CS 410/510

Languages & Low-Level Programming

Mark P Jones
Portland State University

Fall 2018

Week 8: seL4 - capabilities in practice

Copyright Notice

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:
  - seL4 documentation and code from http://sel4.systems
  - Gernot Heiser's presentation on an "Introduction to seL4" [http://www.cse.unsw.edu.au/~cs9242/14/lectures/01-intro.pdf]
  - Dhamika Elkaduwe's PhD dissertation on "A Principled Approach to Kernel Memory Management" [https://ts.data61.csiro.au/publications/papers/Elkaduwe:phd.pdf]

seL4 from 30,000 feet

- A microkernel that uses capabilities throughout for access control and resource management
  - latest versions even use capabilities to manage allocation of CPU time and scheduling
- seL4 was designed with formal verification in mind, and intended to serve as a foundation for building secure systems
- Runs on ARM and IA32 platforms, among others; only the ARM version is formally verified at this time
- In practice, managing lots of capabilities by hand is painful:
  - seL4 programmers can take advantage of user-level libraries that simplify the task of working with capabilities

Kernel objects in seL4

- Types of kernel objects include:
  - Untyped memory
  - TCB objects for representing threads
  - Endpoint and Notification objects for IPC
  - Memory objects (PageDirectory, PageTable, Frame) for building address spaces
  - CNode objects for building capability spaces
  - and more ...
- Capabilities are used to manage user-level access to all of these different types of object

System calls in seL4

- Conceptually, seL4 has an "object-oriented" API with just three system calls:
  - **Send** a message to an object (via a capability)
  - **Wait** for a message from an object (via a capability)
  - **Yield** (does not require an object/capability)
- For example:
  - send a message to an Endpoint object to communicate with another thread
  - send a message to a TCB object to configure the thread
- In practice, there are other variants of Send/Wait to support combined send and receive, RPC, and other patterns

Threads

Thread Control Blocks (TCBs) in seL4

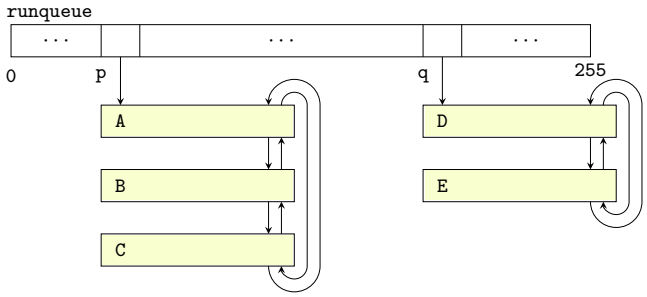
- Threads are represented in the kernel by TCB objects
- Each TCB contains:
  - A context (stores CPU register values for the thread)
  - A pointer to the virtual address space (page directory)
  - A pointer to the capability space (cspace)
  - Scheduling parameters (priority, timeslice, etc.)
  - A pointer to the IPC buffer (MRs) for the thread
  - A capability to a fault handler endpoint for the thread
  - ..
- Unlike L4: no *a priori* limit on the number of threads in an address space, no global thread ids, ...

Operations involving TCBs

- Allocate TCBs (from untyped memory)
- Configure a TCB
  - set registers, vspace, cspace, fault handler, priority, etc...
  - [If two threads run in the same address space, they should be configured to use different locations in memory for data areas, stacks, etc.]
- Resume/pause a thread
  - resume will add the thread to the run queue
  - pause will remove the thread from the run queue

The run queue

- The run queue data structure is an array of circular linked lists of TCBs for runnable threads, one for each priority:



- Every TCB includes space for the two pointers that are used to store it in the run queue (no extra storage is required)
- At a context switch, the scheduler:
  - moves the current thread to the back of its list
  - switches to the first thread in the highest priority non-empty list

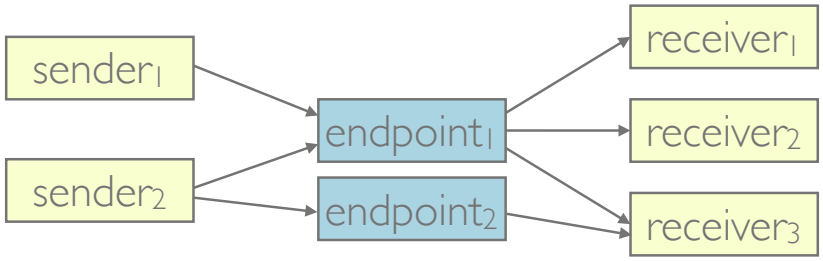
IPC and Endpoints

How to support capability-based IPC?

- How can interprocess communication (IPC) be controlled and protected using capabilities?
- One option would be to use capabilities to TCB objects
  - These are 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

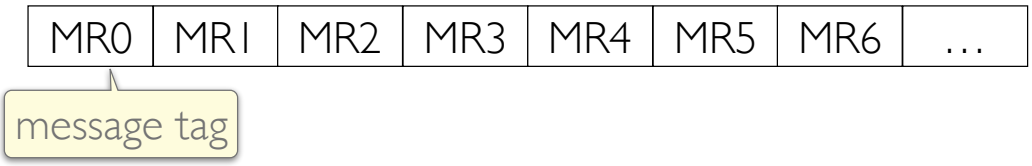
- Interprocess communication (IPC) in seL4 passes messages between threads using (capabilities to) an **endpoint** object:



- 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 ("rendez-vous")
- 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)



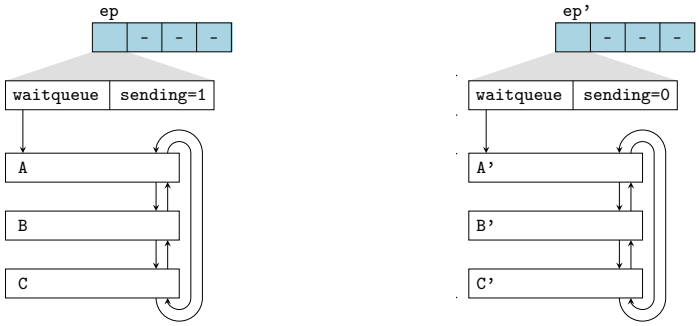
- Each thread can read or write values directly in its IPC buffer
- Each MR holds a single 32 bit word
- Some of the slots in the IPC buffer are reserved for sending or receiving capabilities via IPC

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 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

Endpoints are thread queues

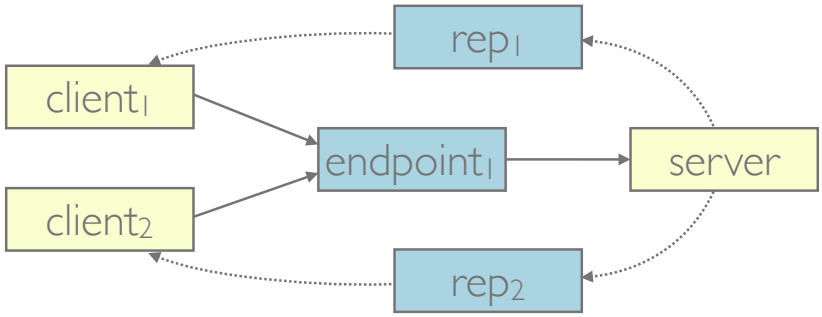
- An endpoint just provides a place to collect a queue of threads that are all waiting either to send or to receive



- 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

Client-server communication

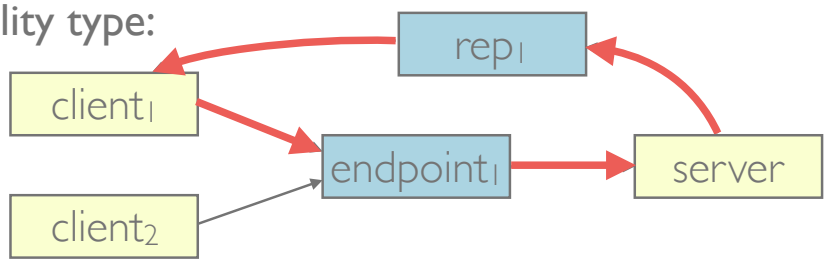
- Practical systems often use a client-server architecture in which one "server" thread performs work for many "clients"



- What if the client needs a reply? How will the server know where to send it?
- The client could send a capability to a "reply" endpoint as part of its request. But this makes extra work for the client, and could be abused by a malicious (or buggy) server.

Reply capabilities

- seL4 tackles this problem by introducing a special "Reply" capability type:



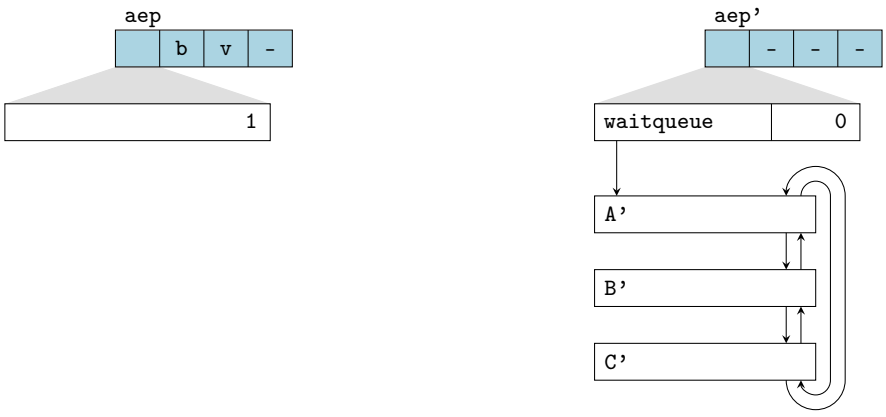
- The **Call** system call combines a **Send** and a **Wait**
- The kernel gives a new "reply capability" to the receiver
- The receiver can move but not copy the reply capability
- The receiver can send a message to the reply capability
- The reply capability is deleted after its first (hence only) use

Asynchronous (non-blocking) IPC

- seL4 also supports (limited) asynchronous/non-blocking IPC via "notification objects" (aka "Asynchronous Endpoints/AEPs)
- How is this possible without an unbounded buffer to store all messages that have been sent but not yet received?
    - Each notification object holds a single data word
    - When you Send to a notification object:
        - you provide a single word of data that is ORed with the data in the notification
        - the sender can resume immediately
    - A receiver can:
        - Poll a notification to read the current data word
        - Wait on a notification, reading and clearing the data word when data becomes available

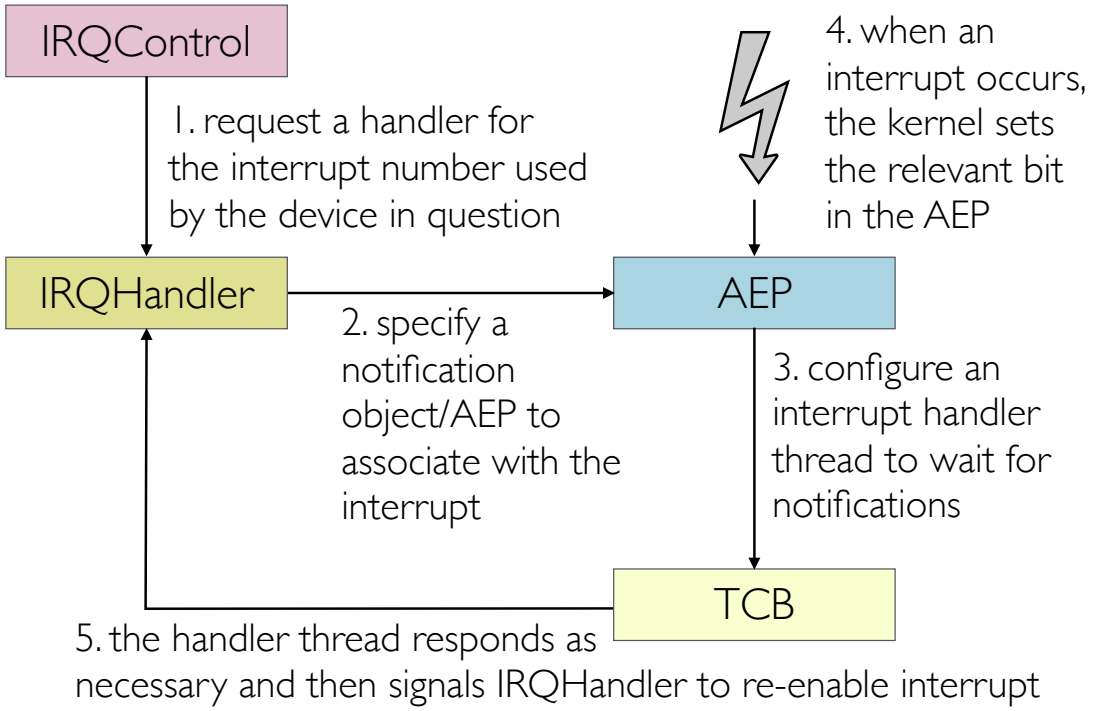
Notifications (asynchronous endpoints)

- A notification object (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

Handling hardware interrupts in seL4



Data Representation

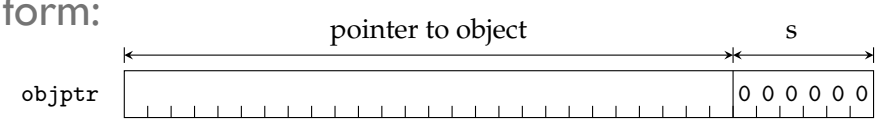
Kernel objects

The kernel deals with a range of different kernel objects:

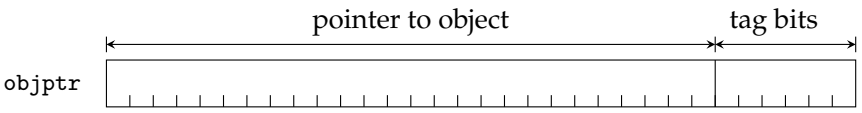
- Platform independent:
  - Untyped memory, TCBs, Endpoints (synchronous and asynchronous), CNodes, …
- Architecture specific:
  - Page table, Page directory, Page, Superpage
  - IOPort range
  - ASID (address space identifier) table
  - IRQ Handler and Control objects
  - …

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:

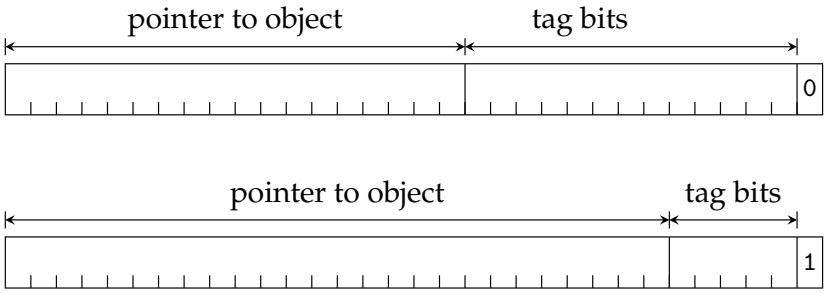


- In practice, we can use the least significant bits to store additional information:



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:



Kernel object sizes

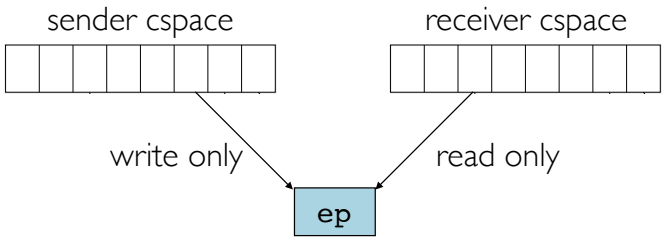
Object	Size
Untyped Memory	2^n bytes, $n \geq 2$
CNode	16×2^n bytes, $n \geq 1$
Endpoint	16 bytes
IRQ Handler	-
Thread Control Block (TCB)	1KB
IA32 4K Frame (page)	4KB
IA32 4M Frame (superpage)	4MB
IA32 Page Directory	4KB
IA32 Page Table	4KB
IA32 ASID Table	-
IA32 Port	-

- 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?

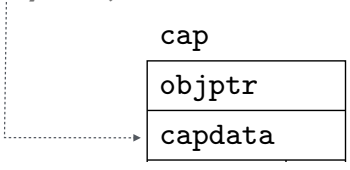
Capability Metadata

Storing metadata in capabilities

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

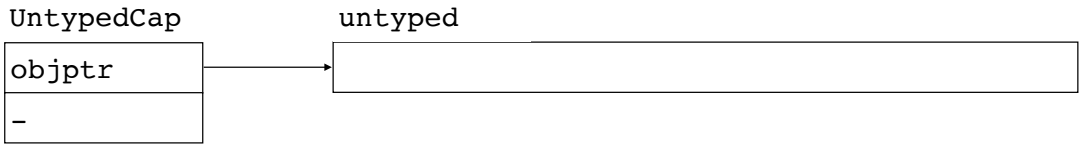


- The obvious place to store the permission settings is in the individual capability objects

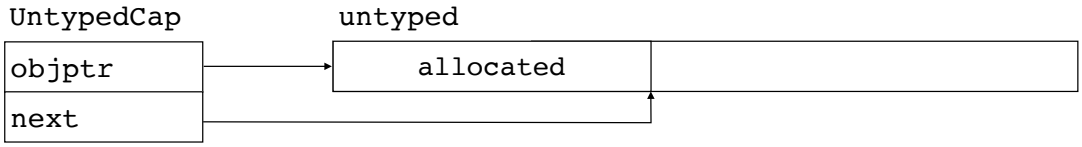


Metadata for untyped memory

- 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

System calls for managing paging structures

- Map a page in to an address space

```
seL4_IA32_Page_Map(pgcap, pdcap, vaddr, rights, attrs)
```

- Unmap a page from an address space

```
seL4_IA32_Page_Unmap(pgcap)
```

how do we find the page directory where this mapping is stored?

- Map a page table in to an address space

```
seL4_IA32_PageTable_Map(ptcap, pdcap, vaddr, attrs)
```

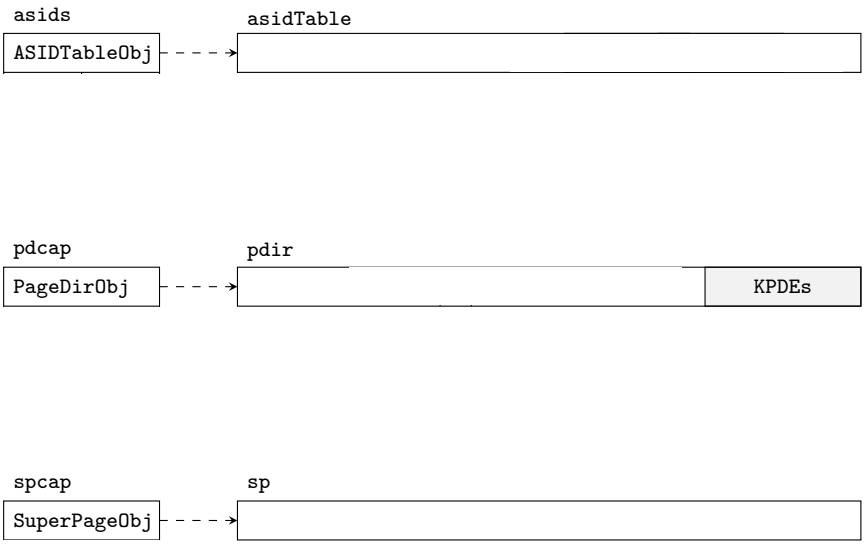
- Unmap a page table from an address space (and zero it out)

```
seL4_IA32_PageTable_Unmap(ptcap)
```

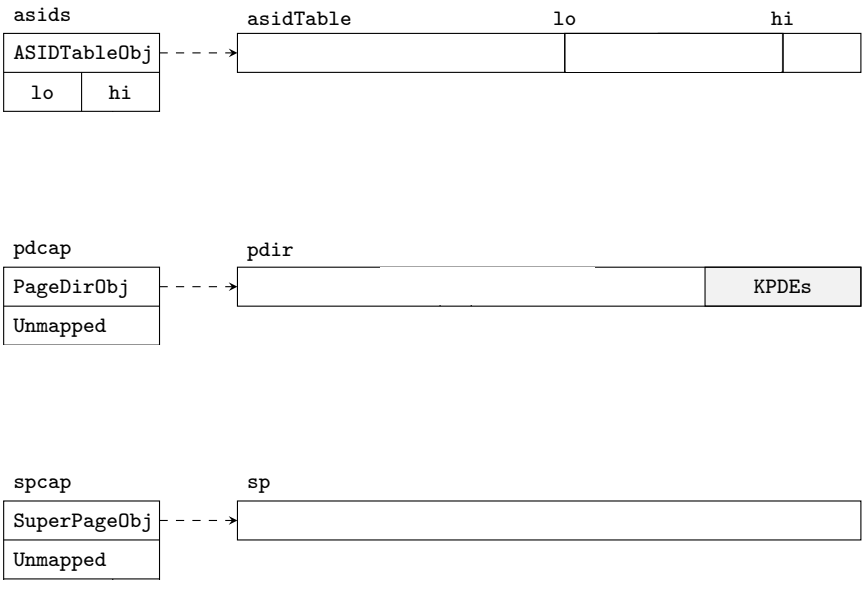
ditto

- User level code must map a page table into an address space before it can map a 4KB page

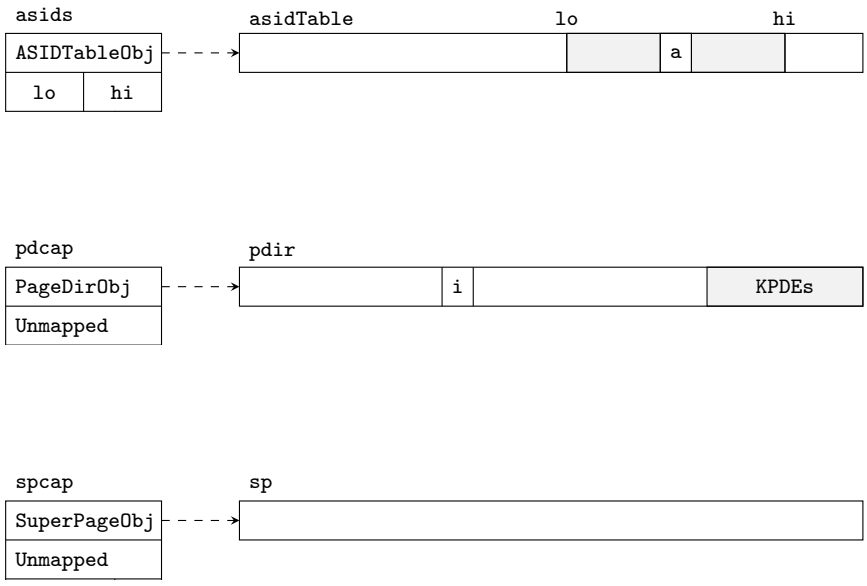
Paging structures



Paging structures, with metadata

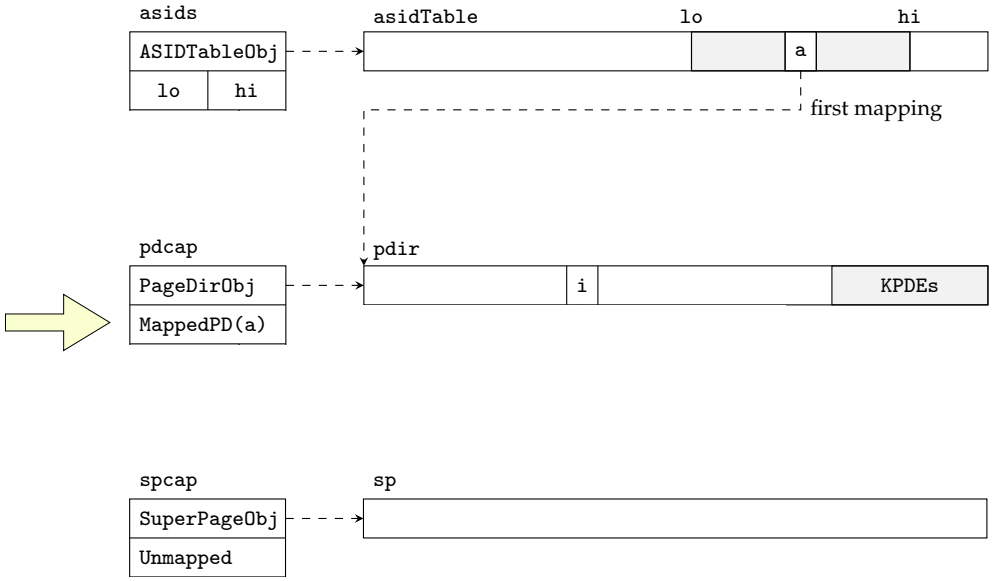


Paging structures, with metadata



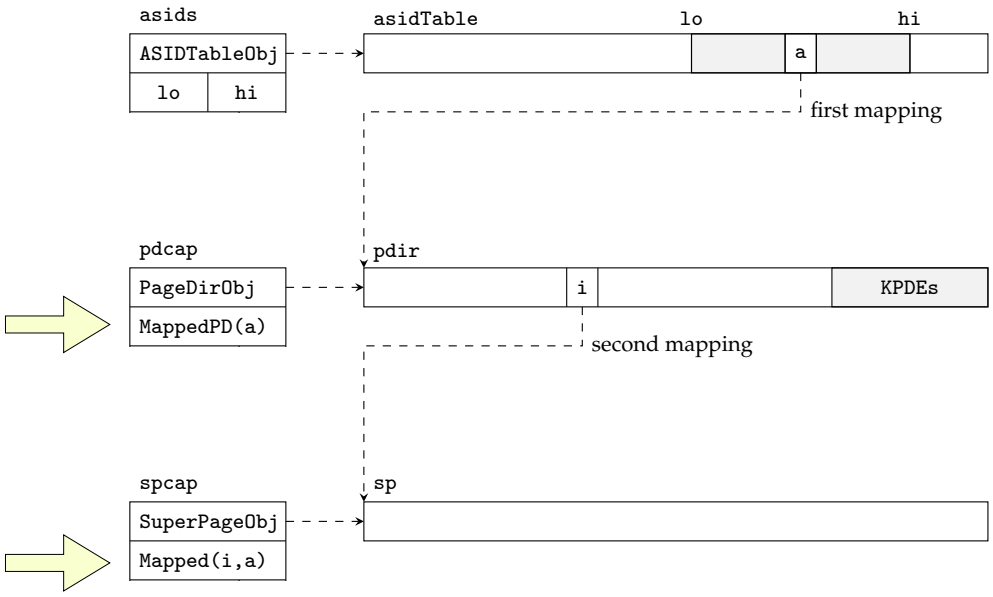
Suppose we want to associate `pdir` with address space `a`, and then map `sp` at index `i` in `pdir`

Paging structures



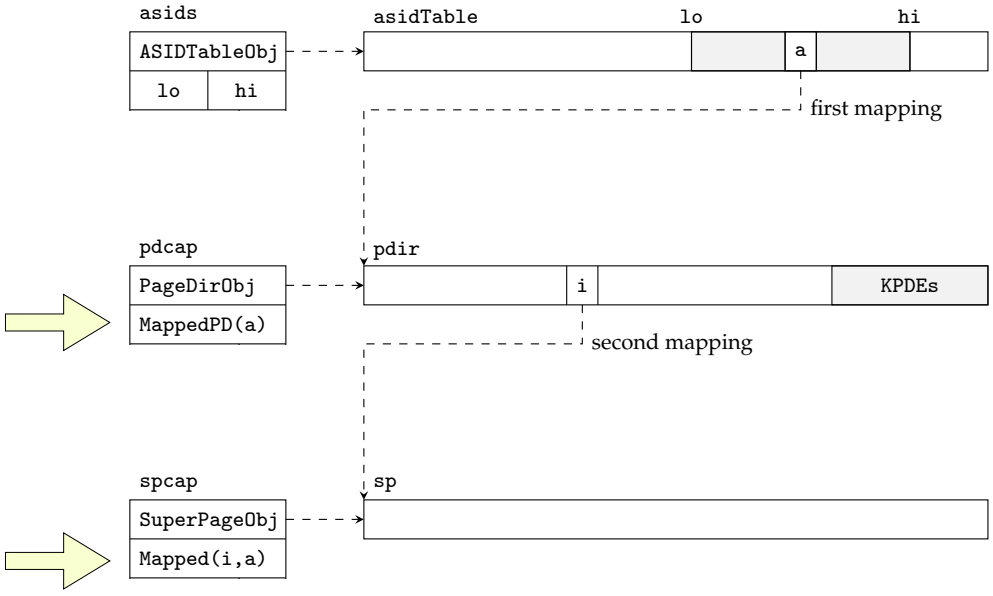
Suppose we want to associate `pdir` with address space `a`, and then map `sp` at index `i` in `pdir`

Paging structures



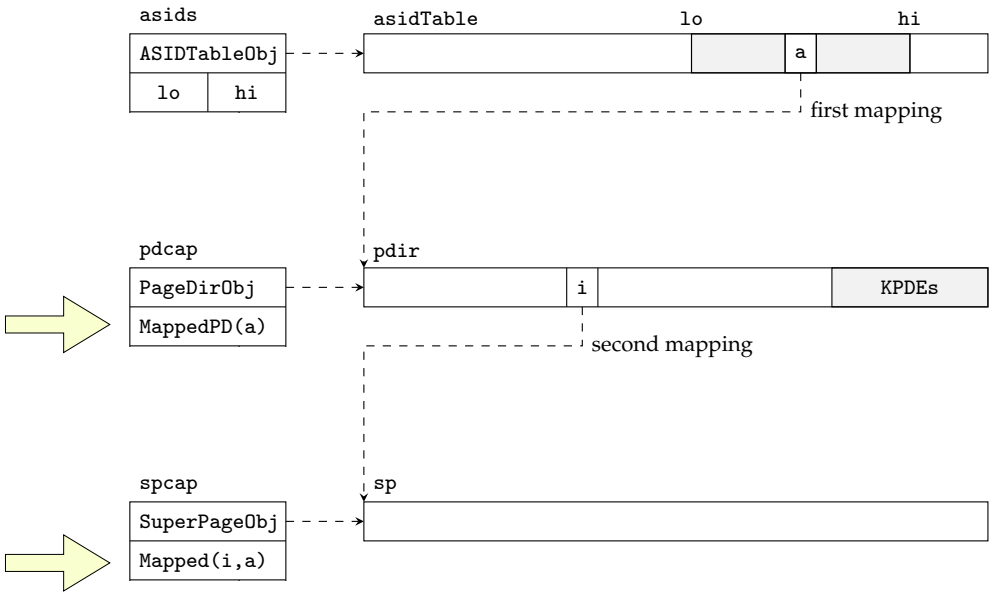
Suppose we want to associate `pdir` with address space `a`, and then map `sp` at index `i` in `pdir`

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` are needed to map `sp` in multiple places (likely increasing complexity of user level code)

Metadata summary

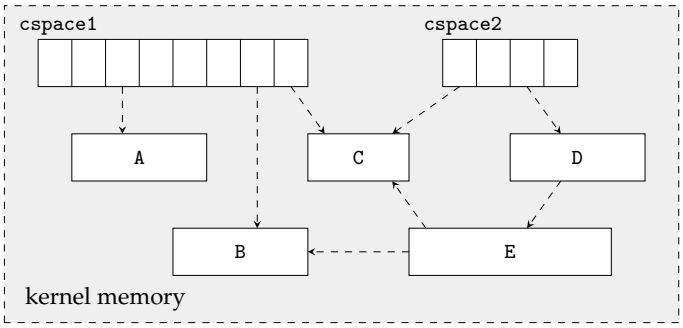
Object	Size	Metadata
Untyped Memory	2^n bytes, $n \geq 2$	"next" pointer
CNode	16×2^n bytes, $n \geq 1$	guard
Endpoint	16 bytes	permissions, badge
IRQ Handler	-	IRQ number
Thread Control Block	1KB	permissions
IA32 4K Frame (page)	4KB	ASID and virtual address for where this object is mapped, if any
IA32 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 …

Capability Spaces

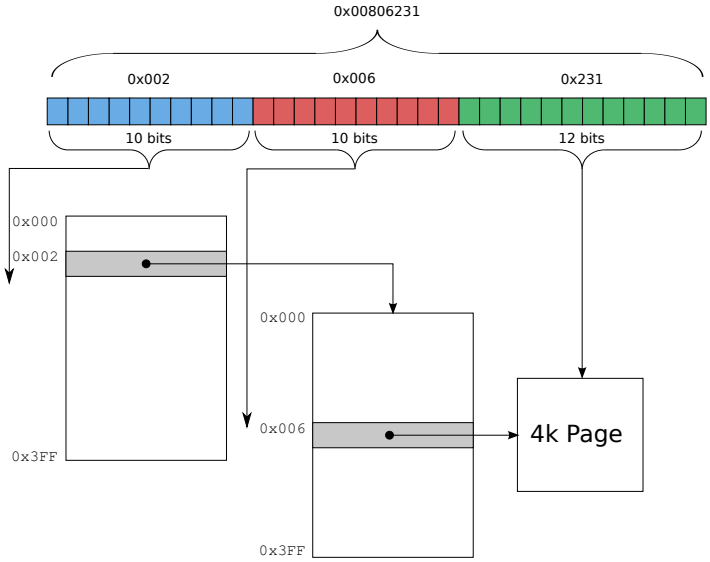
Capability spaces

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



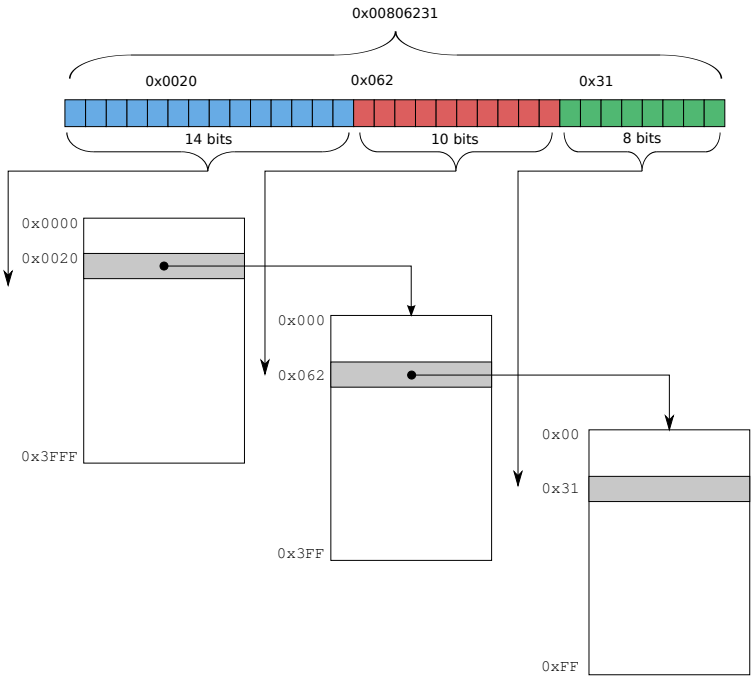
- If a thread doesn't have a capability to an object in its capability space, then it cannot directly access that object
- (cf. if there is no mapping to a particular physical address in a thread's address space, then it cannot access that location)

Page tables



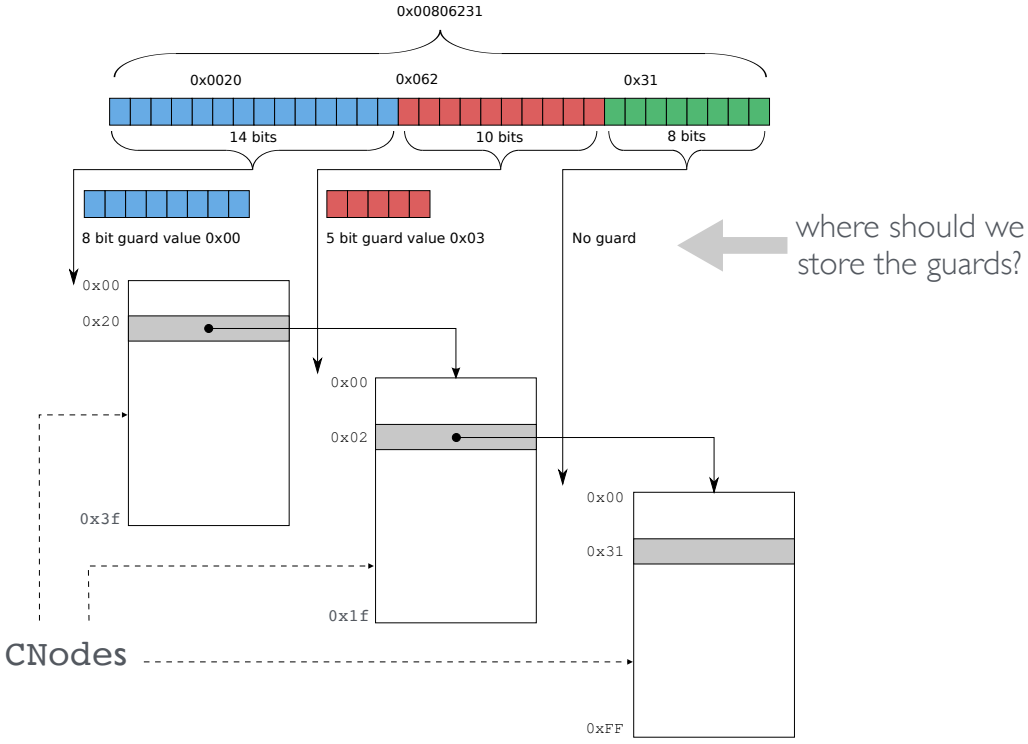
(Diagram credit: seL4 documentation)

"Page tables" for capabilities



(Diagram credit: seL4 documentation)

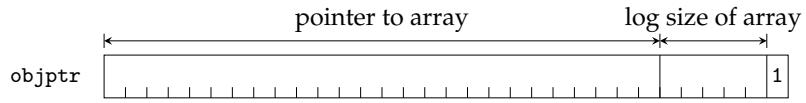
"Guarded page tables" for capabilities



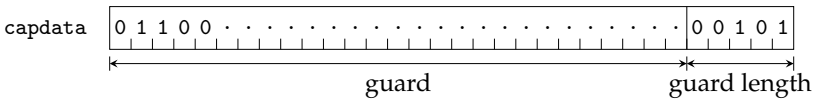
(Diagram credit: seL4 documentation)

Representing CNodes

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

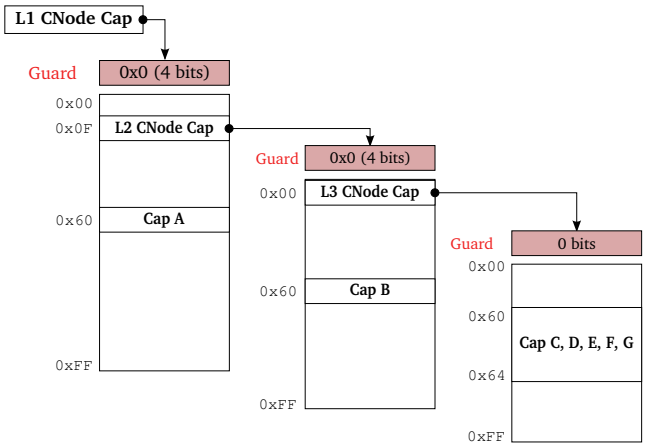


- The `capdata` for a `CNode` specifies a guard



- (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)

(Diagram credit: seL4 documentation)

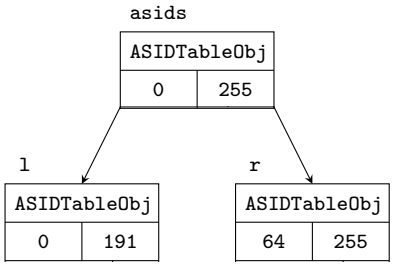
Performance critical?

- 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!

Derived Capabilities

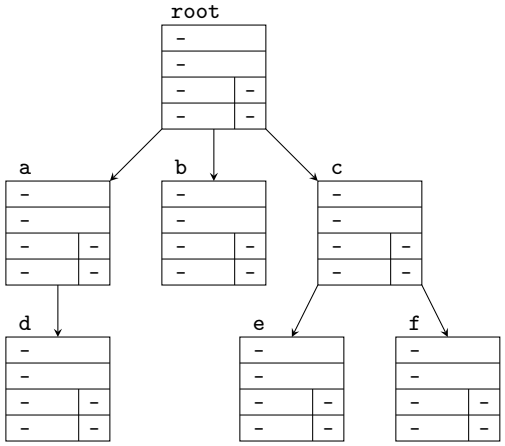
Derived capabilities

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



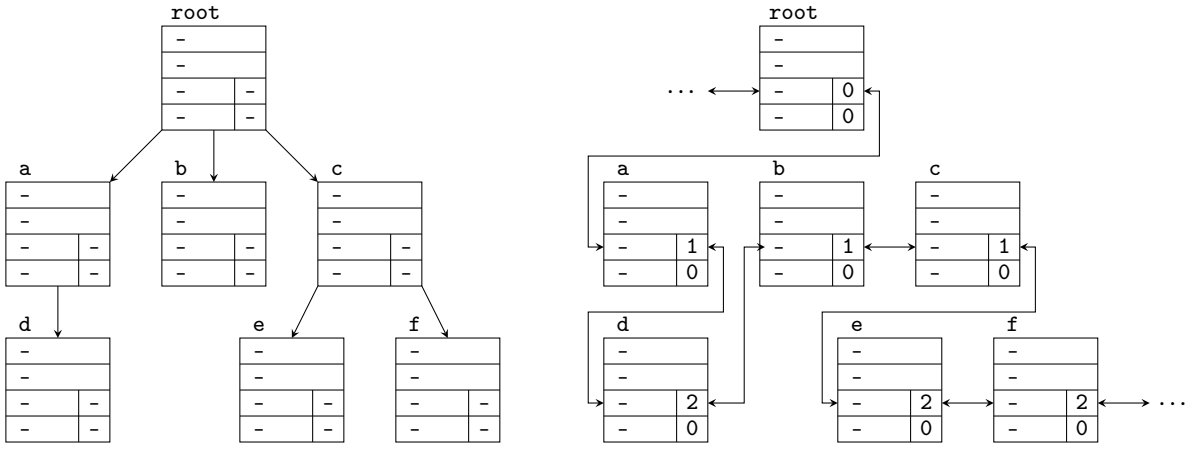
- Another example: root task creates a new endpoint and then hands out two copies of that capability to child threads, one with write permission and one with read permission, to implement a form of "pipe"
- The resulting structure is called the capability derivation tree or CDT

Representing the capability derivation tree



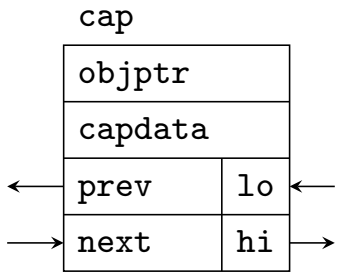
- CDT nodes can have arbitrarily many children
- A conventional implementation would require:
  - unbounded storage per node
  - unbounded recursion (stack) to traverse all children

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

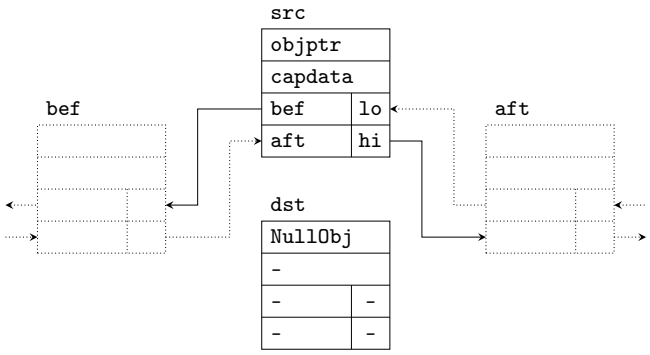
General form



- 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

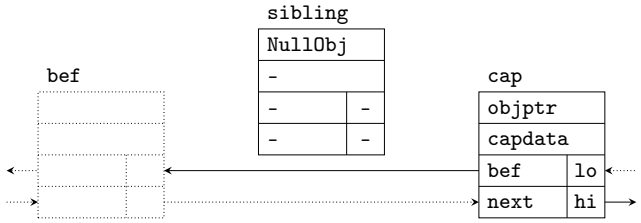
Moving capabilities

Before



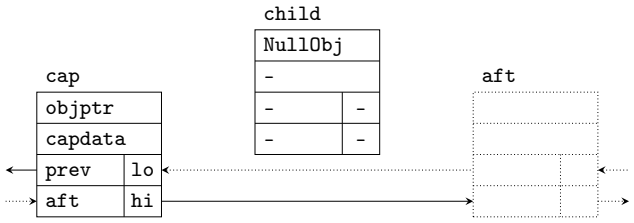
Inserting a sibling

Before

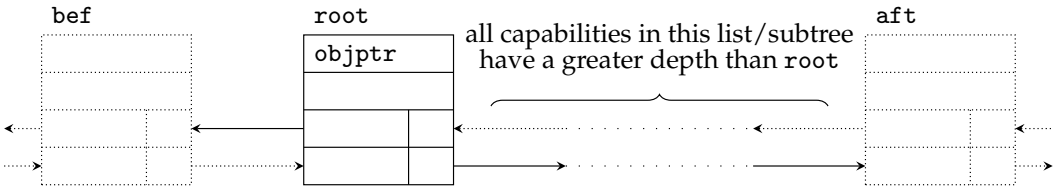


Inserting a child

Before



Visiting a subtree

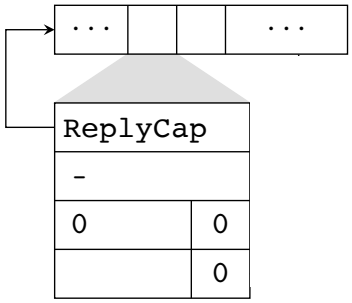


- Pattern for traversing the descendants of a capability:

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

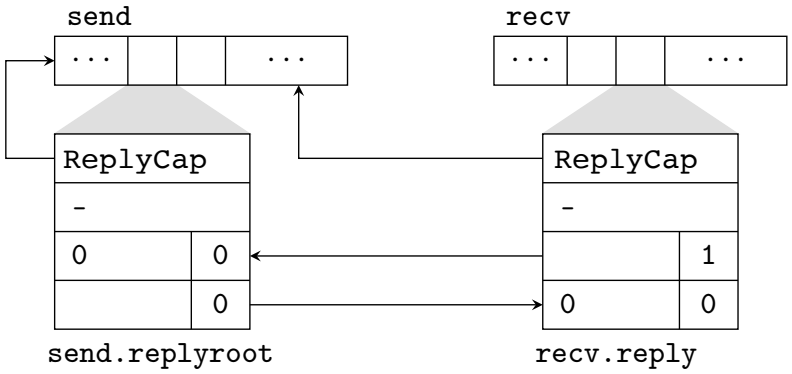
- Typical uses: revoking or deleting a capability

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

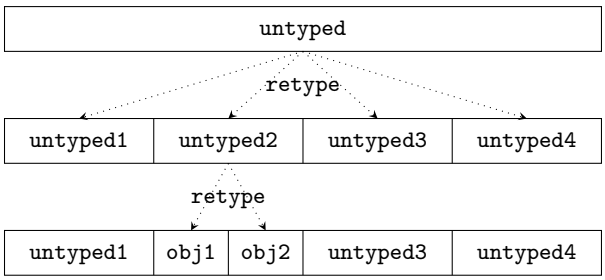
Application: implementing reply capabilities

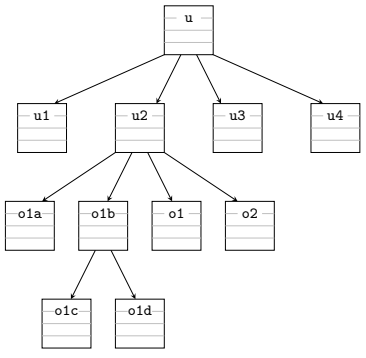


- If one thread makes a "Call" to another, the kernel will insert a child of the sender's master capability in receiver's reply slot
- The receiver can use a "Reply" system call to send a message back to the sender, without knowing its identity
- The kernel can revoke the master reply capability, to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

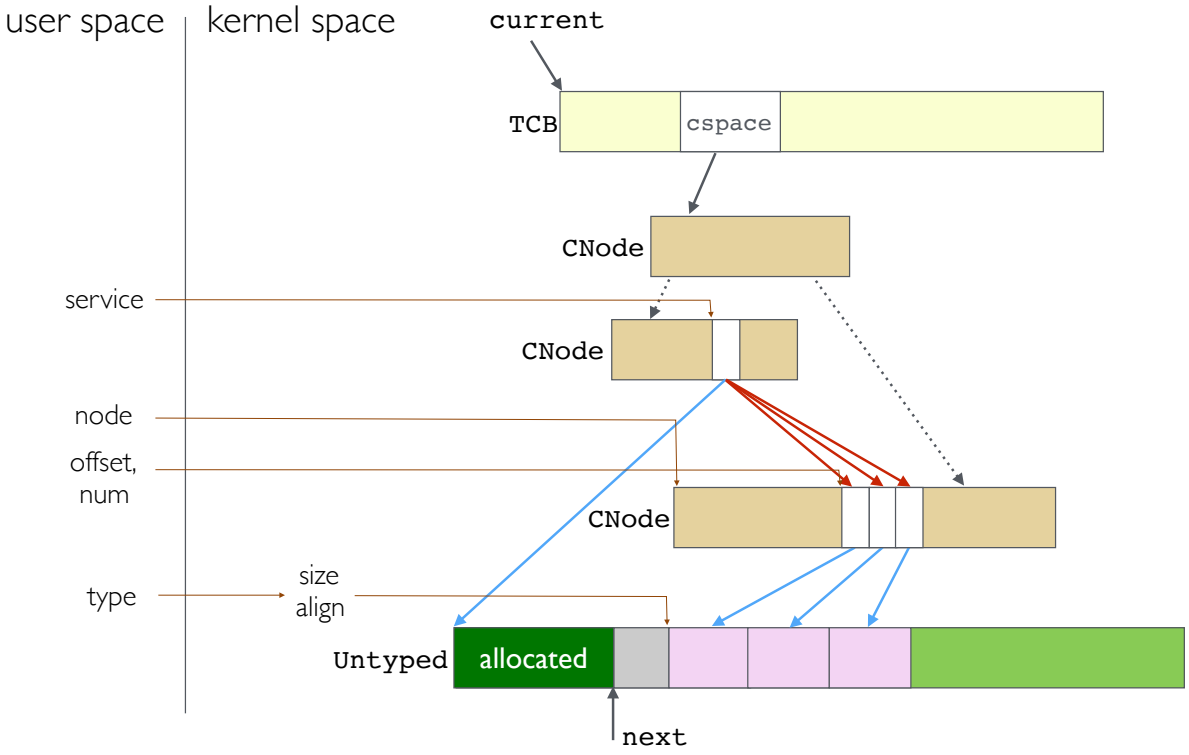
Capability to untyped memory

Type of object to create

CNode where new capabilities should be stored

Window in CNode where new capabilities should be stored

Retype, in pictures



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