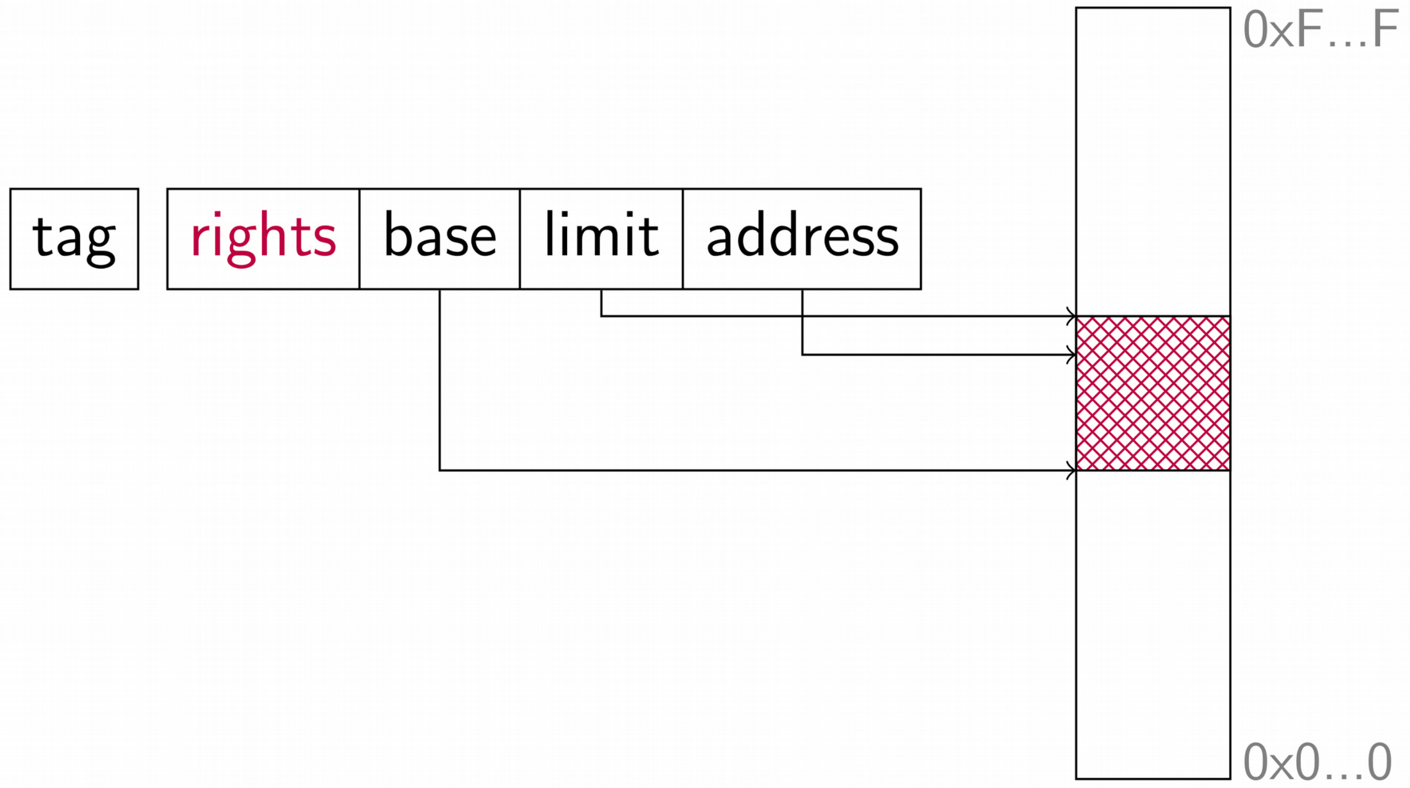
CheriOS

Talk by: Lawrence Esswood

CHERI (briefly)

- Capabilities
  - Identify resources & grants right to use them
  - Unforgeable
- CHERI
  - Hardware enforced capabilities for
    - Memory (split into read/write/execute...)
    - System management
    - Sealing...

A CHERI Capability



A CHERI Capability

- Not just for bounds checking
  - Compartments
  - Sealing

Principles

- CheriOS, designed with failure in mind.
  - Unreasonable to expect people to build systems that are both correct and large

Principles

- Compartmentalise
  - Limit exposure on failure

Principles

- Least Privilege
  - No point splitting roles if each actor has the same privilege

Principles

- Privilege does not imply trust
  - Assume privilege will always be misused

Principles

- Failure should result in only denial of service
  - A system component can
    - Fail to provide its service
    - Provide the service incorrectly
  - A system component cannot
    - Compromise the users confidentiality/integrity
    - Confuse the user into thinking they have been provided a resource they have not

Examples - Scheduler

- Purpose
  - Provide time sharing
- Can
  - Deny scheduling you completely
  - Wake you up from sleeping too early
- Cannot
  - Access your state
  - Change your state when it reschedules you
  - Convince you an event has happened when it hasn’t

Examples – Page Allocator

- Purpose
  - Provide memory resources to users
- Can
  - Fail to provide you memory
  - Take back memory unexpectedly (causing a fault)
- Cannot
  - Read the memory it gives you
  - Change the contents unexpectedly

One Ring to Rule Them All

- Privilege Rings have problems
  - Inherently hierarchical
  - Create high-value targets (get at a given ring, and every outer ring is yours)
  - Large TCBs
    - A user is forced to trust the entire OS
    - e.g. Windows has 45 MLOC!
- Can we build a **flat** system?
- Can we build an **untrusted** system?

Fundamental Problems

- System interfaces are too powerful
  - Exception Vectors
  - Page Tables
- CHERI capability derivation is inherently hierarchical
  - How can a user gain a capability, via the OS, that the OS does not have?
  - How can we stop the user from abusing such a mechanism if it did exist?

The Nanokernel

- Division of system capabilities into larger, less powerful, orthogonal set
- Provide new security primitives
  - Reservations (Memory Integrity & Confidentiality)
  - Foundations (Program Identity & Attestation)
- Really Small!
  - 2625 MIPS **instructions** (about 50 routines)
    - No stack, nearly entirely loop free preemptable leaf functions
  - Compare to microkernel like SEL4, about 10 000 Lines of C

CheriOS

- Microkernel (pre-emptable)
  - Scheduler activations, Message Passing, Interrupts/Exceptions
- All other OS services fully compartmentalised
  - Memory Manager
  - Process Manager
    - WARNING
  - Namespace Service
  - File Systems / Drivers / Type manager …
- All ‘unprivileged’
  - Apart from having nanokernel capabilities a user may not

What can a user expect?

- Spatial Safety
- Temporal Safety
- Compartmentalisation
  - Only explicit argument capabilities can move outside compartment
- Compartment local CFI

The Nanokernel – System Management

- Opaque VCPU contexts
  - Capability to create / destroy
  - Capability to switch
  - Capability to set a context as the exception context
- (Non user) exceptions force context switch
  - Exception context cannot modify state of victim context
- User exceptions can be handled by compartments
  - 210 / 387 (to assembly / to C) cycles to handler and back
- NOT a compartment
  - Nanokernel mostly agnostic to compartments

The Nanokernel – System Management

- Checks Page Table Operations
  - Presently enforces bijective mapping
  - Cannot re-map without either
    - Revoking
    - Proving this would grant no extra power (future extension)
  - Only allows mapping to pages that have been zeroed
  - This makes all memory (by default)
    - Start in a guaranteed state
    - Be non aliasing
    - Seem like one big physical memory, or gives a fault

Reservations

- Capability to *allocate* a range of memory
- Can be copied
- All operations are (somewhat) destructive and apply to all copies
- Software enforced

Reservations - State

- Open
  - Available for allocation (no client has access)
- Taken
  - Has been allocated (allocating client has access)
  - Handle used for revoking
- Merged
  - Revoking delegated to another handle

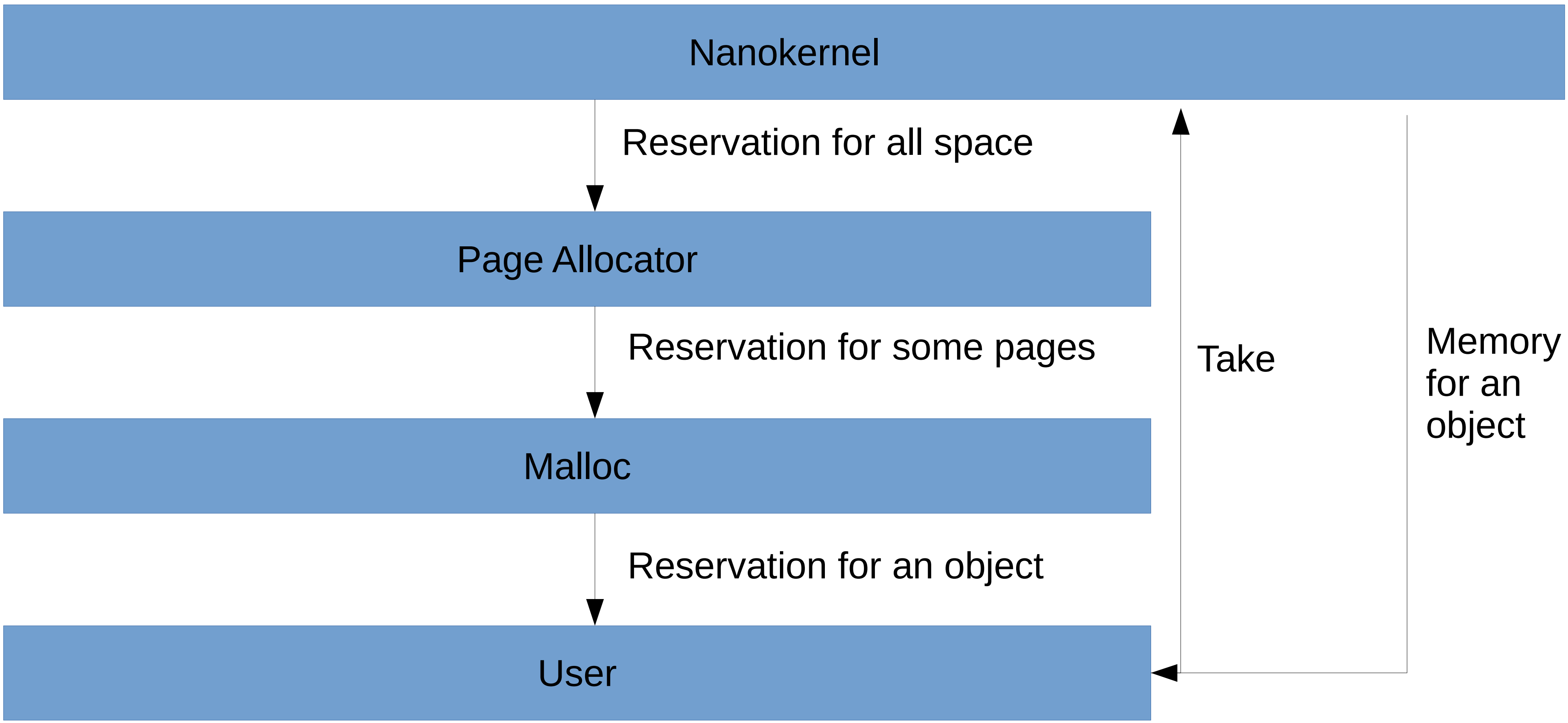
Reservations - Operations

- Take
  - Open → Taken
- Split
  - Open → (Open,Open)
- Merge
  - (Taken, Taken) → (Taken, Merged)
- Revoke
  - Taken → Revoking → Open
  - Destroys all capabilities to region
  - Destroys ALL stale handles (including merged ones)

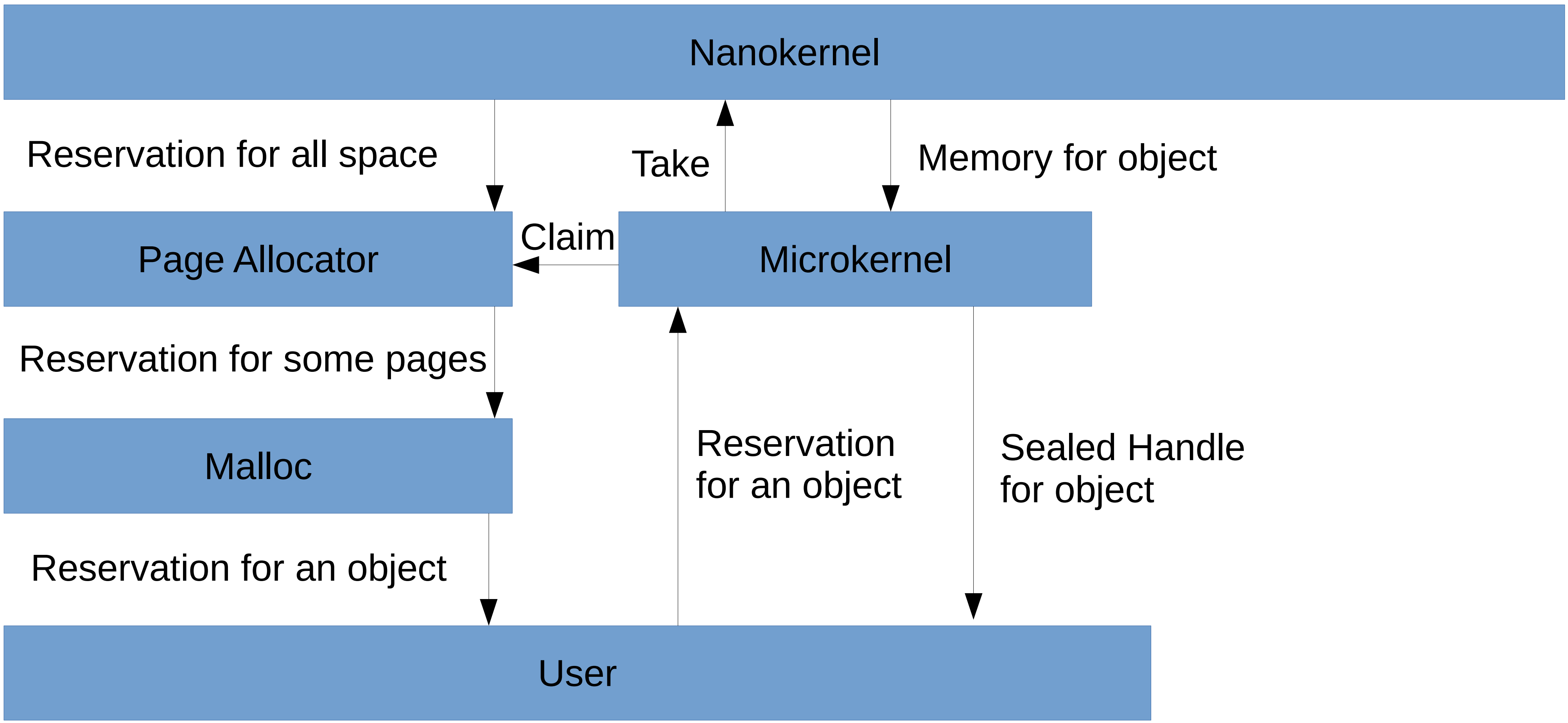
Reservation Guarantees

- When you take a reservation
  - You get a memory capability
  - It is unique and non-aliasing (outside the nanokernel)
  - The nano kernel will not, even in part, grant the same capability again
  - Memory will be in a known starting state (all zero)
- This relies on the rules for virtual memory

Reservations – Typical Allocation



Reservations – More interesting



Spatial Safety

- Get reservations from malloc
  - Guaranteed non aliasing allocation
  - Don’t trust malloc
  - Don’t trust the OS
  - Low overhead of few extra function calls to allocate
  - **No** overhead when being used
- Exploit CHERI bounds checkis
  - All objects have strict bounds checking
  - Including function bodies

Temporal Safety

- Non re-use
  - Allocations cannot alias with old ones
  - Can re-use/pool if performance is critical
- For the stack as well (selectively)!
  - New ‘unsafe’ keyword for C
    - Unsafe will not re-use that stack stack space
  - Automatic compiler assistance for safe/unsafe tagging
  - Less costly than using malloc for every stack frame
  - No cost if keyword is not used and allocations can be proven safe
    - True for majority of frames both statically and dynamically

Compartmentalisation

- Every dynamic library (or even function) can be its own sandbox
  - Elective, potentially uni-directional isolation
- New calling convention
  - Callers can make an ‘untrusting’ call
  - Callees can elect for Callers to be ‘Untrusted’
  - Cost only paid if used
  - Dynamically reconfigurable
  - The microkernel/nanokernel are just dynamic libraries!
    - Syscalls/nanocalls have the same cost as dynamic library calls

CHERI (sealing/ccall)

- Capability to ‘seal/unseal’
  - With a particular type
- Sealed capabilities cannot be used
- CCall
  - Need two capabilities, one code, one data, of the same type
  - Unseals both and atomically jumps to the code
  - Puts data in IDC
    - In Cherios IDC always points to a structure that you should think of as a compartment identifier

Compartment

- Not a VCPU / Scheduler activation
- Identified by a data structure (DLS) used for ccall
  - Per thread
  - Can be used from any VCPU / Scheduler activation
- DLS has entries for (depending on subtype)
  - Subtype
  - Concurrency guard
  - User exception handling
  - Sealing / Unsealing capability
  - Stack
  - Capability to program globals table
  - Capabilities to thread local variables

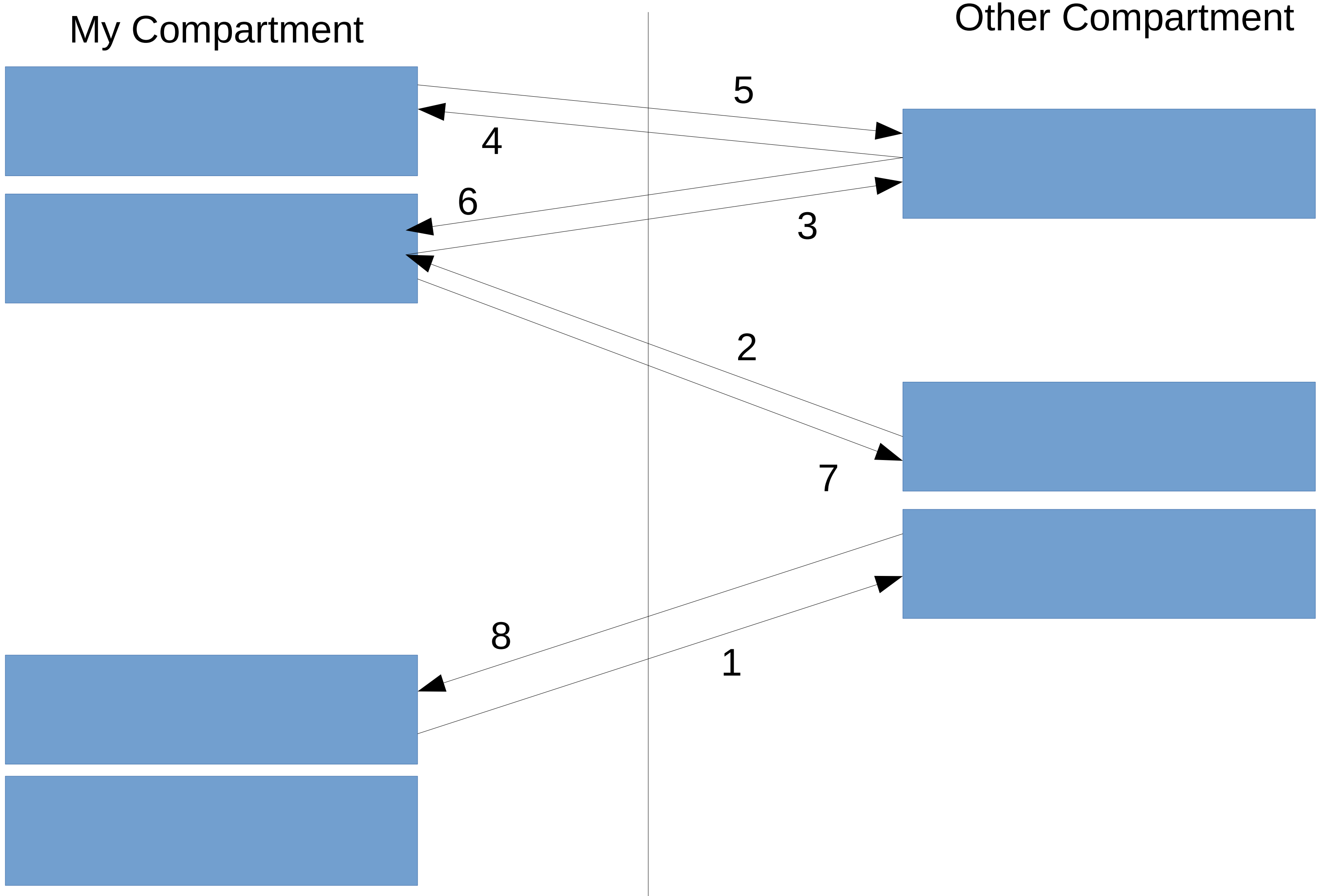
Calling Convention

- Untrusting call (callee untrusted)
  - Save all registers (even callee saves)
  - Make and seal a new DLS
  - Zero registers
  - On return make CFI checks and restore
- Untrusted call (caller untrusted)
  - Make CFI checks on entry
  - On return clear tmps and caller saves

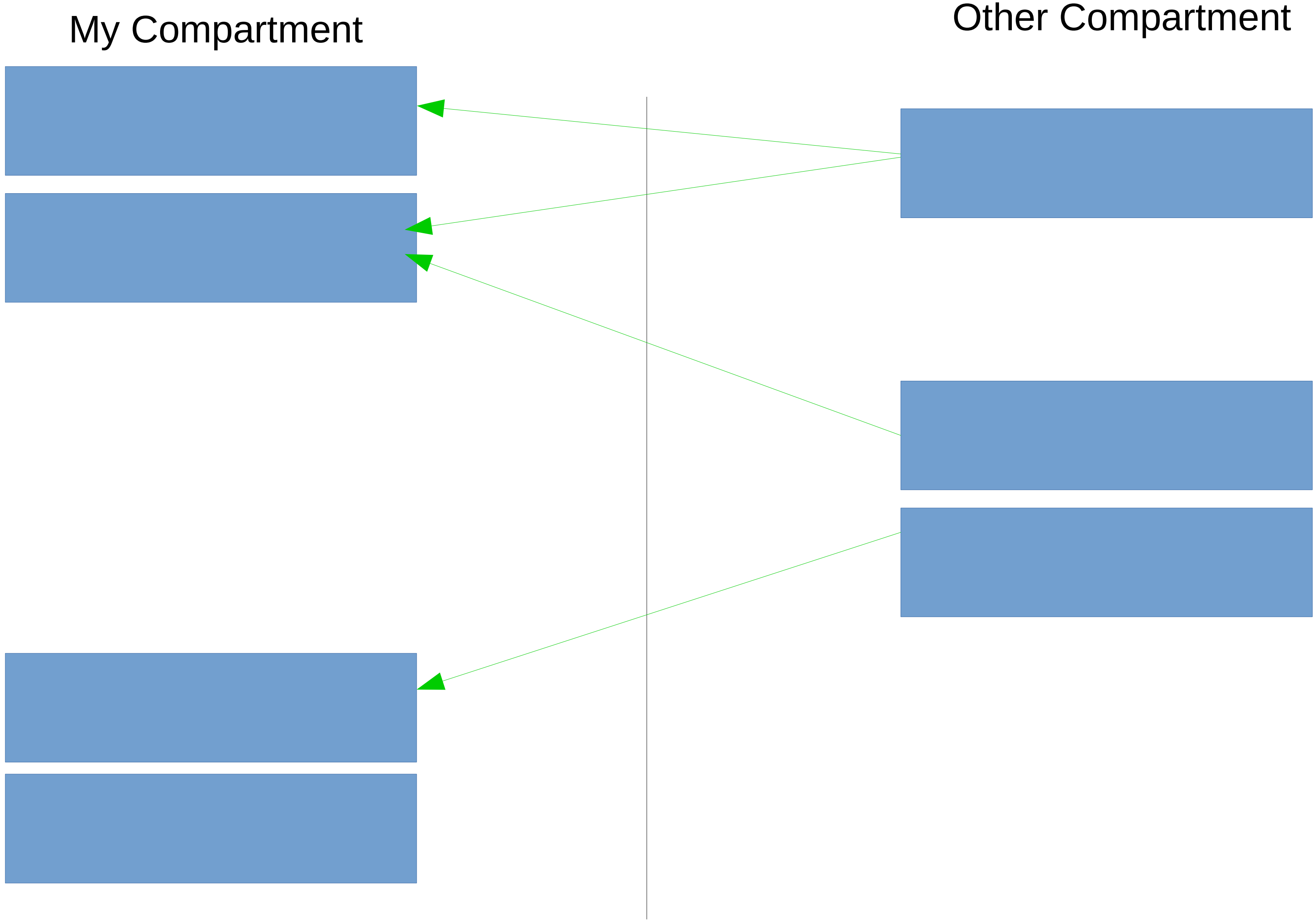
CFI

- CFI edges enforced between compartments
- Calls:
  - Cannot call into a compartments with a particular C thread if already in compartment
  - Can only call starts of functions, and only if exported
  - Cannot return to compartment out of order
  - Cannot return to compartment if return has already been used

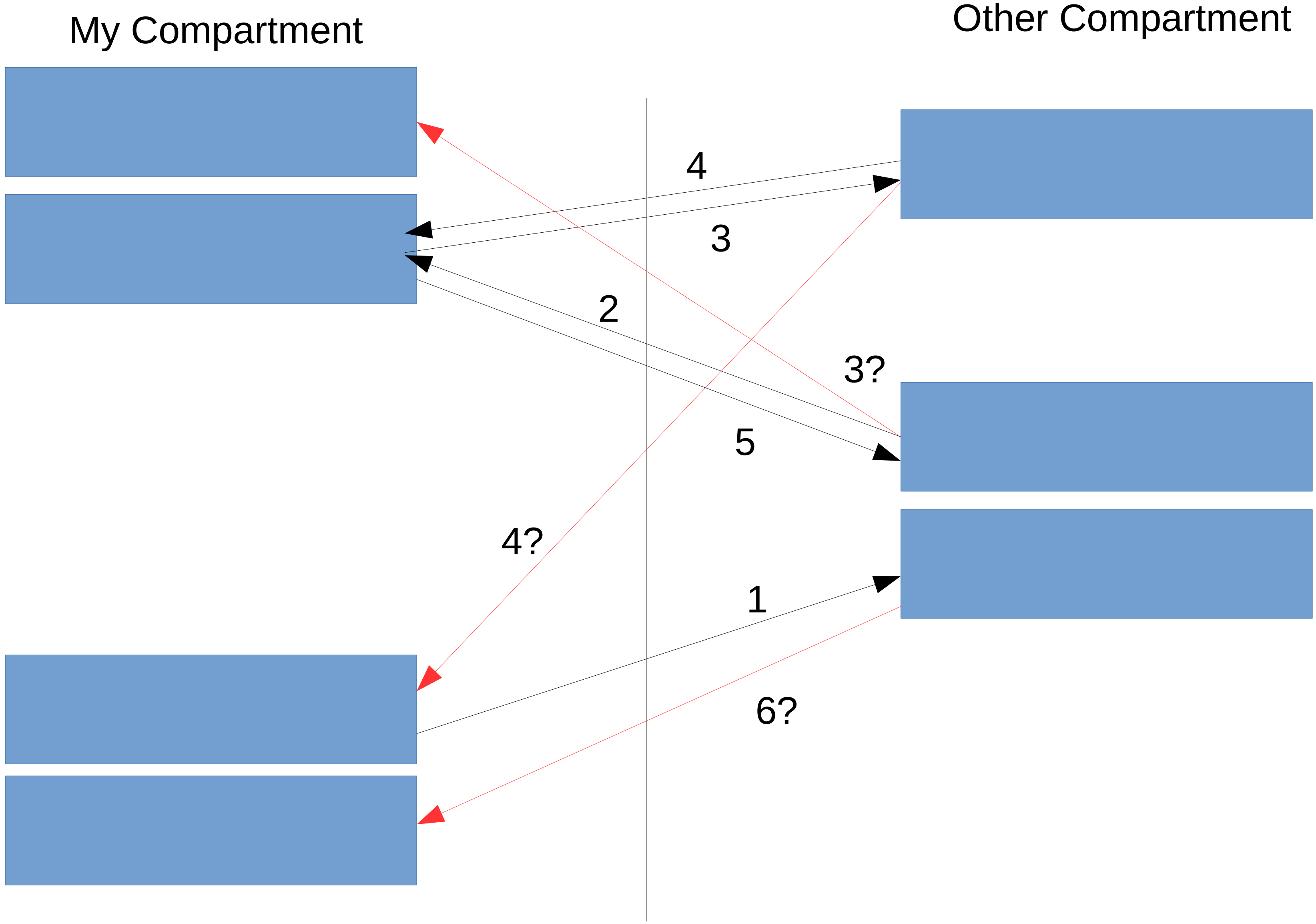
CFI



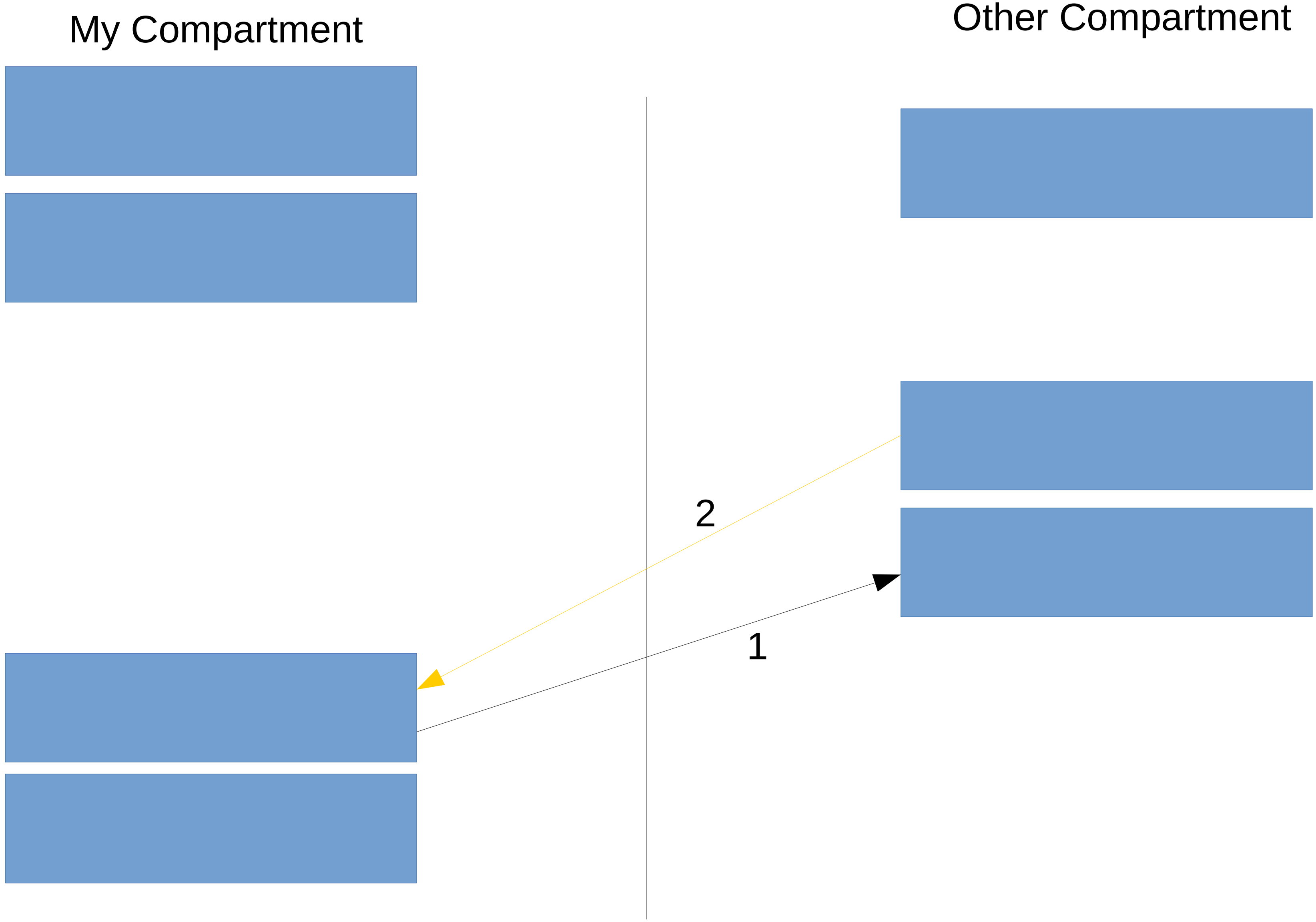
CFI



CFI



CFI



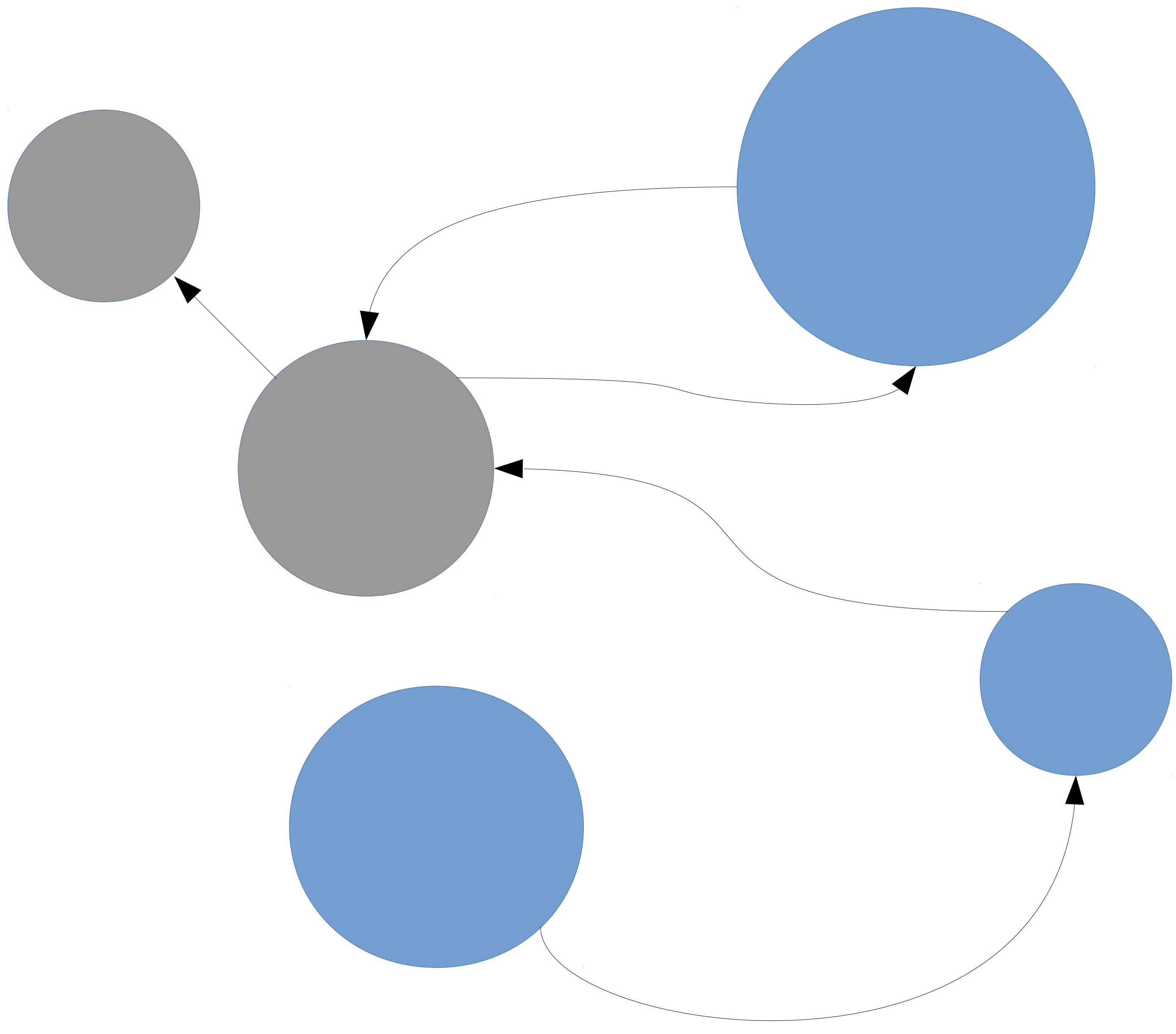
Sharing Capabilities

- Users are encouraged to share capabilities directly
  - Even across process boundaries!
- Means we cannot ever reuse memory
  - Temporal safety ensures we do not
- Lifetime gets more complicated
  - Malloc augmented with ‘claim’
  - Magic safe dereference instruction
  - Or just use exceptions

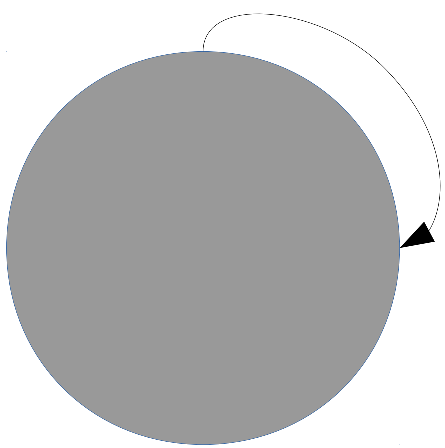
Foundations

- Identity
  - How we take a digest of a capability graph?
- Attestation
  - Given we can trust our own execution, can we verify another program has been loaded correctly?
- Send verifiable messages from and to

Capability Graph - Digest

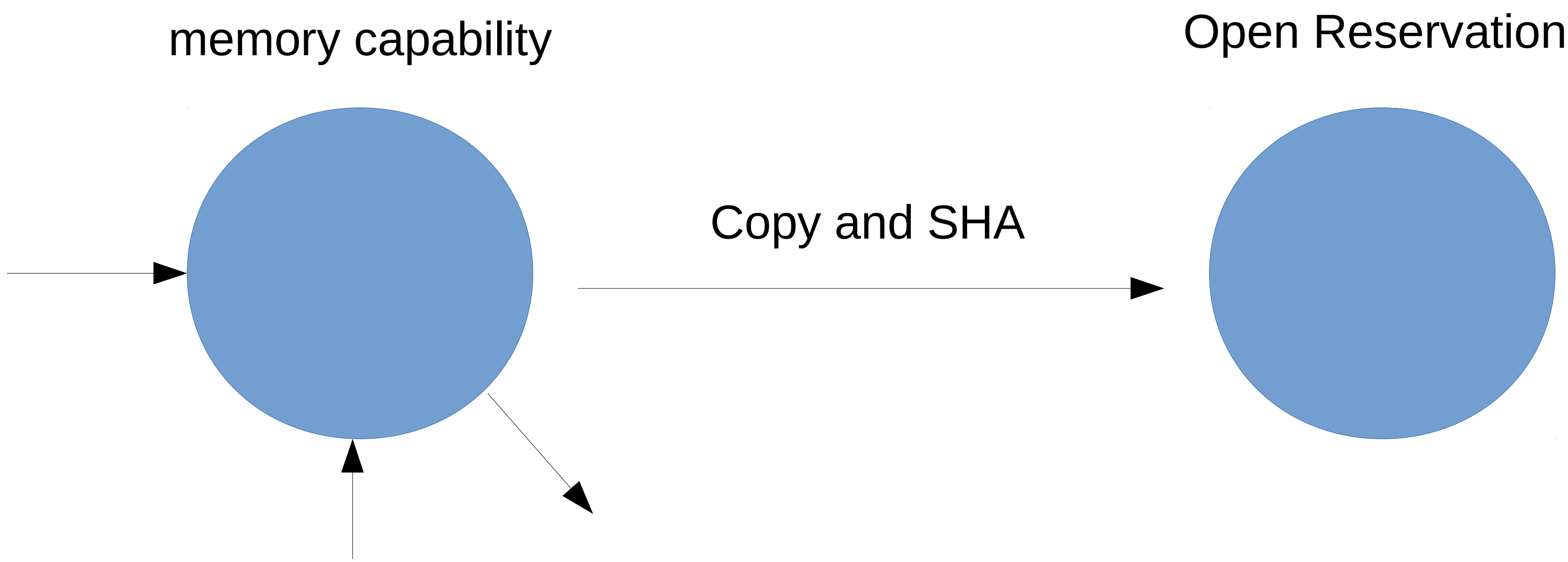


Foundations - Digest



Foundations - Creation

```
entry_t foundation_create(res_t res, size_t image_size,
                          capability image, size_t entry0,
                          size_t n_entries, register_t is_public)
```



Foundations

- Each foundation created has an identity
- Identity is a read-only memory capability
  - A sha256 of some contiguous bytes
  - A length
  - An offset of the first entry point

Foundations - Entry

```
void foundation_enter(entry_t entry)
```

- Jumps to entry point specified by entry token
- Creates a R/W capability for the foundation
- Grants an ‘authority’ token

Foundations - Exit

- Exit is implicit
  - Lose access to the capability
  - Already done by untrusting call
- Re-entry is also implicit
  - Explicit call only generally used for setup
  - Entry / Exit therefore has no overhead over normal untrusting call

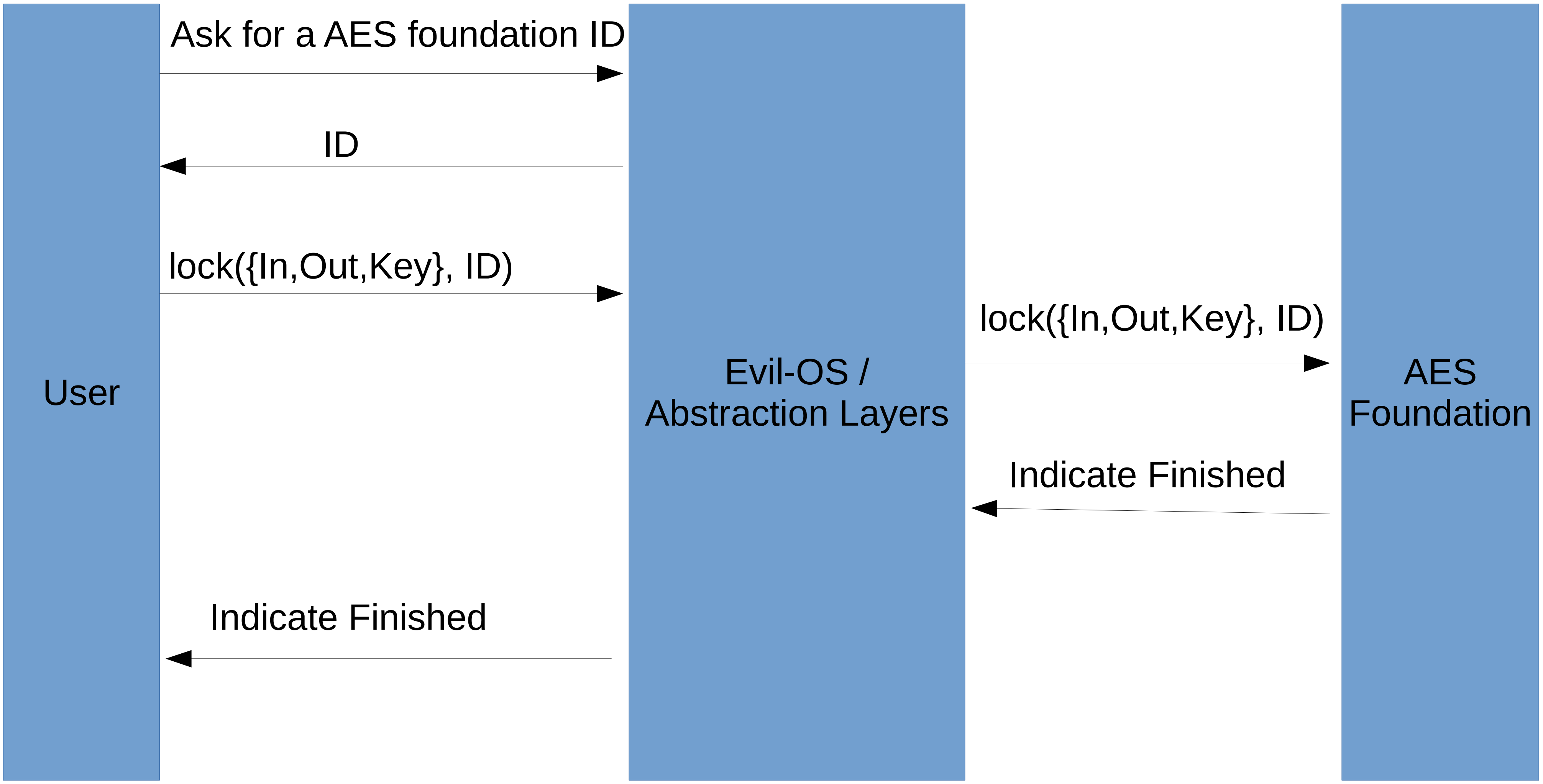
Foundations

- Authority token
  - Symmetric Key, can lock-sym unlock-sym
  - Asymmetric Key, can sign and unlock-asym
  - Can also derive identity token
- Identity Token
  - Asymmetric Public Key, can lock-asym

Foundations

- We can now treat the operating system and IO as an untrusted channel
- Foundations are identities, not compartments
  - i.e., multiple compartments may have the same or multiple authority tokens
- Replace encryption with foundation primitives

Example: AES


User
Ask for a AES foundation ID
ID
lock({In,Out,Key}, ID)
Indicate Finished
Evil-OS /
Abstraction Layers
lock({In,Out,Key}, ID)
Indicate Finished
AES
Foundation

Foundation Uses

- Load an entire OS in a foundation
  - Secure VM boot
- Load an entire program
  - Secure against malicious program loader / OS
- Secure Deduplication
- Callback signing

Foundations vs encryption

- Scalable
  - Merkle trees do not scale
  - Encryption is expensive
- Nestable
  - Can hold multiple, not necessarily hierarchical, set of authority tokens

Revocation

- Capabilities get *everywhere*

Revocation - Stage1

- Unmap
  - Nano kernel will not allow remapping virtual
  - Nano kernel will not allow remapping physical until zerod
  - Allows for ~60bits of space before we cannot use this method
- Can merge old reservation handles
  - Only need one handle per finished range
  - Can unmap metadata as well!

Revocation – Stage2

- Concurrent scan of all **physical** memory
  - Revoke a vast **virtual** range
  - Can do without suspending any users
  - Takes on the order of ms

- Will return a new open reservation for the virtual space

Problems with CheriOS

- This is all defunct with DMA
  - Need a capability aware DMA engine
- Fragmentation due to non-reuse (heap)
  - Need nano kernel assisted compacting allocator

State of CheriOS

- Runs on QEMU/FPGA CHERI-MIPS
- Has a FAT file-system
- Runs a HTTP server
  - Webstack is a port of LWIP
  - HTTP server is a port of NGINX
  - Hosting a webpage at cherios-fpga.sec.cl.cam.ac.uk
- Publicly available!
  - https://github.com/CTSRD-CHERI/cherios/tree/lawrence

FPGA Numbers

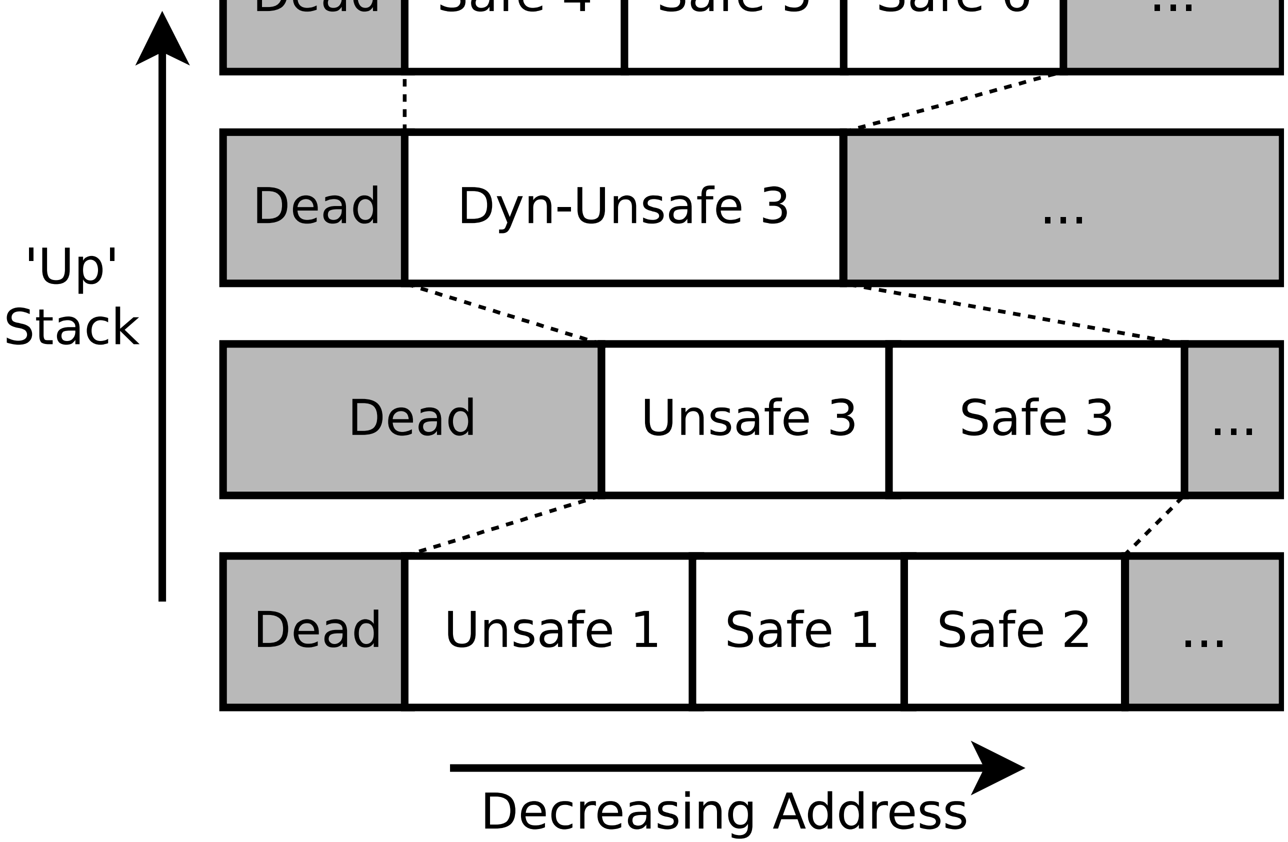
- RPC (Cross process microkernel sync message-send, 2 context switches)
  - 1056 cycles (Trust microkernel)
  - 1256 cycles (Distrust microkernel)
- Calling between compartments (call and return)
  - 5 instructions, 26 cycles (To nanokernel)
  - 71 cycles, Trusting / Trusted
  - 119 cycles, Trusting / Untrusted
  - 152 cycles, Unstrusting / Trusted
  - 210 cycles, Untrusting / Untrusted

Questions!

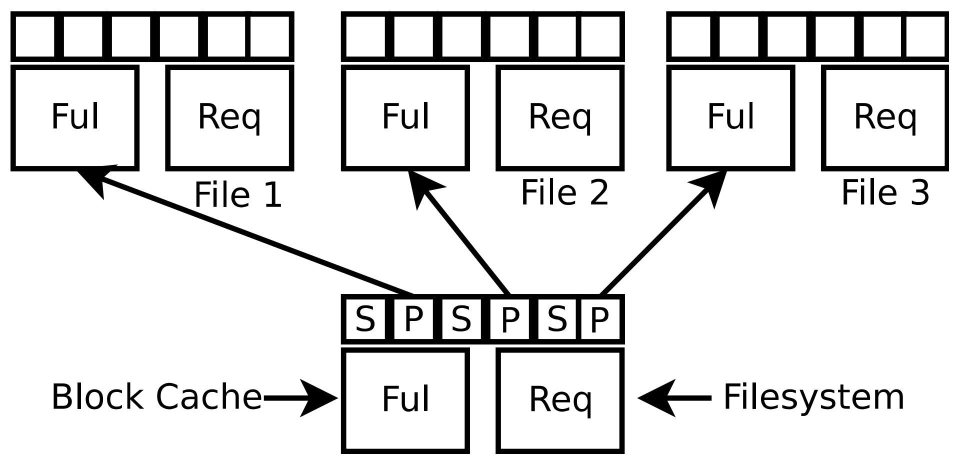
Public foundation

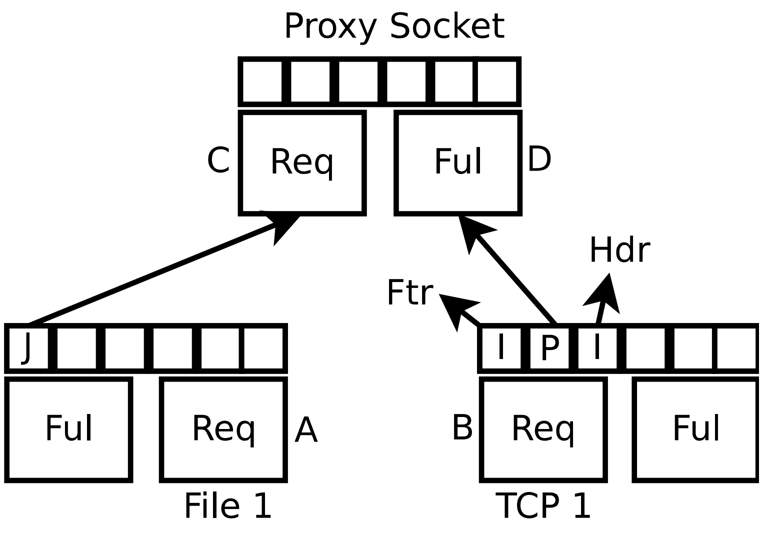
- Contents are not secret, but should not be modified
- Can request read only capability given an ID
- Used for secure deduplication

Slinky Stacks



CheriOS sockets





The Nanokernel - Access

- Accessed via CCall
- Interface can be provided by syscall
  - syscall will always trap to the nanokernel
  - Nano kernel will not deliver this syscall to the system
  - Allows users to circumvent the OS if they desire
  - OS can tell nano kernel to not give out particular interfaces to users