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 µs IPC

Second generation
- Eg L4 (’95)
- ~7 syscalls
- ~10 KLOC
- ~1 µs IPC

Third generation
- seL4 (’09)
- ~3 syscalls
- 9 KLOC
- < 1 µs IPC
**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 GMD assembler kernel (’95)
  - Fiasco (Dresden ’98), Hazelnut (Karlsruhe ’99), Pistachio (Karlsruhe/UNSW ’02), L4-embedded (NICTA ’04)
  - L4-embedded commercialised as OKL4 by Open Kernel Labs
  - Deployed in ~ 1.5 billion phones
  - Commercial clones (PikeOS, P4, CodeZero, …)
  - Approach adopted e.g. in QNX (’82) and Green Hills Integrity (’90s)

**Issues of 2G L4 Kernels**

- L4 solved performance issue [Härtig et al, SOSP’97]
- Left a number of security issues unsolved
- Problem: ad-hoc approach to protection and resource management
  - Global thread name space ⇒ covert channels
  - Threads as IPC targets ⇒ insufficient encapsulation
  - Single kernel memory pool ⇒ DoS attacks
  - Insufficient delegation of authority ⇒ limited flexibility, performance
- Addressed by seL4
  - Designed to support safety- and security-critical systems

**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**

- 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
- Capabilities (Caps)
  - mediate access
- System calls
  - Send, Wait (and variants)
  - Yield
Capabilities (Caps)

- Token representing privileges [Dennis & Van Horn, ’66]
  - Cap = “prima facie evidence of right to perform operation(s)”

- Object-specific ⇒ fine-grained access control
  - Cap identifies object ⇒ is an (opaque) object name
  - Leads to object-oriented API:
    ```c
    err = method( cap, args );
    ```
  - Privilege check at invocation time

- Caps were used in microkernels before
  - KeyKOS (’85), Mach (’87)
  - EROS (’99): first well-performing cap system
  - OKL4 V2.1 (’08): first cap-based L4 kernel

seL4 Capabilities

- Stored in cap space (CSpace)
  - Kernel object made up of CNodes
  - each a set 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) [Yes, there should be Execute!]

- 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
    - Only affects object if last cap is deleted
  - **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
    - One partner may have to block
    - Single copy user ⇒ user by kernel

- Two endpoint types:
  - Synchronous (Endpoint) and asynchronous (AsyncEP)

Synchronous Endpoint

- Threads must rendez-vous for message transfer
  - One side blocks until the other is ready
- 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

- Avoids blocking
  - send transmits 1-word message, OR-ed to receiver 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)

Receiving from Sync and Async Endpoints

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
  - Mechanism:
    - AsyncEP is bound to thread with BindAEP() syscall
    - thread waits on synchronous endpoint
    - async message delivered as if been waiting on AsyncEP

Client-Server Communication

- Asymmetric relationship:
  - Server widely accessible, clients not
  - How can server reply back to client?
- Client can pass (session) reply cap in first request
  - server needs to maintain session state
  - client must trust server not to use cap beyond session
- 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

Call RPC Semantics

Client

Kernel

Server

Call(ep,...)

mint rep
deliver to server

Wait(ep,&rep)

process

send rep
deliver to client

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
  - 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 fixed memory location
- 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 message register is special, contains message tag
- Data word for asynchronous IPC
  - Accumulates async messages (reset by Wait)
  - As with interrupts, information is lost if not collected timely
- Reply cap
  - Overwritten by next receive
  - Can move to CSpace with SaveCaller()

---

### IPC Message Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Raw data</td>
</tr>
<tr>
<td>Message</td>
<td>Caps (Send)</td>
</tr>
<tr>
<td>Caps</td>
<td># Caps</td>
</tr>
<tr>
<td>Labels</td>
<td>Message</td>
</tr>
<tr>
<td>CSpace</td>
<td>Length</td>
</tr>
<tr>
<td>Bitmap</td>
<td>Meaning</td>
</tr>
<tr>
<td>Indicating</td>
<td>Defined by</td>
</tr>
<tr>
<td>Caps</td>
<td>IPC protocol</td>
</tr>
</tbody>
</table>

**Note:** Details hidden behind library wrappers
Server Saving Reply Cap

Server

om_endpoint_t ep = om_new_endpoint();
seL4_CPtr ep_cap = om_map_endpoint(adr_space, ep, seL4_all_rights,
seL4_CapMap_MakeBadge(0));

seL4_Word badge;
seL4_MessageInfo msg = seL4_Wait(ep, &badge);
seL4_CPtr slot = om_new_cslot(adr_space);
om_save_reply_cap(slot);

...

seL4_MessageInfo reply = { .raw=0 };
seL4_Send(slot, reply);
om_free_cslot(slot);

Save reply cap in CSpace

Explicit use of reply cap.

Reply cap no longer valid

Derived Capabilities

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

IPC Operations Summary

- **Send (ep_cap, …), Wait (ep_cap, …), Wait (aep_cap, …)**
  - blocking message passing
  - needs Write, Read permission, respectively
- **NBSend (ep_cap, …)**
  - discard message if receiver isn’t ready
- **Call (ep_cap, …)**
  - equivalent to Send (ep_cap, …) + reply-cap + Wait (ep_cap, …)
- **Reply (…)**
  - equivalent to Send (rep_cap, …)
- **ReplyWait (ep_cap, …)**
  - equivalent to Reply (…) + Wait (ep_cap, …)
  - purely for efficiency of server operation
- **Notify (aep_cap, …), Poll (aep_cap, …)**
  - non-blocking send / check for message on AsyncEP
  - No failure notification where this reveals info on other entities!

seL4 System Calls

- **Notationally, seL4 has 8 syscalls:**
  - **Yield():** invokes scheduler
    - only syscall which doesn’t require a cap!
  - **Send(), Receive() and 5 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 Send()/Receive() on an object cap
    - Each object has a set of kernel protocols
      - operations encoded in message tag
      - parameters passed in message words
    - Mostly hidden behind “syscall” wrappers
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 most of this behind the object manager (OM) server API

seL4 Memory Management Approach

- Resources fully delegated, allows autonomous operation
- Strong isolation, No shared kernel resources

Global Resource Manager

RM

Addr Space

Addr Space

Addr Space

Resource Manager

Resource Manager

Global Resource Manager

 RAM

 Kernel Data

 GRM Data

seL4 Address Spaces (VSpaces)

- Very thin wrapper around hardware page tables
  - Architecture-dependent
  - ARM and 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
  - To use it must be associated with “ASID pool”
    - hidden by OM
  - Deleting the PD deletes the VSpace

Memory Management Mechanics: Retype

Retype (Untyped, 2^1)

Retype (Frame, 2^2)

Retype (CNode, 2^m, 2^n)

Retype (TCB, 2^n)

Mint(r)

Revoke()
Address Space Operations

```c
om_frame_t new_frame = om_new_frame();
om_map_frame(adr_space, new_frame, 0xA0000000, seL4_AllRights);
bzero((void *)0x80000000, PAGESIZE);
om_unmap_frame(adr_space, new_frame, 0xA0000000);
om_free_frame(new_frame);
```

Memory Management Caveats

- The object manager handles allocation for you
- However, it is very simplistic, you need to understand how it works
- Simple rule:
  - 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 \( 2n \)

<table>
<thead>
<tr>
<th>Object</th>
<th>size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>( 2^{12} )</td>
</tr>
<tr>
<td>Page directory</td>
<td>( 2^{14} )</td>
</tr>
<tr>
<td>Endpoint</td>
<td>( 2^{4} )</td>
</tr>
<tr>
<td>Cslot</td>
<td>( 2^{4} )</td>
</tr>
<tr>
<td>TCB</td>
<td>( 2^{9} )</td>
</tr>
<tr>
<td>Page table</td>
<td>( 2^{10} )</td>
</tr>
</tbody>
</table>

- All kernel objects must be size aligned!

Memory Management Caveats

- Objects are allocated by `Retype()` of Untyped memory
- Free 4 frames for making a page directory may not work
  - only if they are part of the same Untyped object
  - and they are the full Untyped object

- Be careful with allocations!
- Don’t try to allocate all of physical memory as frames, as you need more memory for TCBs, endpoints etc.
- Allocate big objects first
  - eg `om_address_space_t`
- Be aware that page table objects are also being created behind the scenes.

Untyped Memory \( 2^{15} \) B

Memory Management Caveats

- 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)
    - Yes, this is broken! (Will be fixed soon…)
- These must be explicitly managed
  - … but our object manager hides a lot of the tedious stuff
### 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!

---

### Creating a Thread in Own VSpace

```c
static char stack[100];
int thread_fct() {
    while(1);
    return 0;
}
```

```c
om_endpoint_t exept_ep = om_new_endpoint();
seL4_CPtr endpoint_cap = om_map_endpoint(adr_space, exept_ep, seL4_AllRights, seL4_CapData_MakeBadge(0));
char * dite = (char *)dite_lookup(appdite, "test") -> p_base;
unsigned int entry = elf_getEntryPoint(dite);
```

```c
om_tcb_t tcb = om_new_tcb(adr_space, exct_hdlr, badge, 0, ipc_buf, 0XA0000000);
```

```c
om_start_thread(tcb, &stack, &thread_fct, 0);
```

---

### Creating a Thread in a New VSpace

```c
om_endpoint_t except_ep = om_new_endpoint();
seL4_CPtr endpoint_cap = om_map_endpoint(adr_space, except_ep, seL4_AllRights, seL4_CapData_MakeBadge(0));
```

```c
char * dite = (char *) dite_lookup(appdite, "test") -> p_base;
unsigned int entry = elf_getEntryPoint(dite);
```

```c
om_tcb_t tcb = om_new_tcb(adr_space, entry_hdlr, badge, 0, ipc_buf, 0XA0000000);
```

```c
om_start_thread(tcb, &stack, &thread_fct, 0);
```
**seL4 Scheduling**

- 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**

- 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

**Interrupt Management**

- seL4 models IRQs as messages sent to an AsyncEP
  - Interrupt handler has Receive cap on that EP
- 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 AsyncEP to interrupt
- Afterwards:
  - AsyncEP is used to receive interrupt
  - IRQHandler is used to acknowledge interrupt

```c
seL4_IRQHandler interrupt = om_new_interrupt(usb, tcb);
seL4_IRQHandler_ack(interrupt);
```

### Device Drivers

- 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
  - Only have to map the appropriate page in the driver’s VSpace

```c
om_device_frame_t frame = om_get_device_frame(DEVICE_ADDRESS, 0);
om_map_device_frame(adr_space, frame, 0XA0000000);
... *(void *) 0XA0000000 = 5;
```

### Project Platform: NSLU2 (Slug)

- **Xscale CPU**
- **32 MB Memory**
- **IXP420 Network Processor**
- **Serial USB Port**
- **UART**
- **Ethernet**
- **M0 – make user applications print to here**
  - `seL4_DebugPutChar()`
- **M6 – Network File System (NFS)**
- **Timer & other devices**

### Stuff & Gallery

- **to cover**
  - scheduling