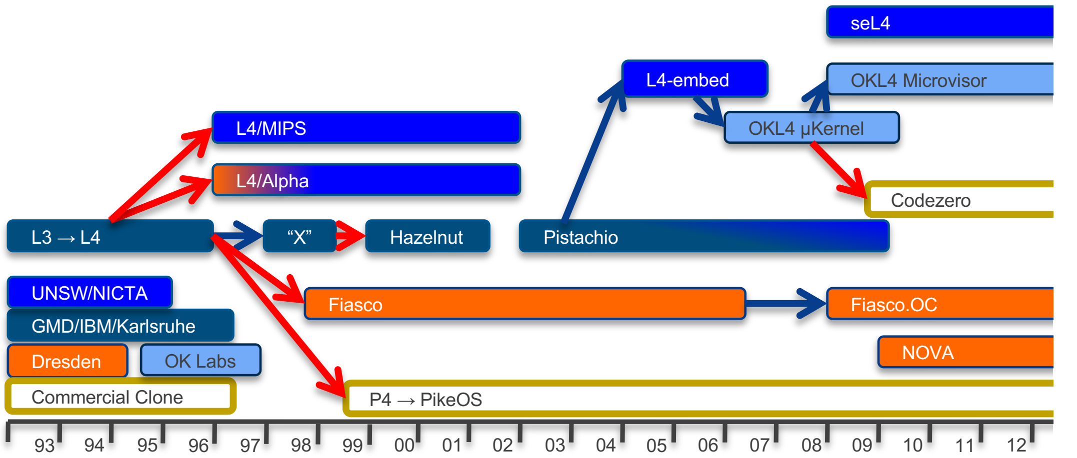
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2020 T2 Week 01a

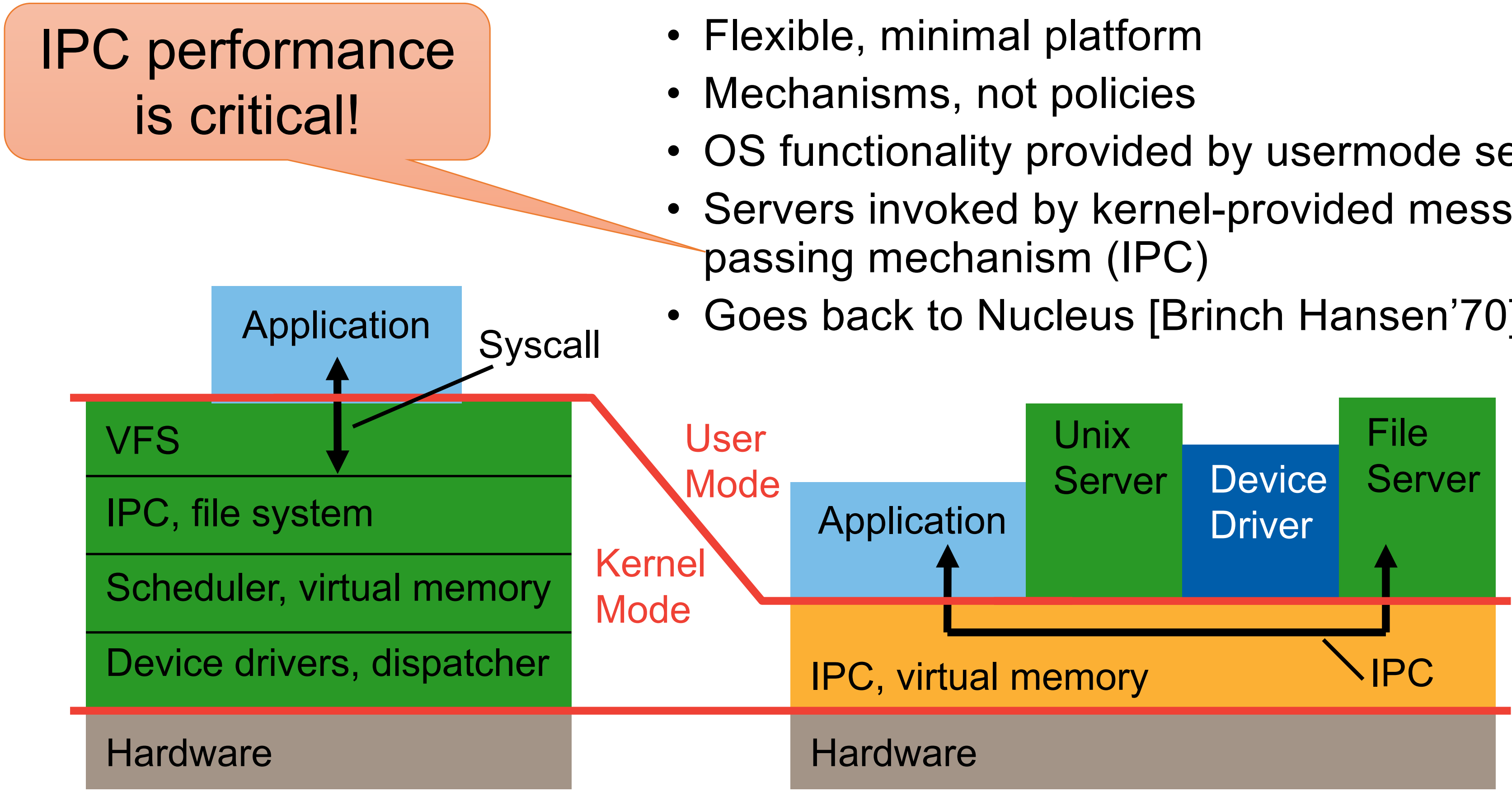
Introduction: Microkernels and seL4

@GernotHeiser



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]



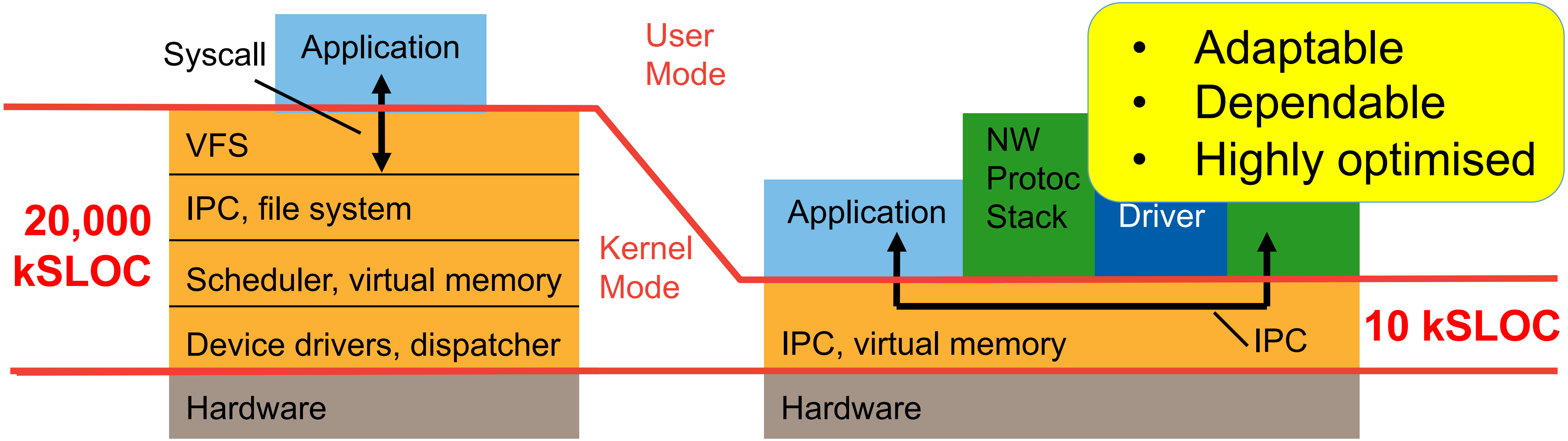
Monolithic vs Microkernel OS Evolution

Monolithic OS

- New features add code kernel
- New policies add code kernel
- Kernel complexity grows

Microkernel OS

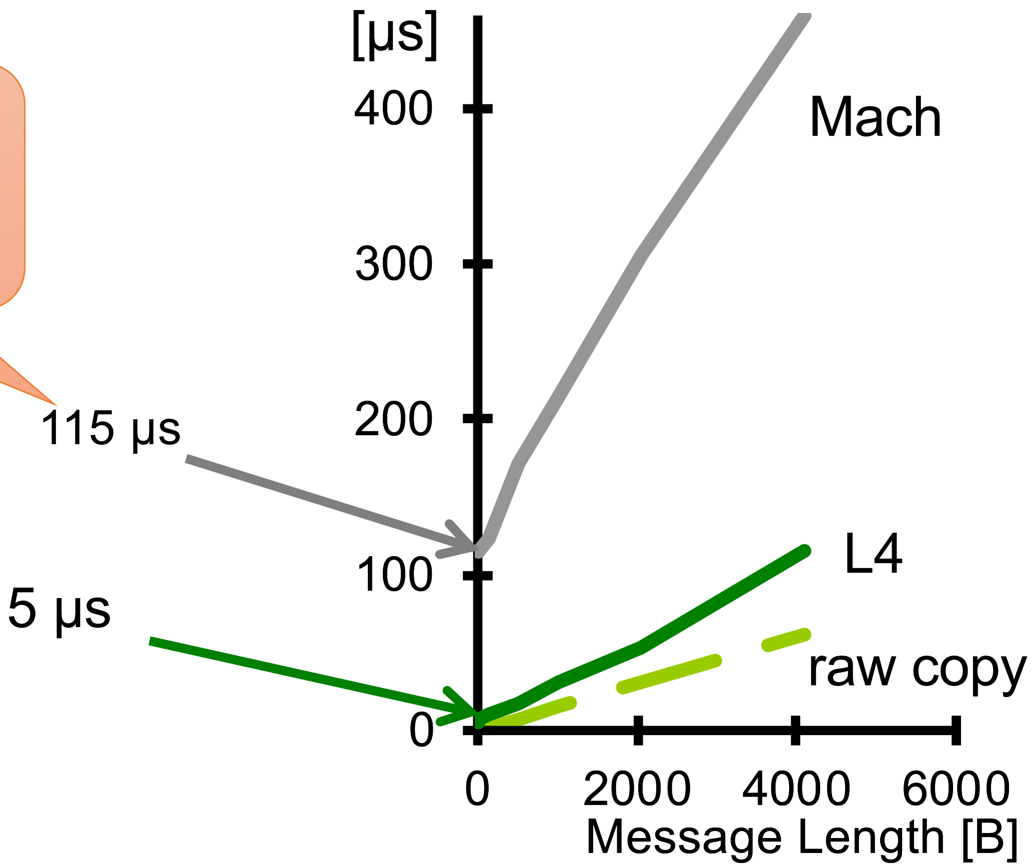
- Features add usermode code
- Policies replace usermode code
- Kernel complexity is stable



- Adaptable
- Dependable
- Highly optimised

1993 “Microkernel”: IPC Performance

Culprit:
Cache footprint
[Liedtke’95]



i486 @
50 MHz

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

Limited by arch-specific micro-optimisations

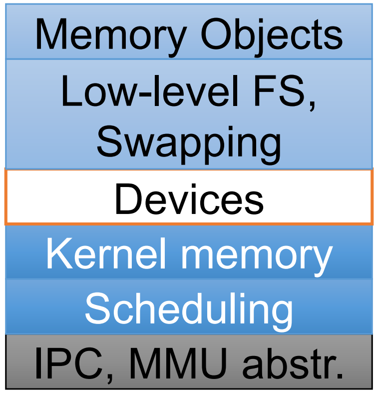
Small attack surface, fewer failure modes

UNSW SYDNEY

Microkernel Evolution

First generation

Mach ['87], QNX, Chorus

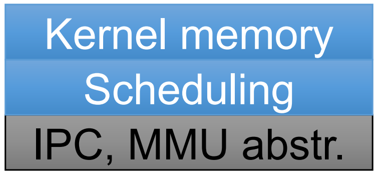


180 syscalls, 100 kSLOC

100 μs IPC

Second generation

L4 ['95], PikeOS, Integrity

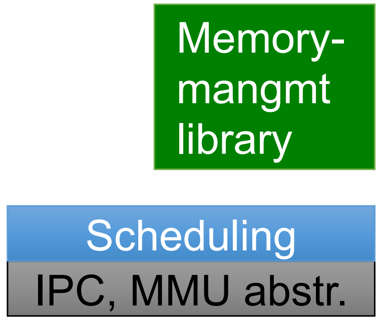


~7 syscalls, ~10 kSLOC

~ 1 μs IPC

Third generation

seL4 ['09]



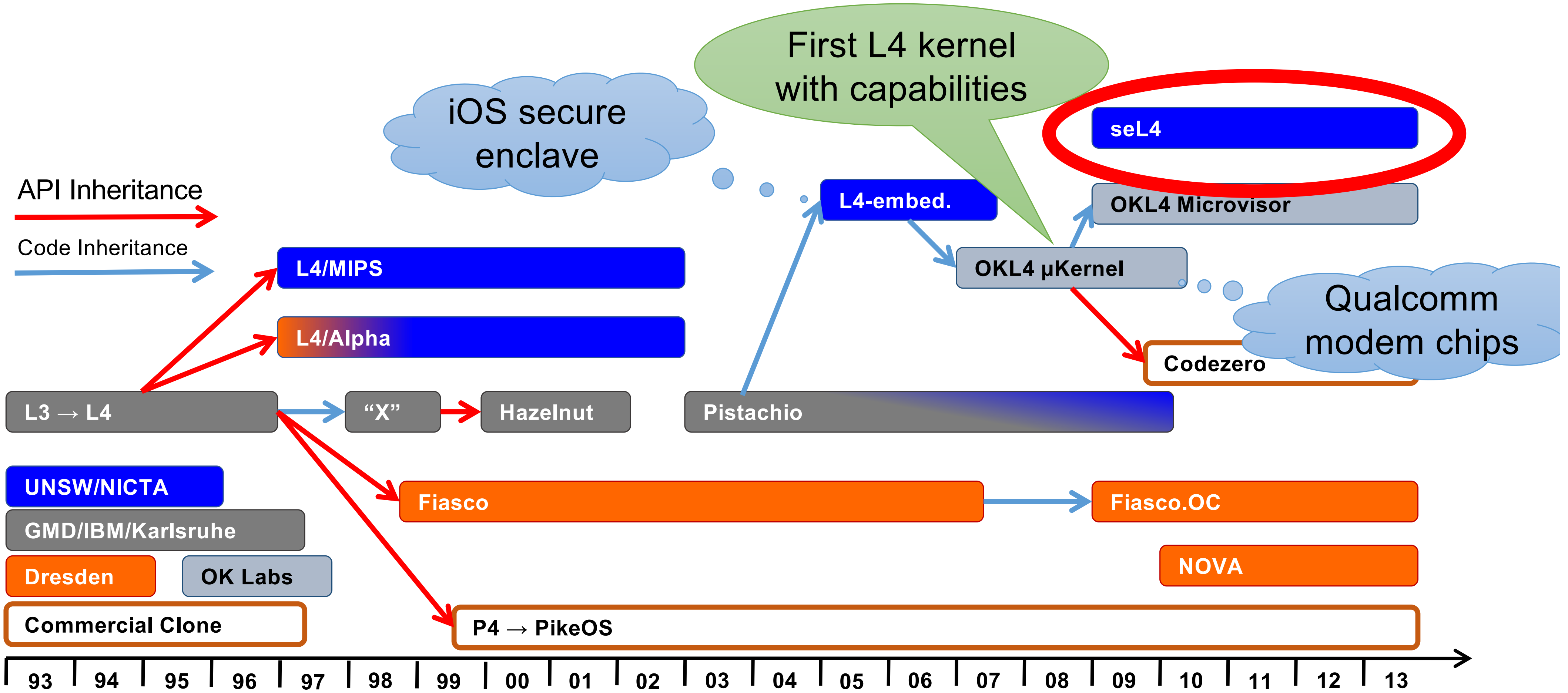
~3 syscalls, ~10 kSLOC

0.1 μs IPC

Capabilities

Design for isolation

L4: 25 Years High Performance Microkernels



Issues With 2G Microkernels

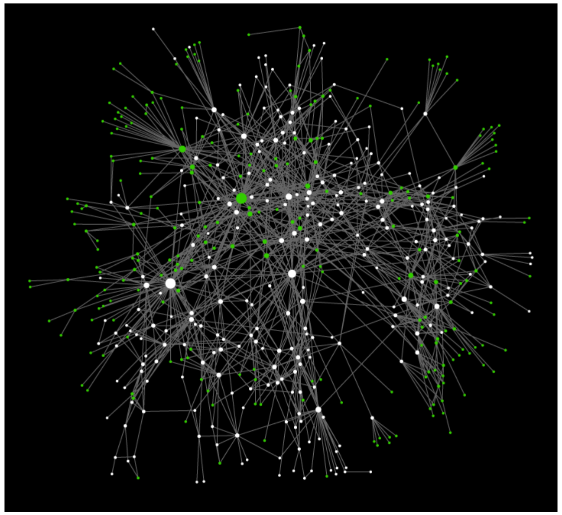
- L4 solved microkernel performance [Härtig et al, SOSP'97]
- Left a number of issues unsolved
- Problem: ad-hoc approach to security and resource management
  - Global thread name space ⇒ covert channels [Shapiro'03]
  - Threads as IPC targets ⇒ insufficient encapsulation
  - Single kernel memory pool ⇒ DoS attacks
  - No delegation of authority ⇒ impacts flexibility, performance
  - Unprincipled management of time

- Addressed by seL4
  - Designed to support safety- and security-critical systems
  - Principled time management (new MCS configuration)

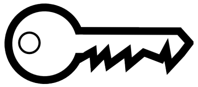
The seL4 Microkernel

seL4 Principles

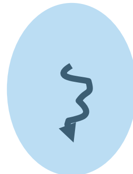
- Single protection mechanism: capabilities
  - Now also for time: MCS configuration [Lyons et al, EuroSys'18]
- All resource-management policy at user level
  - Painful to use
  - Need to provide standard memory-management library
    - Results in L4-like programming model
- Suitable for formal verification
  - Proof of implementation correctness
  - Attempted since '70s
  - Finally achieved by L4.verified project at NICTA [Klein et al, SOSP'09]

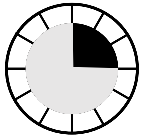


seL4 Concepts in a Slide

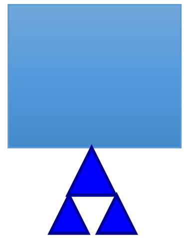


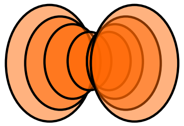
- Capabilities (Caps): reference kernel objects
- 10 kernel object types:
  - Threads (thread-control blocks: TCBs)



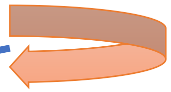


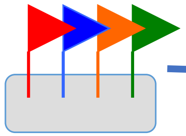
  - Scheduling contexts (SCs)
  - Address spaces (page table objects: PDs, PTs)



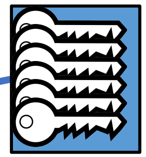


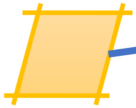
  - Endpoints (IPC)
  - Reply objects (ROs)



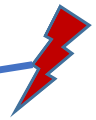


  - Notifications
  - Capability spaces (CNodes)





  - Frames
  - Interrupt objects (architecture specific)



  - Untyped memory

- System calls:
  - `Call()`, `ReplyRecv()` (and one-way variants)
  - `Yield()`

Not a Concept: Hardware Abstraction

Why?

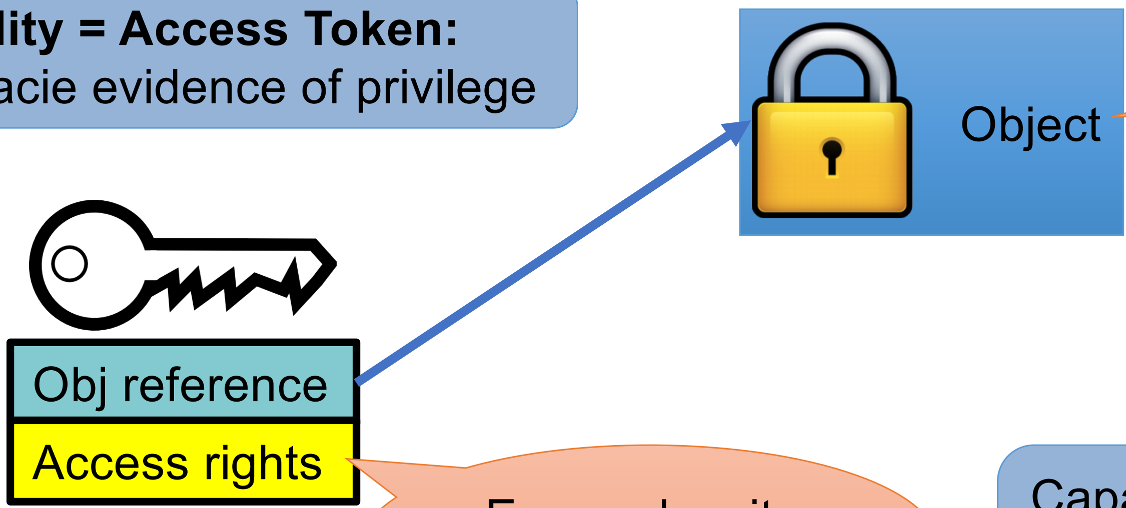
- Hardware abstraction violates minimality
- Hardware abstraction introduces policy

True microkernel:

- Minimal wrapper of hardware, just enough to safely multiplex
- “CPU driver” [Charles Gray]
- Similarities with Exokernels [Engeler ’95]

What Are (Object) Capabilities?

Capability = Access Token:
Prima-facie evidence of privilege



Eg. thread, address space

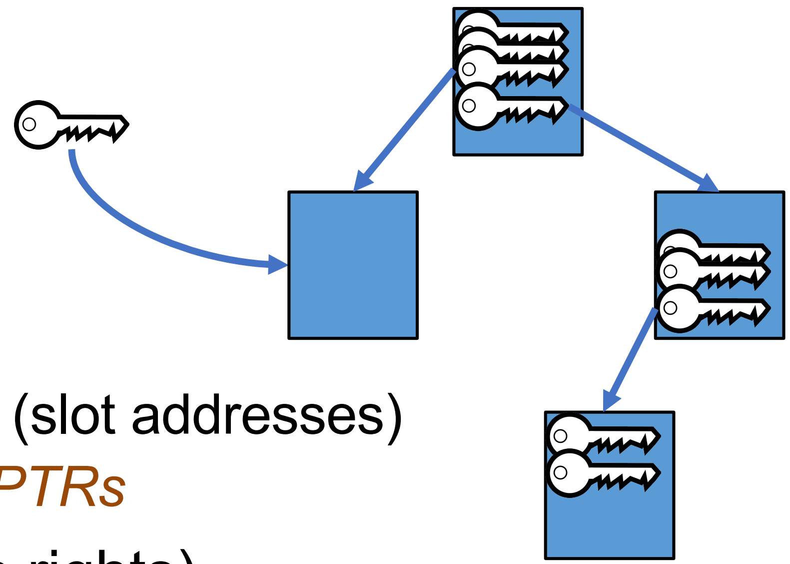
Eg. read, write, send, execute…

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

Any system call is invoking a capability:

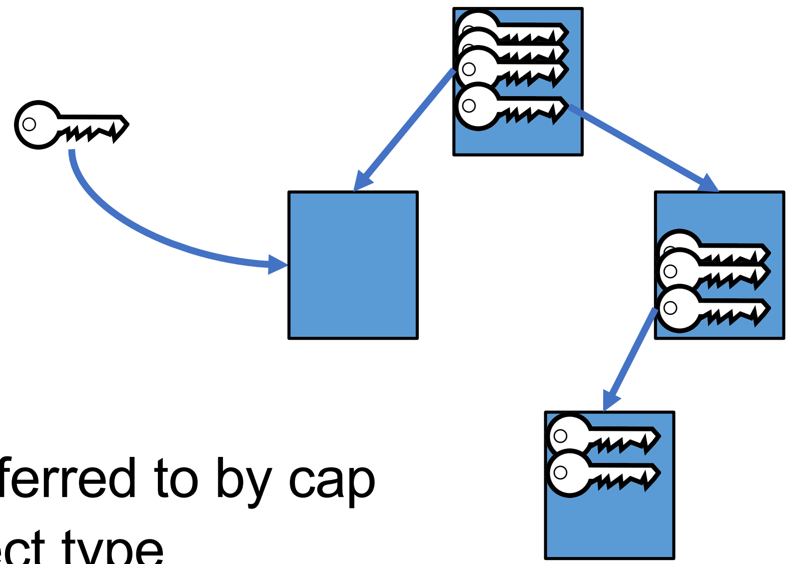
```
err = cap.method( args );
```

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`, `GrantReply` (call), `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 (eg. copied or minted) caps

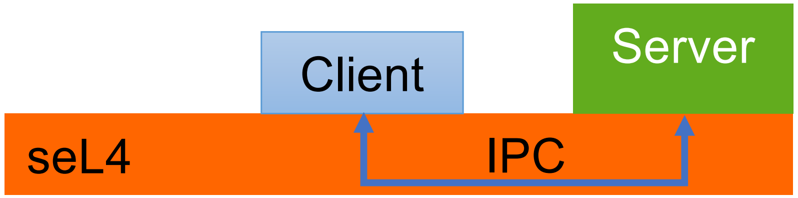
seL4 Mechanisms

IPC & Notifications

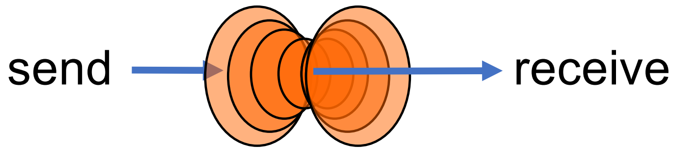
Cross-Address-Space Invocation (IPC)

Fundamental microkernel operation

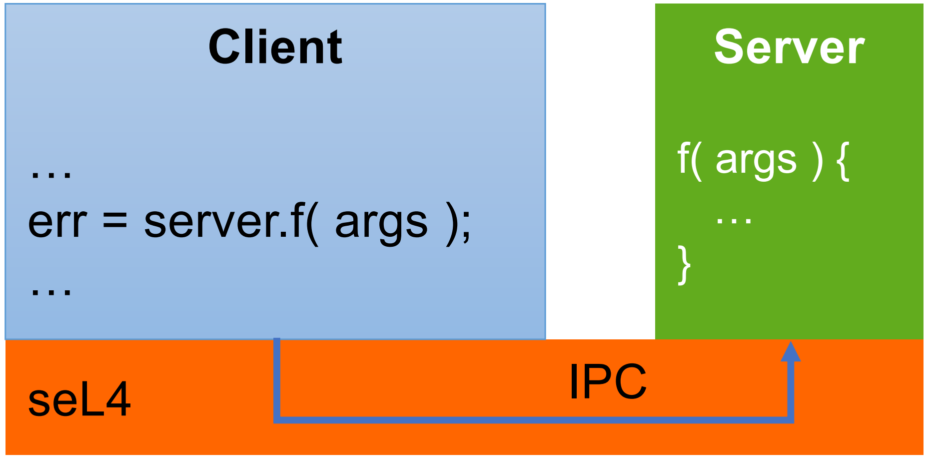
- Kernel provides no services, only mechanisms
- OS services provided by (protected) user-level server processes
- invoked by IPC



- 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**:

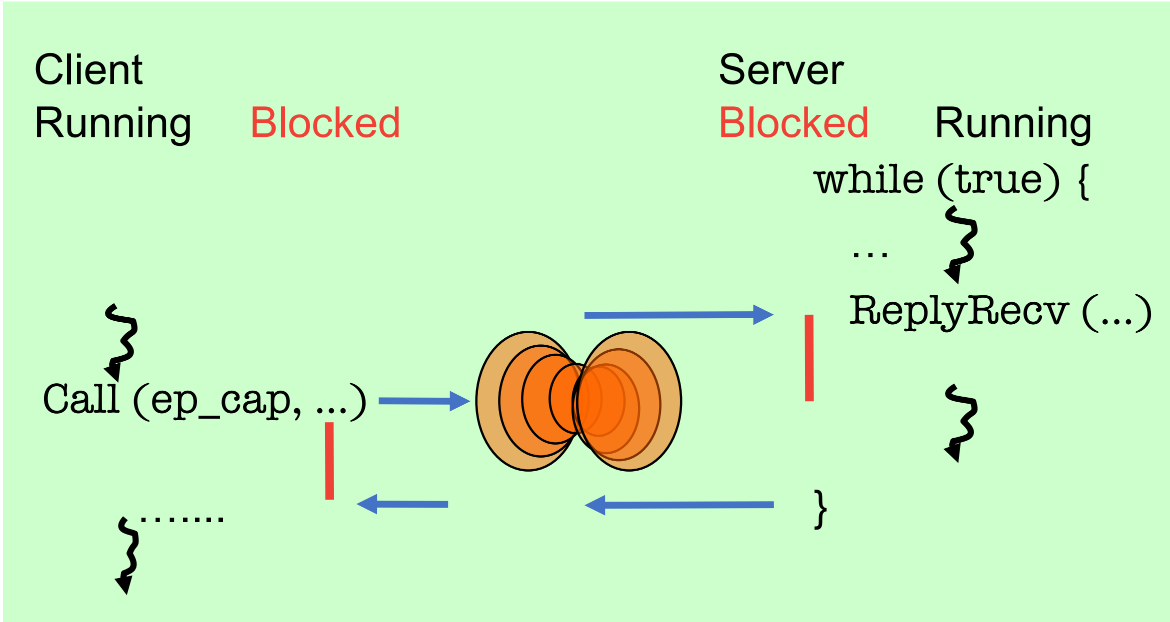
- A mechanism for shipping data
- A synchronisation mechanism
  - side effect, not purpose

seL4 IPC **is**:

- A protected procedure call
- A user-controlled context switch

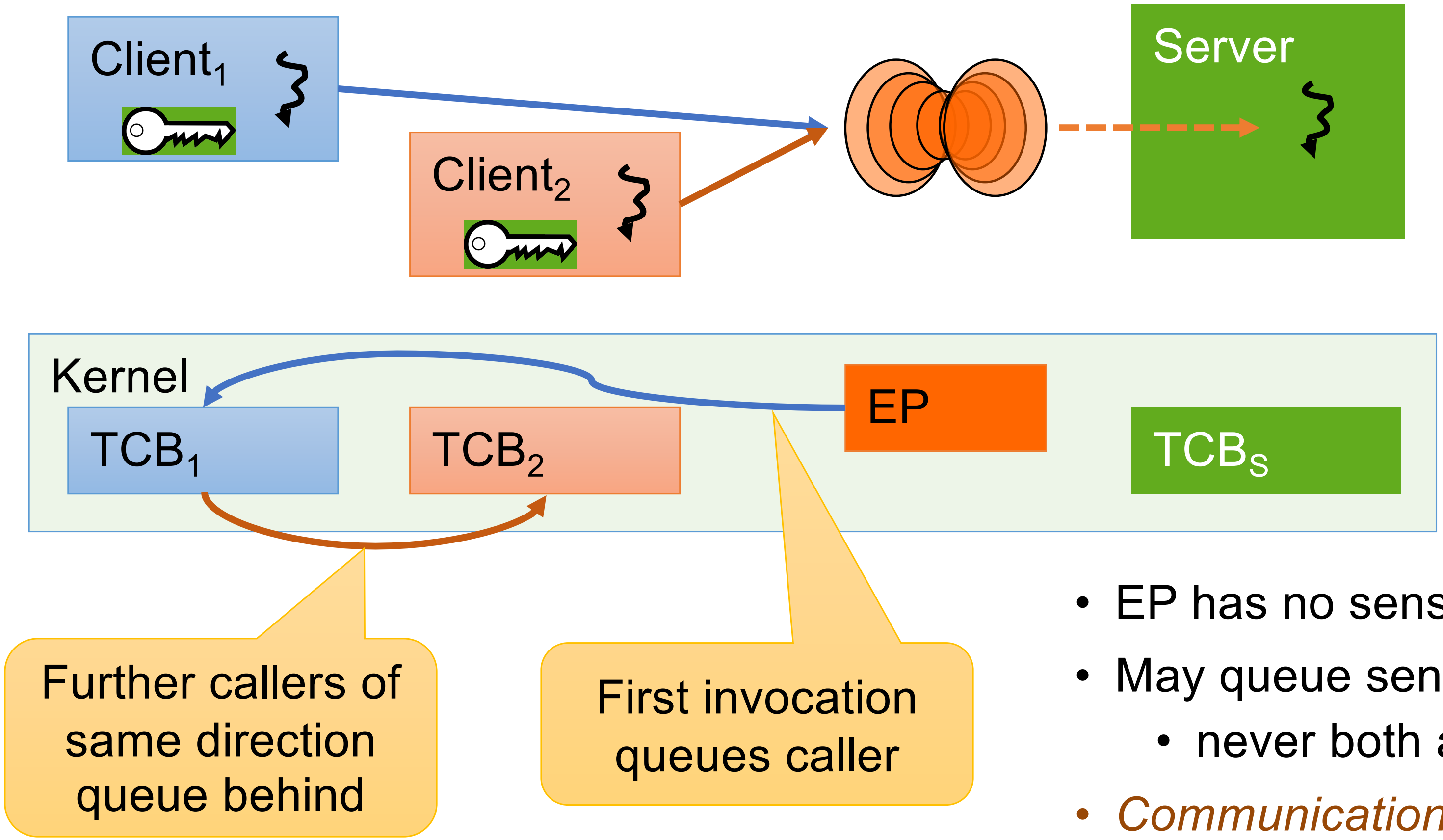
IPC: Endpoints

- Involves 2 threads, but always one blocked
- logically, thread moves between address spaces



- 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

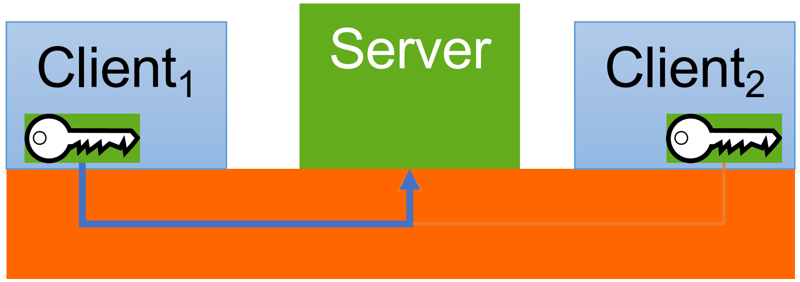


Note: Should not normally get queues on a single core, server should have higher priority than clients!

- 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)?

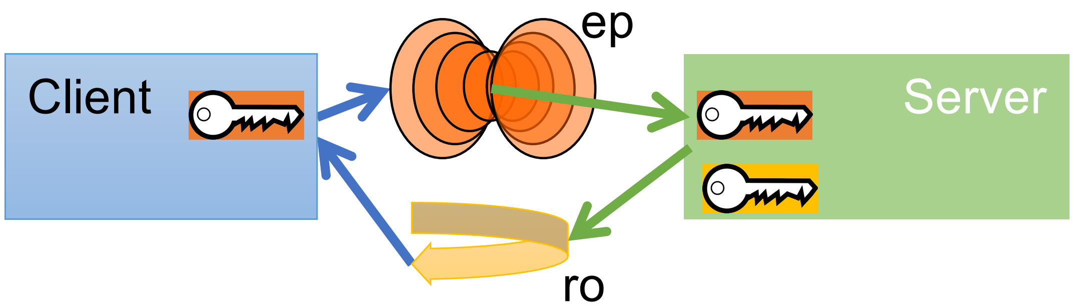


- Client can pass (session) reply cap in first request
  - server needs to maintain session state
  - forces stateful server design

New MCS kernel semantics!

- seL4 solution: Kernel creates channel in *reply object* (RO)
  - server provides RO in `ReplyRecv()` operation
  - kernel connects RO to client when executing receive phase
  - server invokes RO for send phase (only one send until refreshed)
  - only works when client invokes with `Call()`

Call Semantics



Client | **Kernel** | **Server**

ReplyRecv(ro,ep,&args)

Call(ep, args) →

deliver to server

block client on RO →

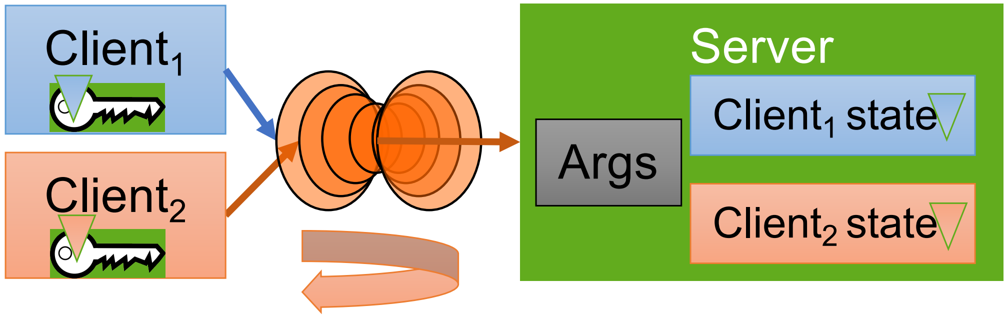
process

← ReplyRecv(ro,ep,&args)

deliver to client

process ←

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)
  - 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 (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

IPC Mechanics: Virtual Registers

- Like physical registers, virtual registers are thread state
  - context-switched by kernel
  - implemented as physical registers or thread-local memory
- Message registers
  - contain message transferred in IPC
  - architecture-dependent subset mapped to physical registers
    - 4 on ARM & x64
    - library interface hides details
    - 1st transferred word is special, contains *message tag*
  - API MR[0] refers to next word (not the tag!)

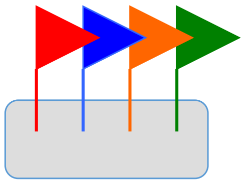
IPC Operations Summary

- `Call (ep_cap, ...)`
  - *Atomic*: guarantees caller is ready to receive reply
  - Sets up server's reply object
- `ReplyRecv (ep_cap, ...)`
  - Invokes RO, waits on EP, re-inits RO
- `Recv (ep_cap, ...)`, `Reply(...)`, `Send (ep_cap, ...)`
  - For initialisation and exception handling
  - needs Write, Read permission, respectively
- `NBSend (ep_cap, ...)`
  - Polling send, message lost if receiver not ready

Not really useful

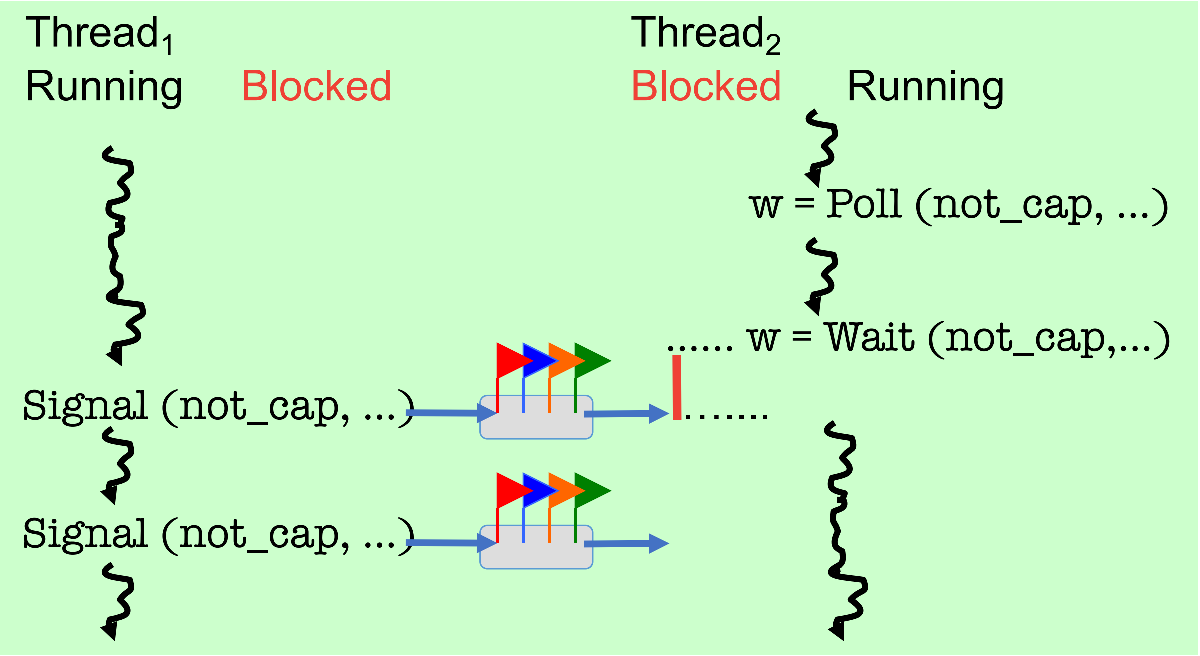
Need error handling protocol !

No failure notification where this reveals info on other entities!



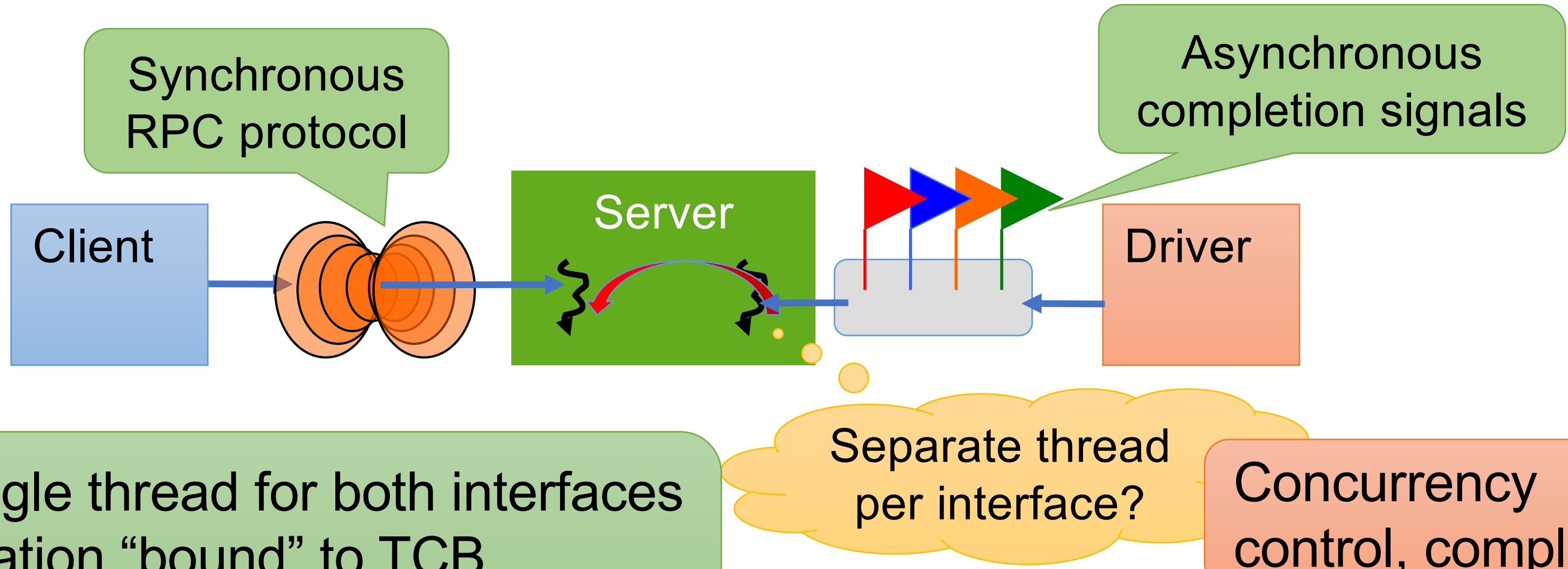
Notifications – Synchronisation Objects

- 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



Receiving from EP *and* Notification

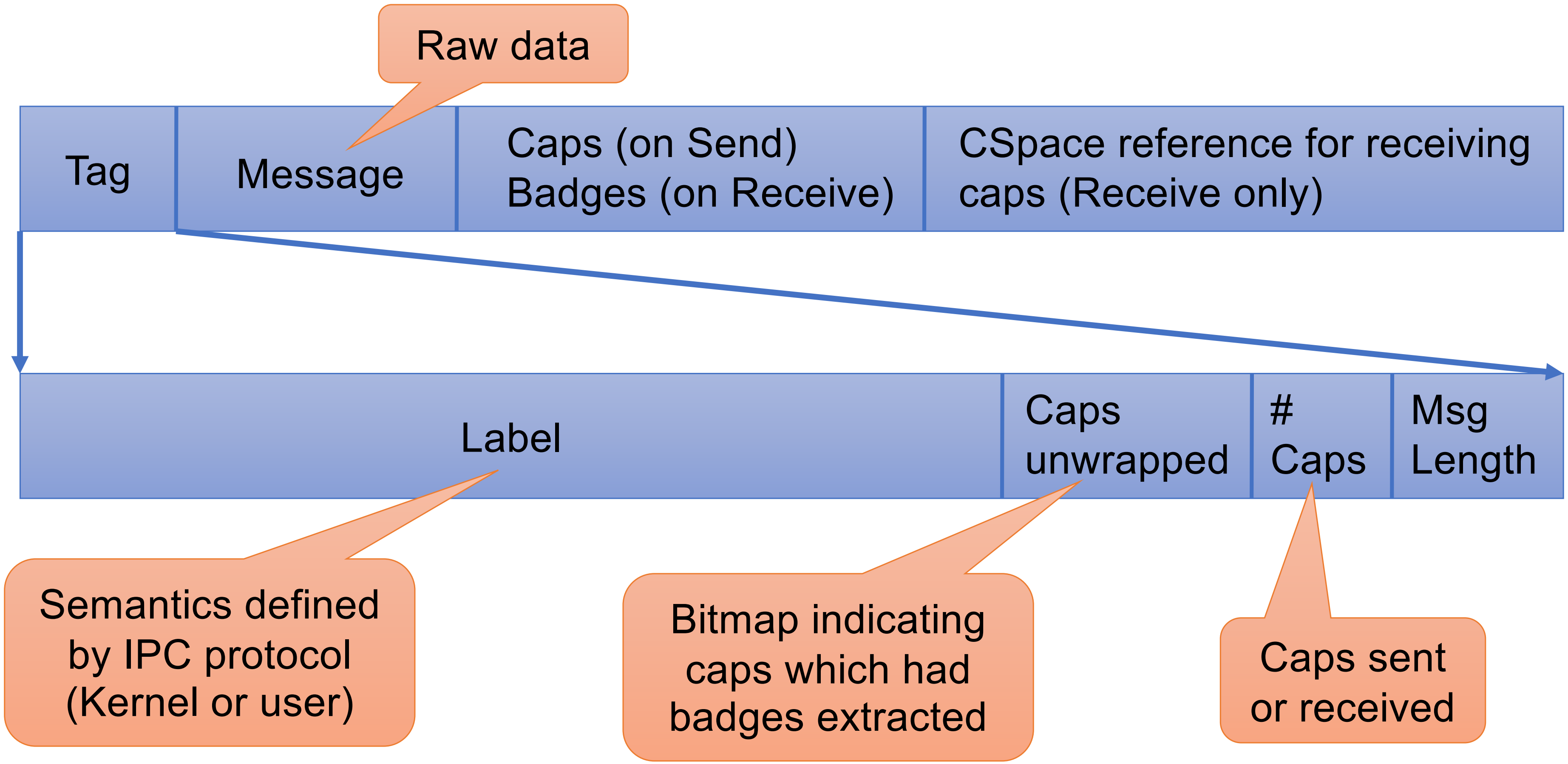
Server with synchronous and asynchronous interface



Concurrency control, complexity!

Better: single thread for both interfaces
- Notification "bound" to TCB
- Signal delivered as "IPC" from EP

IPC Message Format



Client-Server IPC Example

Client

```
seL4_MessageInfo_t tag = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_SetTag(tag);
seL4_SetMR(0,1);
seL4_Call(server_c, tag);
```

Load into tag register

Set message register #0

Server

```
ut_t* reply_ut = ut_alloc(seL4_ReplyBits, &cspace);
seL4_CPtr reply = cspace_alloc_slot(&cspace);
err = cspace_untyped_retype(&cspace, reply_ut->cap, reply,
                            seL4_ReplyObject, seL4_ReplyBits);
seL4_CPtr badged_ep =  cspace_alloc_slot(&cspace);
cspace_mint(&cspace, badged_ep, &cspace, ep, seL4_AllRights, 0xff);
…
seL4_Word badge;
seL4_MessageInfo_t msg = seL4_Recv(ep, &badge, reply);
…
seL4_MessageInfo_t response = seL4_MessageInfo_new(0, 0, 0, 1);
seL4_NBSend(reply,response);
```

Allocate slot & retype to RO

Mint cap with badge 0xff

Wait on EP, receiving badge, setting RO

Reply to sender identified by RO

Should really use ReplyRecv!