Microkernels: Reducing the Trusted Computing Base

- Idea of microkernel:
  - Flexible, minimal platform
  - Mechanisms, not policies
  - OS functionality provided by usermode servers
  - Servers invoked by kernel-provided message-passing mechanism (IPC)
- Goes back to Nucleus [Brinch Hansen'70]

IPC performance is critical!

Monolithic vs Microkernel OS Evolution

**Monolithic OS**
- New features add code kernel
- New policies add code kernel
- Kernel complexity grows

**Microkernel OS**
- Features add usermode code
- Policies replace usermode code
- Kernel complexity is stable

20,000 kSLOC

10 kSLOC
Microkernel Principle: Minimality

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. [SOSP’95]

- Advantages of resulting small kernel:
  - Easy to implement, port?
  - Easier to optimise
  - Hopefully enables a minimal trusted computing base
  - Easier debug, maybe even prove correct?

- Challenges:
  - API design: generality despite small code base
  - Kernel design and implementation for high performance

Microkernel Evolution

First generation  Second generation  Third generation
Mach [‘87], QNX, Chorus  L4 [‘95], PikeOS, Integrity  seL4 [‘09]

- 180 syscalls, 100 kSLOC
- 100 µs IPC

- ~7 syscalls, ~10 kSLOC
- ~1 µs IPC

- ~3 syscalls, ~10 kSLOC
- 0.1 µs IPC

Capabilities
- Design for isolation

L4: 25 Years High Performance Microkernels

- First L4 kernel with capabilities
- iOS secure enclave
- Code Inheritance
- API Inheritance

- seL4
- OKL4 µKernel
- OKL4 Microvisor
- L4-embed
- L4-
- Fiasco
- Fiasco.OC
- L4MIPS
- L4Alpha
- Hazelnut
- Pistachio
- L3 → L4
- L4-embed
- OKL4 Microvisor
- Codezero
- Qualcomm modem chips

- UNSW/NICTA
- GMD/IBM/Karlsruhe
- Dresden
- UN/CTA
- OK Labs
- Commercial Clone
- P4 – PikeOS

© Gernot Heiser 2019 – CC Attribution License
Issues With 2G Microkernels

- L4 solved microkernel performance [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 [Shapiro’03]
  - Threads as IPC targets ⇒ insufficient encapsulation
  - Single kernel memory pool ⇒ DoS attacks
  - No delegation of authority ⇒ limited flexibility, performance issues
  - Unprincipled management of time
- Addressed by seL4
  - Designed to support safety- and security-critical systems
  - Principled time management (MCS branch)

The seL4 Microkernel

Principles

- Single protection mechanism: capabilities
  - Now also for time [Lyons et al, EuroSys’18]
- 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]

Concepts

- Capabilities (Caps)
  - mediate access
- Kernel objects:
  - Threads (thread-control blocks: TCBs)
  - Address spaces (page table objects: PDs, PTs)
  - Endpoints (IPC)
  - Notifications
  - Capability spaces (CNodes)
  - Frames
  - Interrupt objects (architecture specific)
  - Untyped memory
- System calls:
  - Call, Reply&Wait (and one-way variants)
  - Yield
What Are (Object) Capabilities?

Any system call is invoking a capability:

\begin{verbatim}
err = cap.method(args);
\end{verbatim}

Capabilities provide:
- Fine-grained access control
- Reasoning about information flow

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, Execute, Grant (cap transfer)

Capabilities

- 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 (e.g., copied or minted) caps

Cross-Address-Space Invocation (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
seL4 IPC: Cross-Domain Invocation

- seL4 IPC is not:
  - A mechanism for shipping data
  - A synchronisation mechanism

seL4 IPC is:
- A protected procedure call
- A user-controlled context switch

IPC: Endpoints

- Threads must rendez-vous
  - 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!

Endpoints are Message Queues

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

Server Invocation & Return

- 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
  - allows server to reply to client
  - cannot be copied/minted/re-used but can be moved
  - one-shot (automatically destroyed after first use)
**Call Semantics**

1. **Client** calls **Server** with `Call(srv, args)`
2. **Kernel** delivers the request to **Server** and assigns a reply cap.
3. **Server** processes the request with `process ep=ReplyRecv(ep,&args)`
4. **Kernel** delivers the reply to **Client**.
5. **Client** destroys the reply cap with `destroy reply cap`.

**IPC Mechanics: Virtual Registers**

- Like physical registers, virtual registers are thread state.
- Implemented as physical registers or thread-local memory.
- Message registers contain message transferred in IPC.
- Architecture-dependent subset mapped to physical registers.
- 4 on ARM & x64, 2 on ia32.
- 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 Operations Summary**

- `Call(ep_cap, ...)` generates reply cap on-the-fly.
- `ReplyRecv(ep_cap, ...)`, `Send(ep_cap, ...), Recv(ep_cap, ...), Reply(...)` consumes reply cap.
- For initialisation and exception handling.
- Needs Write, Read permission, respectively.
- `NBSend(ep_cap, ...)` polling send, message lost if receiver not ready.

**Stateful Servers: Identifying Clients**

- Server 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).
  - 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 (session) caps to clients.
  - Separate endpoints for opening session, further invocations.
  - Server tags client state with badge.
  - Kernel delivers badge to receiver on invocation of badged caps.

**Error Handling**

- Need error handling protocol!
Notifications

- Logically, a Notification is an array of binary semaphores
  - Multiple signalling, select-like wait
  - Not a message-passing IPC operation!

- Implemented by *data word* in Notification
  - 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

```
Thread 1       Thread 2
Running   Blocked    Blocked   Running

w = Poll (not_cap, ...)

...... w = Wait (not_cap,...)
```

Receiving from EP and Notification

- 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 events
  - Notification is bound to thread with TCB_BindNotification()
  - thread waits on Endpoint
  - Notification delivered as if caller had been waiting on Notification