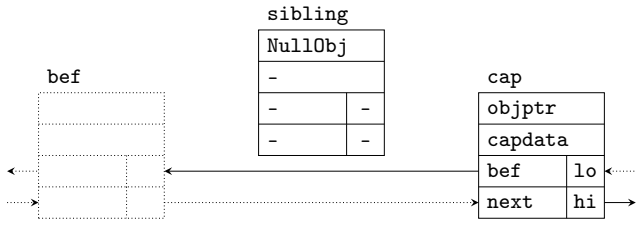


After

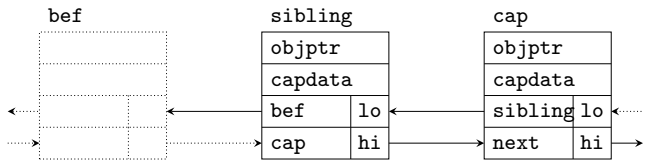


Inserting a sibling

Before

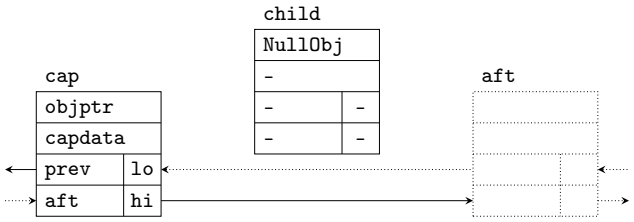


After

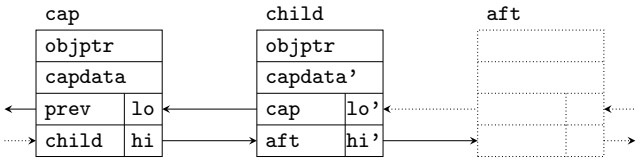


Inserting a child

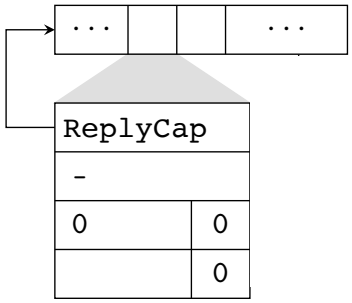
Before



After

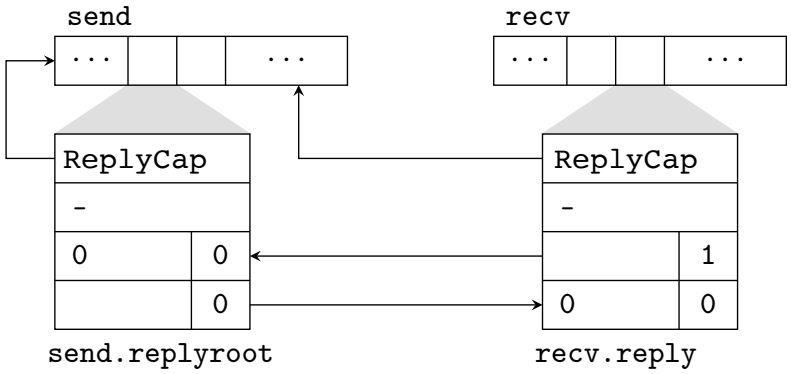


Application: implementing reply capabilities



- Reply capabilities are a new capability type that store a pointer to the sending TCB
- Every TCB contains two capability slots:
  - a "replyroot" capability that holds a ReplyCap
  - a "reply" slot that is initially empty

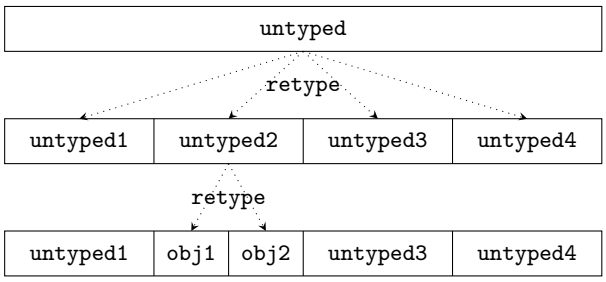
Application: implementing reply capabilities

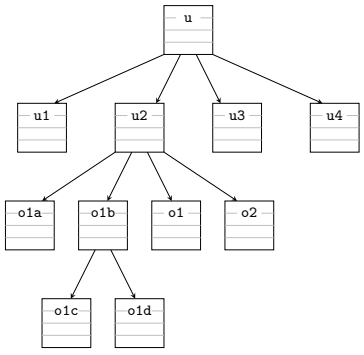


- If one thread makes a "Call" to another, the kernel will insert a child of the sender's master capability in receiver's reply slot
- The receiver can use a "Reply" system call to send a message back to the sender, without knowing its identity
- The kernel can revoke the master reply capability, to remove the child, even if the receiver has moved it to a different slot

Application: allocating from untyped memory

- Retyping is a fundamental operation that user-level threads can use to repurpose an untyped memory area





- Kernel tracks use via the "capability derivation tree" (CDT)
- Cannot retype an untyped memory area if it is already in use (i.e., if it has children in the CDT)

The retype system call

```
seL4_Untyped_Retype(
  CPtr   service,
  int    type,
  int    size_bits,
  CPtr   root,
  int    node_index,
  int    node_depth,
  int    node_offset,
  int    num_objects)
```

Capability to untyped memory

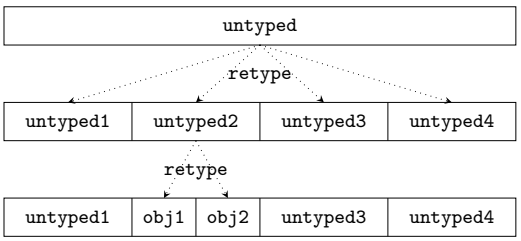
Type of object to create

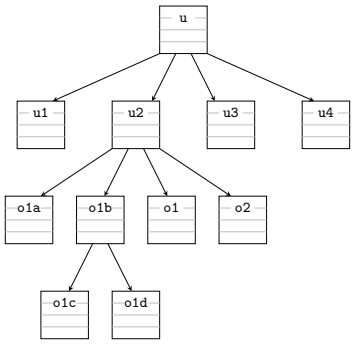
CNode where new capabilities should be stored

Window in CNode where new capabilities should be stored

Operations on capabilities

- Copy or Move (possibly reducing rights/permissions)
- Revoke (remove any derived capabilities in the CDT)



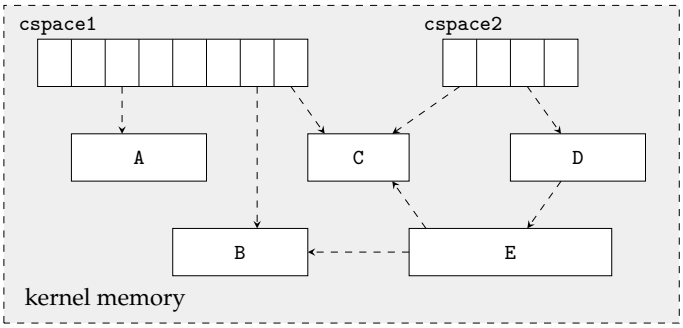


- Delete (and free storage for the associated object if this was the last capability for that object)
- Invoke (respond to a message: corresponds to a system call on a kernel object)

Capability Spaces

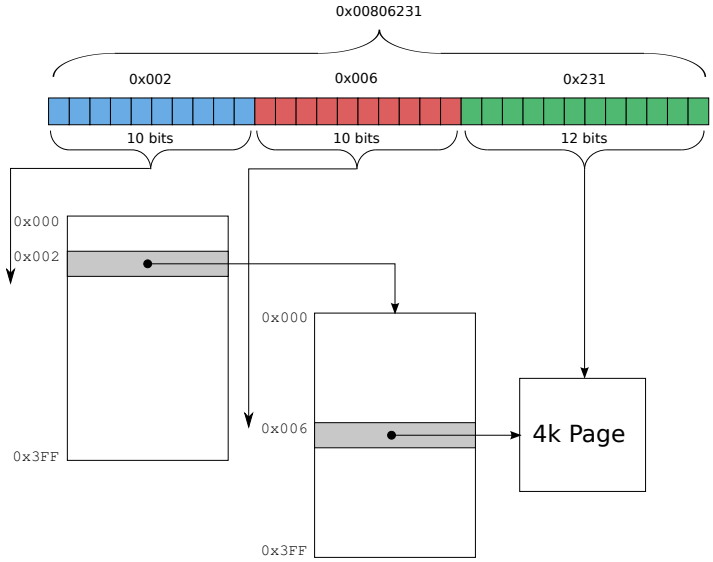
Capability spaces

- Every thread has a "capability space", which is a table mapping capability indexes to kernel objects

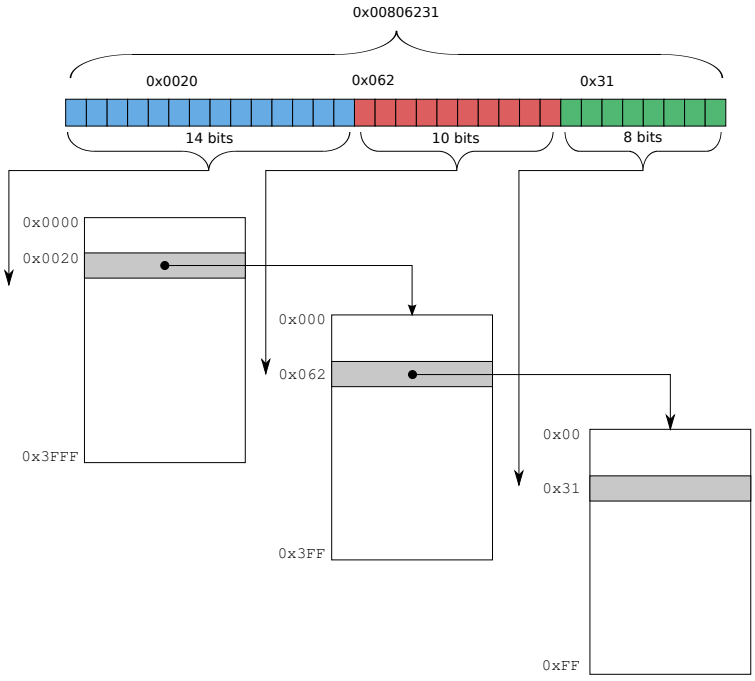


- If a thread doesn't have a capability to an object in its capability space, then it cannot directly access that object
- (cf. if there is no mapping to a particular physical address in a thread's address space, then it cannot access that location)

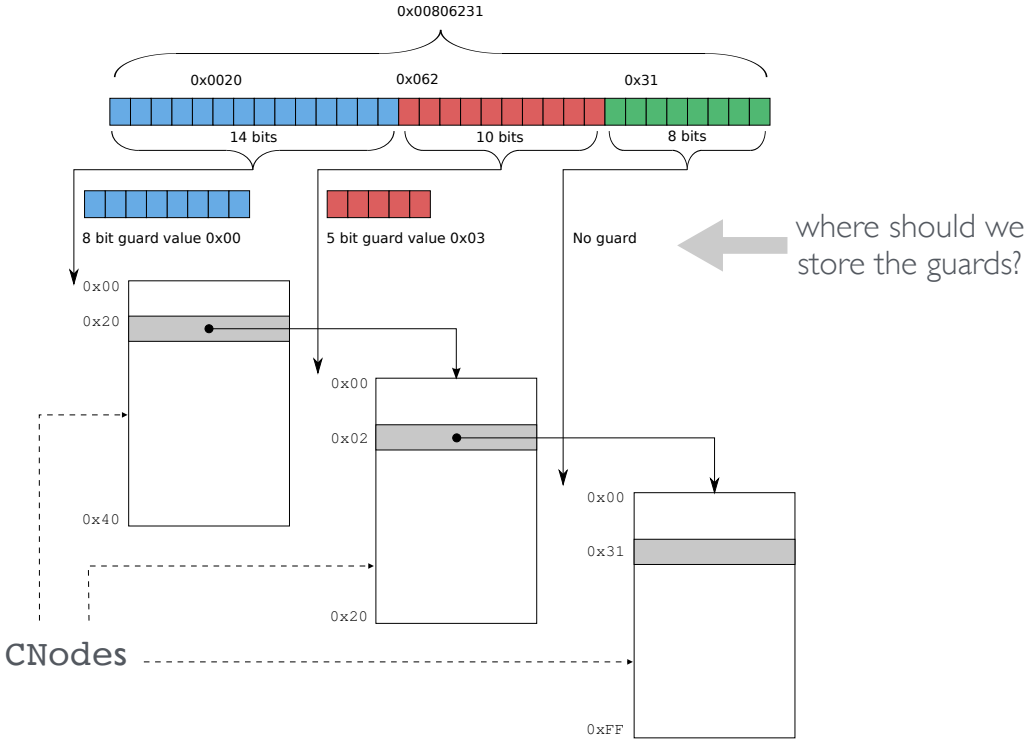
Page tables



"Page tables" for capabilities

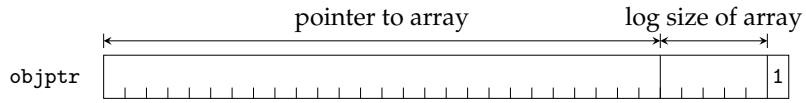


"Guarded page tables" for capabilities

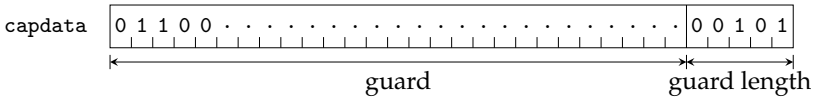


Representing CNodes

- An object pointer to a `CNode` includes the size of the `CNode` as part of the pointer representation

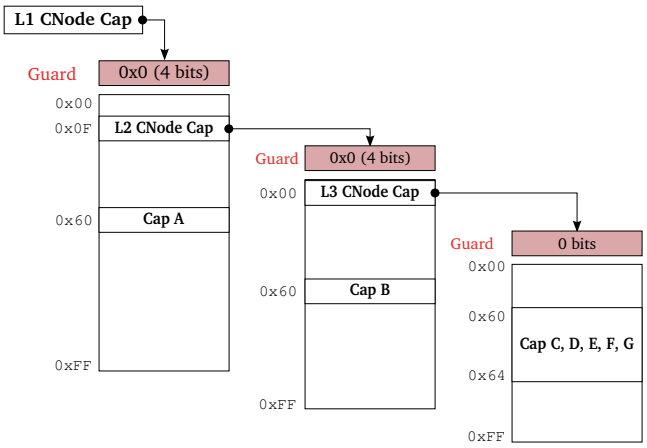


- The `capdata` for a `CNode` specifies a guard



- (This is not the exact representation used in seL4, but is sufficient to illustrate the key concepts)

General capability addressing



Cap	Address (hex)
A	`0_60_XXXXX`, depth 32
B	`0_0F_0_60_XX`, depth 32
C	`0_0F_0_00_60`, depth 32
C-G	`0_0F_0_00_60`, window size 5
L2 CNode	`0_0F_XXXXX`, depth 12
L3 CNode	`0_0F_0_00_XX`, depth 24

- General form of capability address uses:
  - a 32 bit "root" `CPtr` to a `CNode` in the caller's cspace
  - An index, relative to that root
  - A depth (number of bits to decode, required for `CNode`)
  - A window size (to specify a range of capabilities)

Capability lookup

- `edi` contains the capability ptr
- `edx` contains the depth (i.e., number of significant bits) to decode
- `esi` points to the current capability root (presumably a CNodeCap)

```
        jmp     depth

        ...

depth:  # Check to see if we have exhausted the specified depth
        orl     %edx, %edx      # zero depth => done
        jz      bitTerminated

        # Read data from the capability:
        movl    (%esi), %eax    # Get pointer word in eax
        movl    4(%esi), %ebx   # Get guard word in ebx

        test    $1, %eax        # Confirm that this is a CNodeCap
        jz      objectTerminated

        # Parse the guard portion of a cptr index:
        movl    %ebx, %ecx      # Get guard length in ecx
        andl    $0x1f, %ecx
        jz      index           # Skip if there is no guard

        subl    %ecx, %edx      # Update depth to account for bits
        jl      depthMismatch   # that are matched by guard

        xorl    %edi, %ebx      # Test guard bits against cptr value
        rol     %cl, %edi       # Rotate guard bits out of the way
        negl    %ecx            # The following shift only uses the
        shr     %cl, %ebx       # least sig 5 bits, so negating ecx
                                # is equivalent to computing 32 - guard len
        jz      index
```

Capability lookup

- `edi` contains the capability ptr
- `edx` contains the depth (i.e., number of significant bits) to decode
- `esi` points to the current capability root (presumably a CNodeCap)

```
        jmp     depth

index:  # Parse the index portion of a cptr
        movl    %eax, %ecx      # Find size, s, of CNode in ecx
        shr     $1, %ecx
        andl    $0x1f, %ecx

        subl    %ecx, %edx      # Update depth to account for index bits
        jl      depthMismatch

        rol     %cl, %edi       # Rotate index into lower bits

        movl    $1, %esi        # Compute mask for lower s bits
        shl     %cl, %esi
        decl    %esi            # (2^s - 1)
        andl    %edi, %esi      # Extract index from edi
        shl     $4, %esi        # Multiply by 16 (size of a cap. slot)
        andl    $(~0x3f), %eax  # Extract pointer from eax
        orl     %eax, %esi      # And find new capability slot

depth:  ...
```


Capability lookup

- `edi` contains the capability ptr
- `edx` contains the depth (i.e., number of significant bits) to decode
- `esi` points to the current capability root (presumably a CNodeCap)

```
        jmp     depth

index:  # Parse the index portion of a cptr
        ...

depth:  ...

guardMismatch:
        # take action to indicate a guard mismatch
        jmp     done

depthMismatch:
        # take action to indicate a depth mismatch
        jmp     done

objectTerminated:
        # unable to move past a non-CNode cap
        jmp     done

bitTerminated:
        # used all of the specified number of cptr bits
        jmp     done

done:
        # esi points to the selected capability
        # edx records the number of undecoded bits
```

Performance critical?

- Efficient capability lookup is important because every system call (except Yield) requires at least one lookup operation
- Wouldn't it be nice if the hardware could do this for us? (an exercise in appreciating the role of a traditional MMU!)
- Is assembly language required to obtain good performance?
- If so, then representation transparency is also important!

Threads

Threads

- Theads 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)
    - Yes, this is broken! (Will be fixed soon…)
- These must be explicitly managed
  - … we provide an example you can modify

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()`

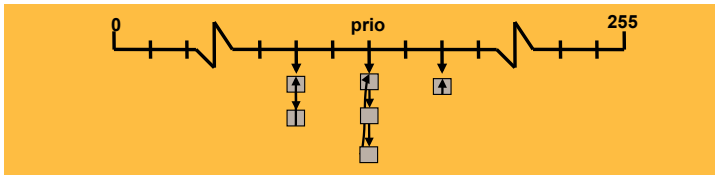
Threads and Stacks

- 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];
  . . .
}
```


seL4 Scheduling

- Presently, seL4 uses 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



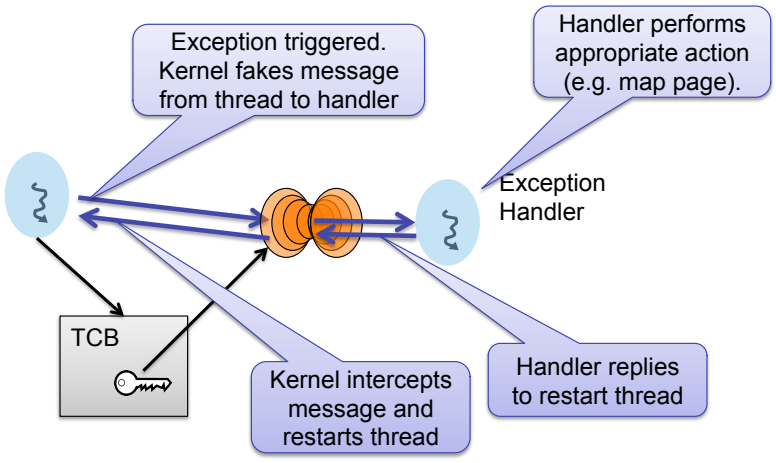
Exception Handling

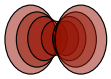
NICTA

- 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!

Exception Handling

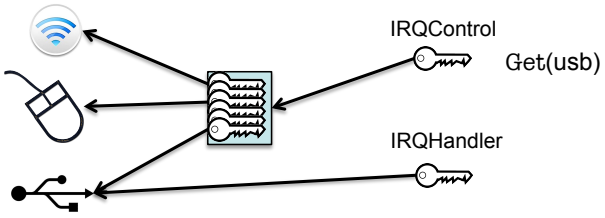
NICTA

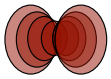




Interrupt Management

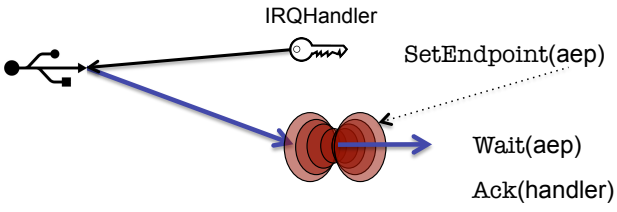
- seL4 models IRQs as messages sent to an AEP
  - Interrupt handler has Receive cap on that AEP
- 2 special objects used for managing and acknowledging interrupts:
  - Single IRQControl object
    - single IRQControl cap provided by kernel to initial VSpace
    - only purpose is to create IRQHandler 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_IRQHander_ack(interrupt);
```

Ack first to unmask IRQ

Summary

- seL4 represents nearly two decades of experience and evolution in L4 microkernel development
- Fundamental abstractions: threads, address spaces, IPC, and physical memory
- Fine-grained access control via capabilities
- Novel approach to resource management
  - no dynamic memory allocation in the kernel; shifts responsibility to user level