Twizzler: Rethinking the Operating System Stack for Byte-Addressable NVM

Professor Ethan L. Miller

Center for Research in Storage Systems (UCSC) & Pure Storage

Work done at CRSS

What's wrong with the current OS stack?

- **Modern operating systems were designed for block-oriented I/O**
  - Go through the OS for *each* access to persistent data: slow
  - Sharing through memory is awkward
- **We can do better!**
  - Implement a *data-centric* OS
  - Keep the OS out of the data access path
- **But the system must**
  - Allow sharing
  - Enforce security

Memory hardware trends

`sys_read`

~100–300 ns

Growing, becoming persistent

~1 us

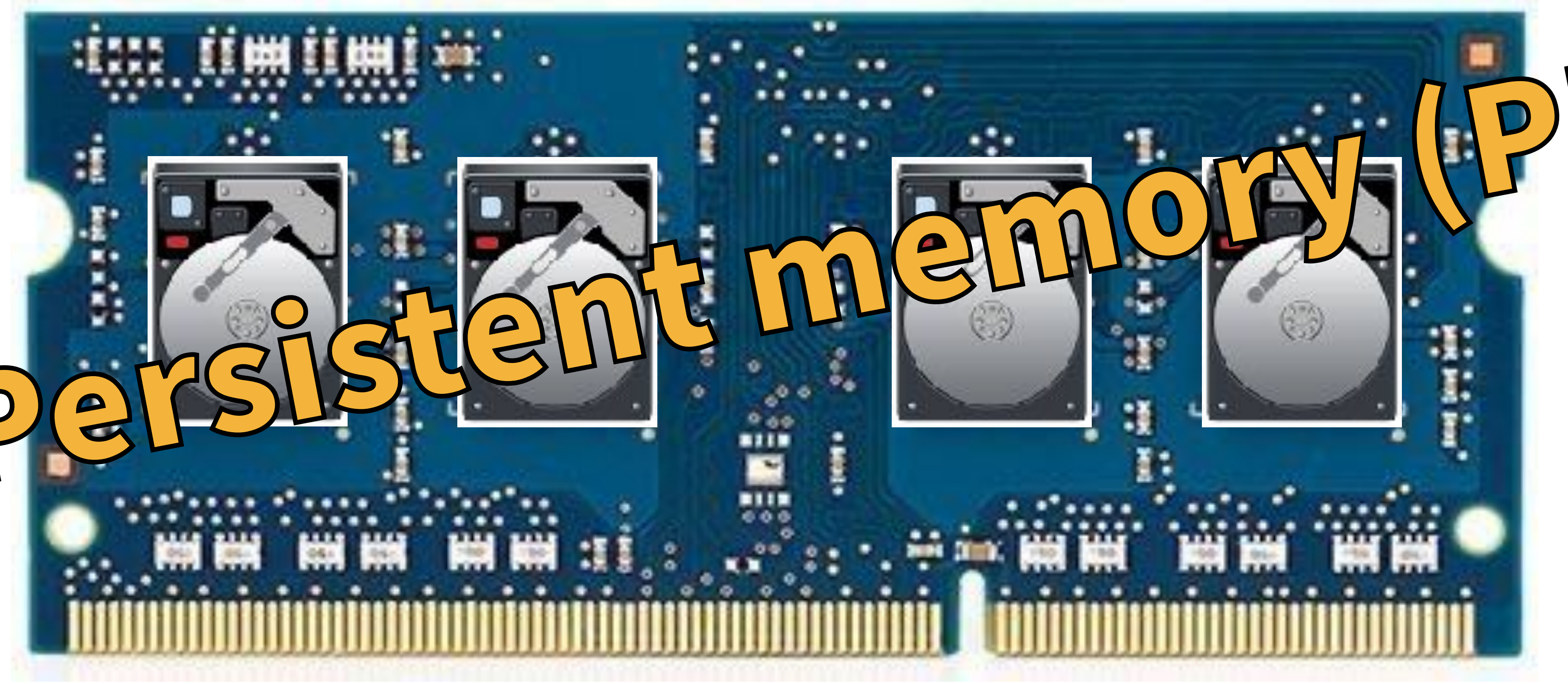
Outdated interface

~1–10 ms

No direct CPU access

Persistent data should be operated on *directly* and *like memory*

Memory hardware trends



`sys_read`

~100–300 ns

Growing, becoming persistent

~1 us

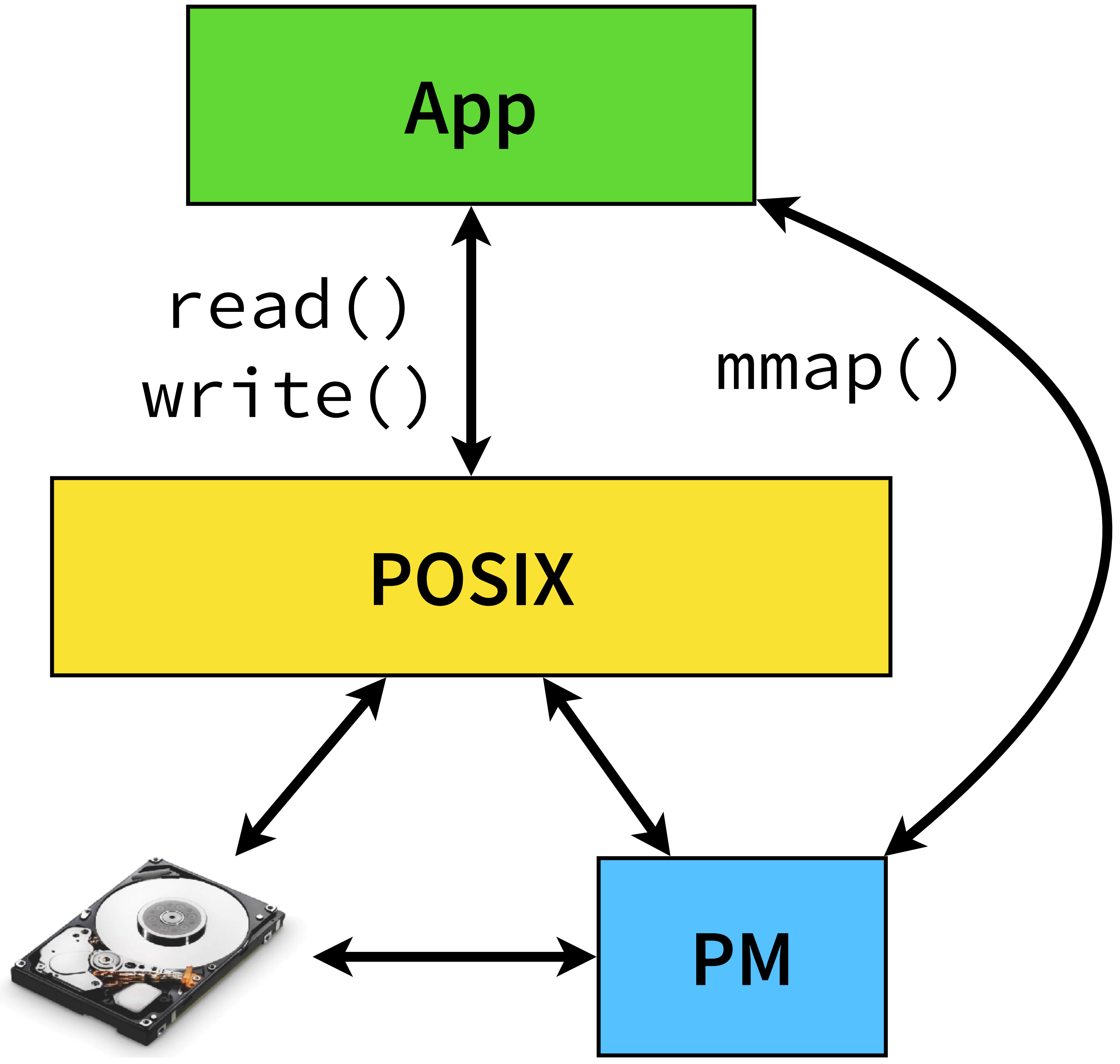
Outdated interface

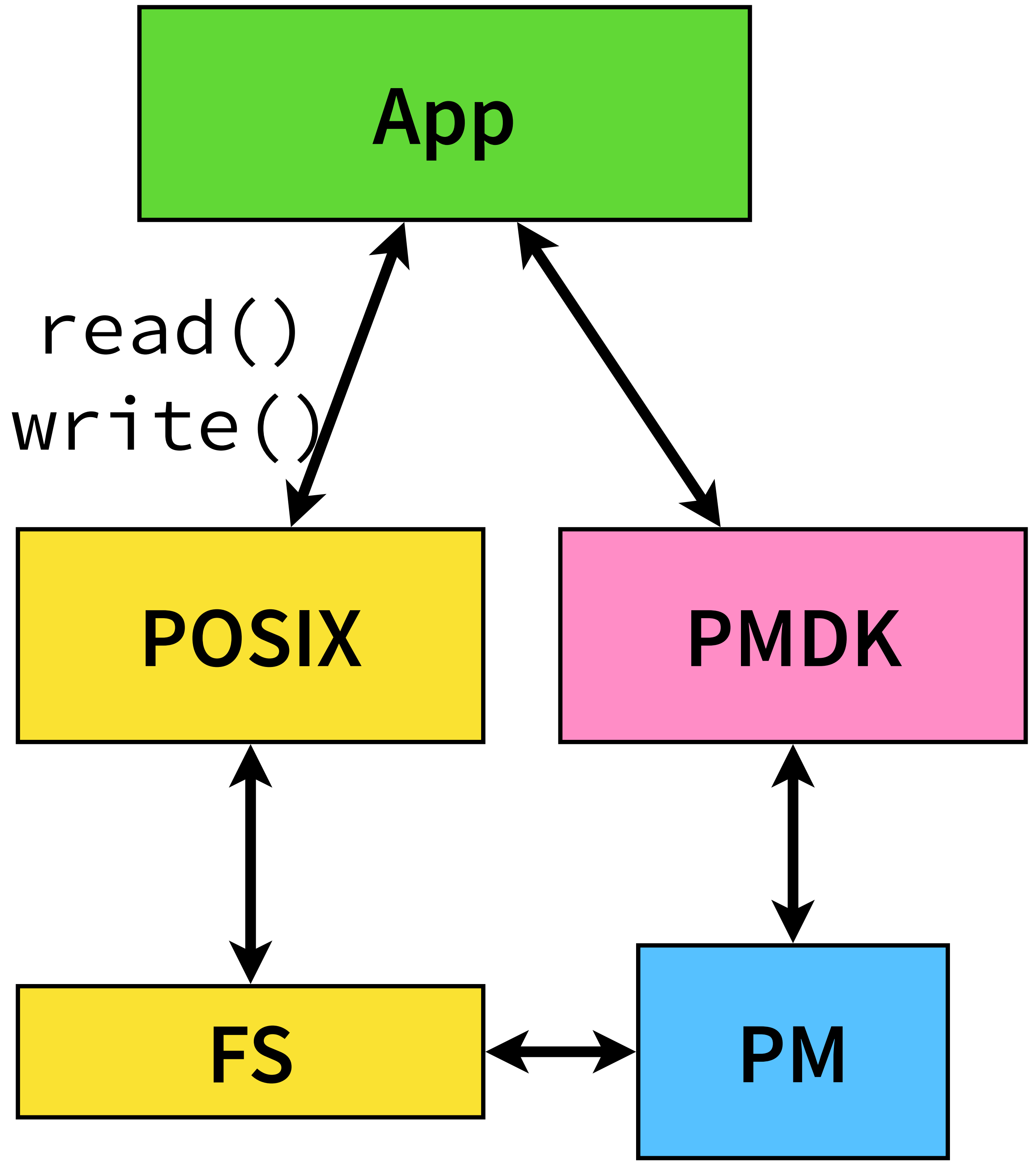
~1–10 ms

No direct CPU access

Persistent data should be operated on *directly* and *like memory*

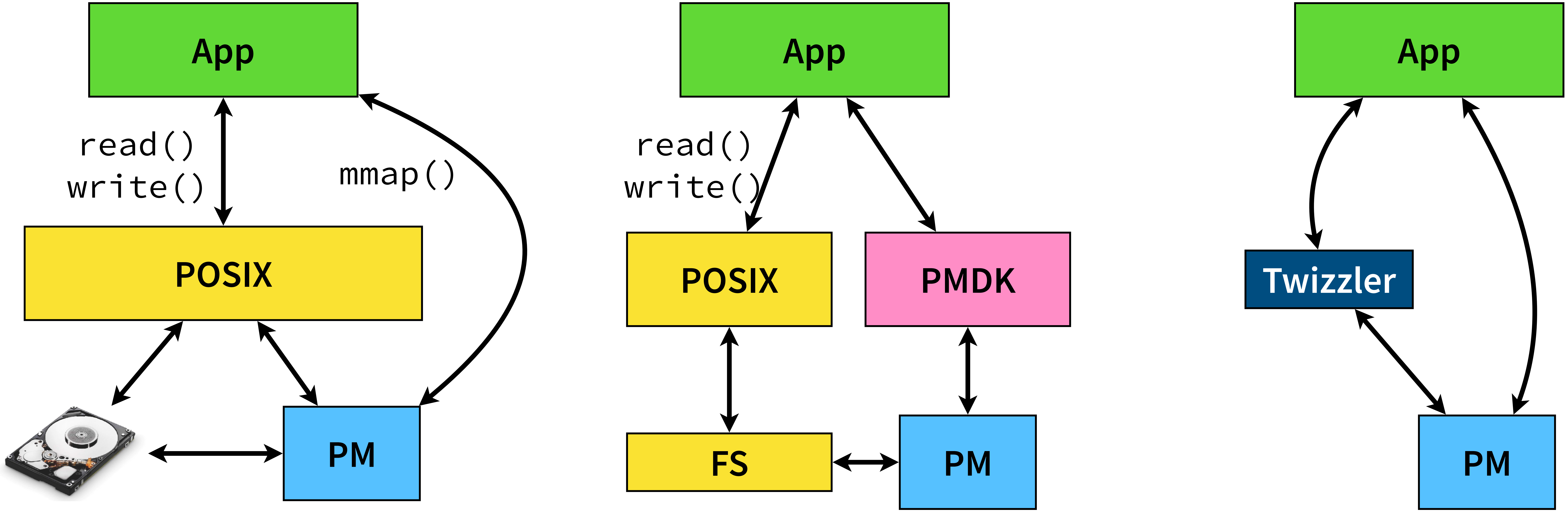
Different approaches for PM



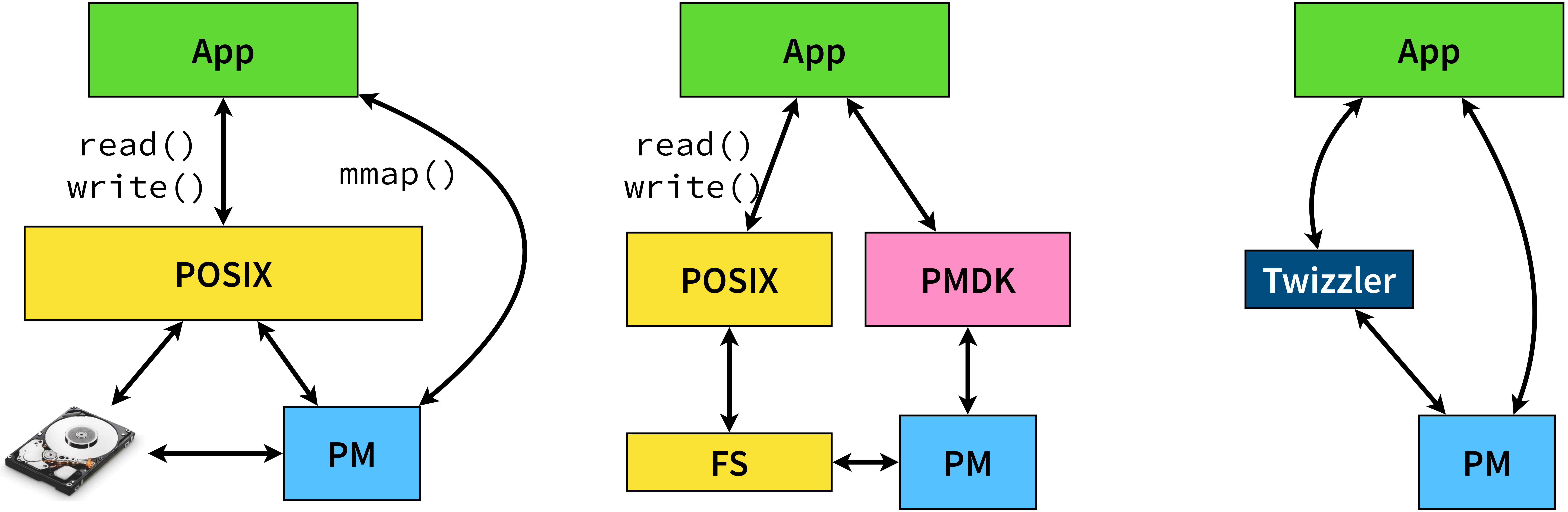


- Remove the kernel from the persistence path
- Design for pointers that last forever
- Provide strong and flexible security

Different approaches for PM



Different approaches for PM



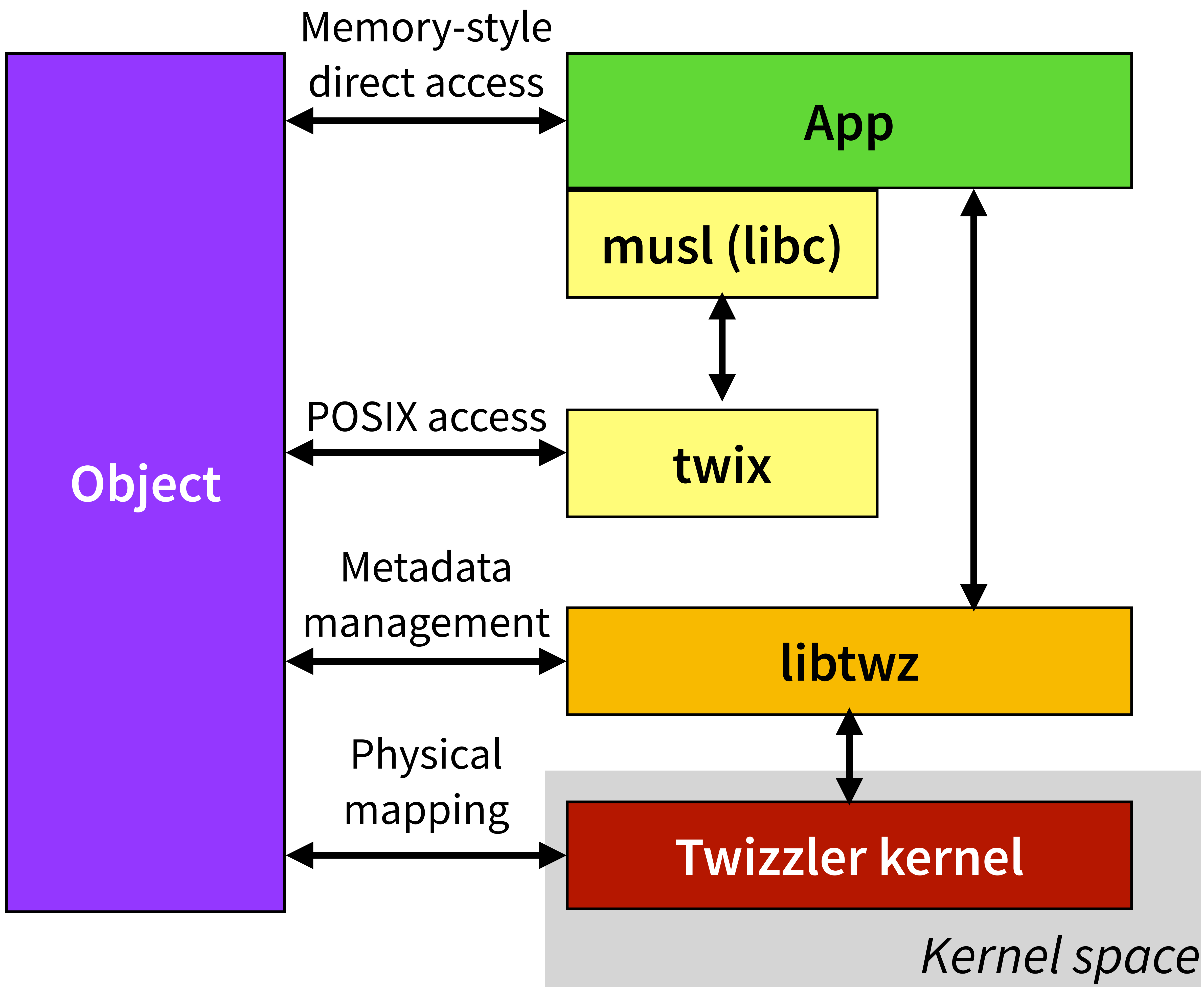
Processes and virtual addresses are the wrong abstraction!

The data-centric OS

- Data is the core concept in the OS
  - All pointers consistent and valid in *all* threads
    - Access still subject to security constraints
  - All threads "see" data the same way: no per-process virtual address space
    - Minimal per-thread state
- OS manages access to memory-based persistent data structures
  - Leverages MMU to provide consistent view and security
- Privileged kernel can be very small!

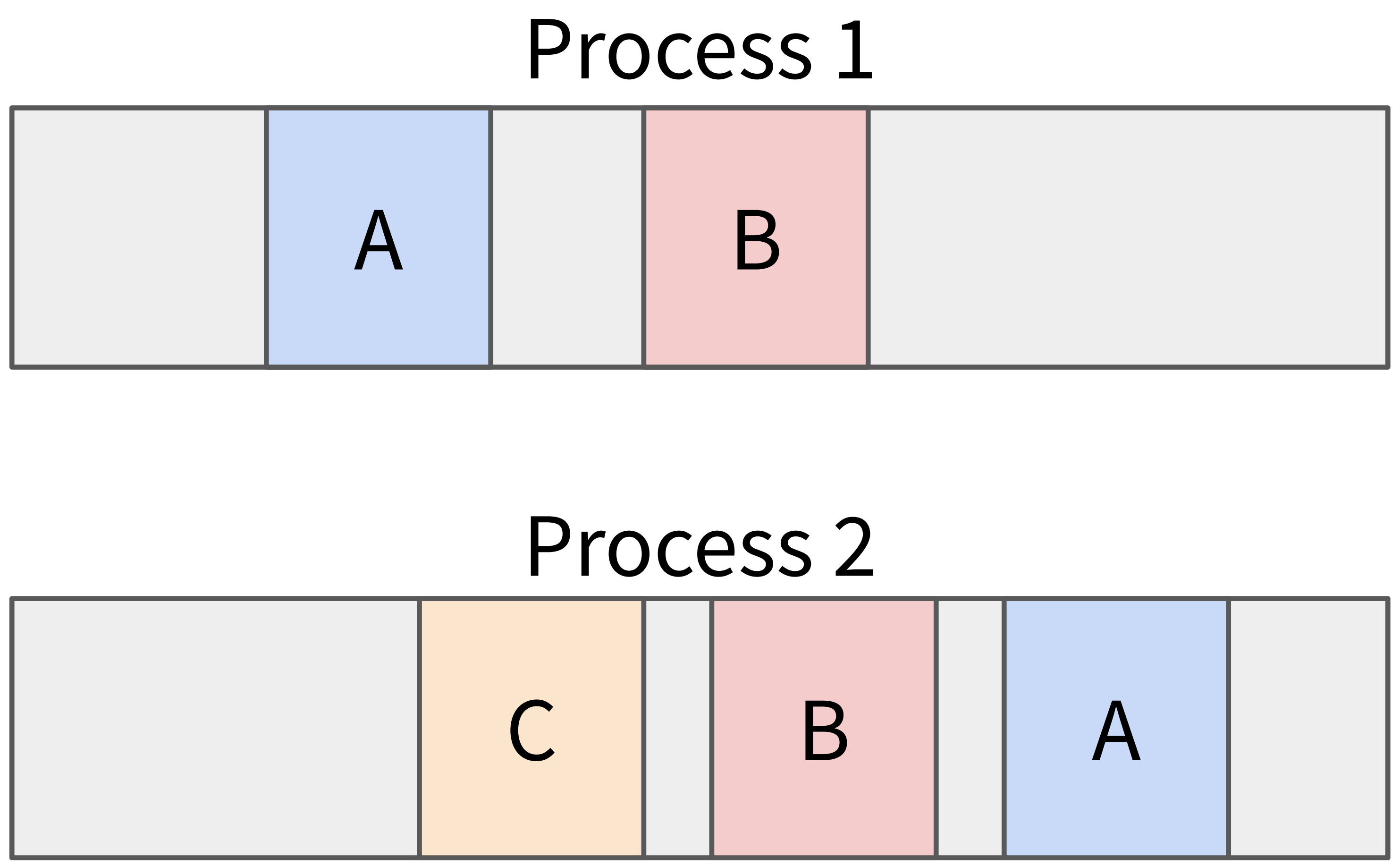
Our approach: Twizzler

- Object-based
  - Object is a region of memory
  - Single object: semantically-related data
- Minimal kernel
  - Manages physical resources
  - Manages MMU and scheduling
  - Ensures security policies followed
- LibOS (libtwz)
  - Most traditional OS functionality implemented in user-space
- twix emulates POSIX



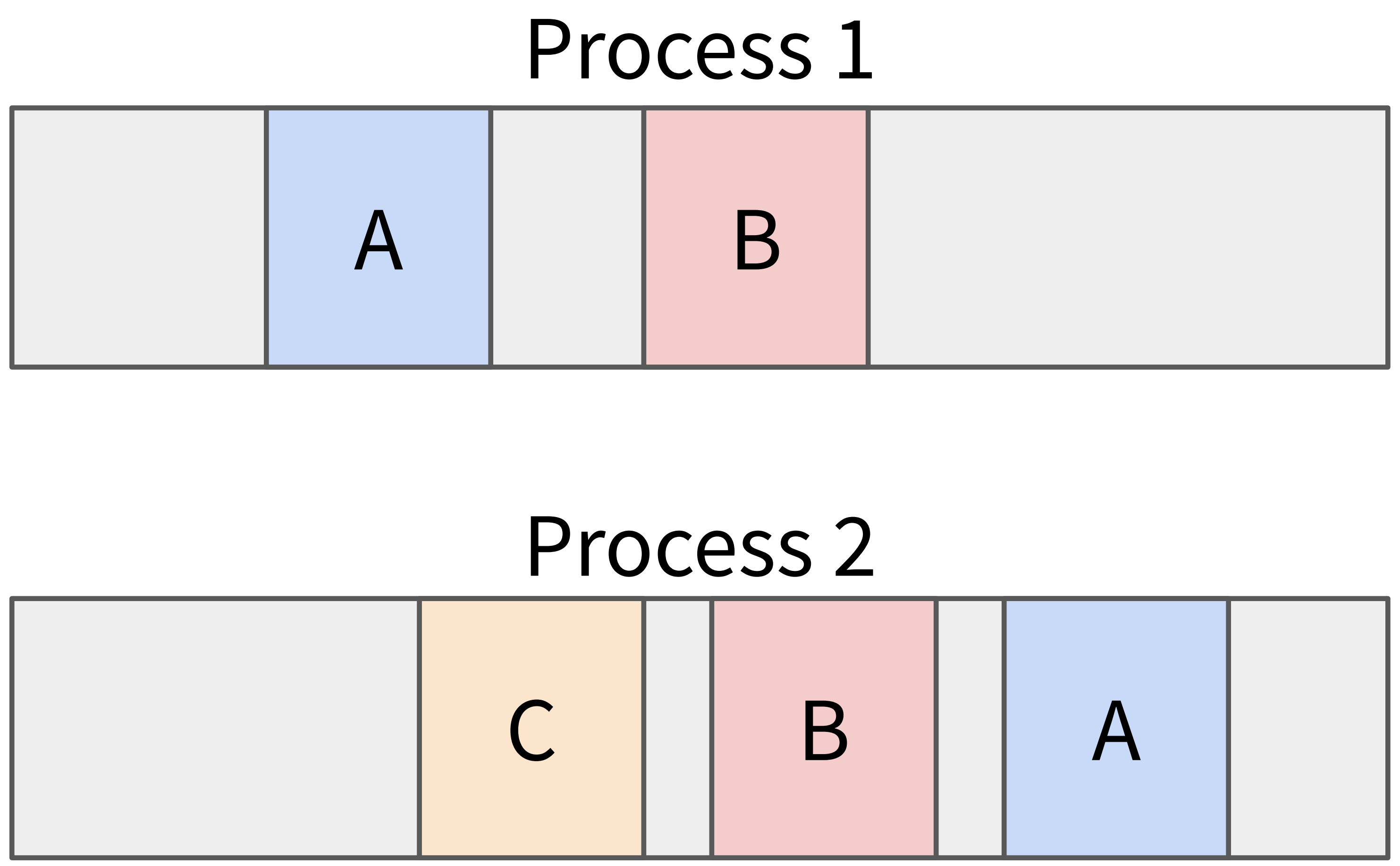
Data-centric programming

Persistent data should be operated on *directly* and *like memory*



Data-centric programming

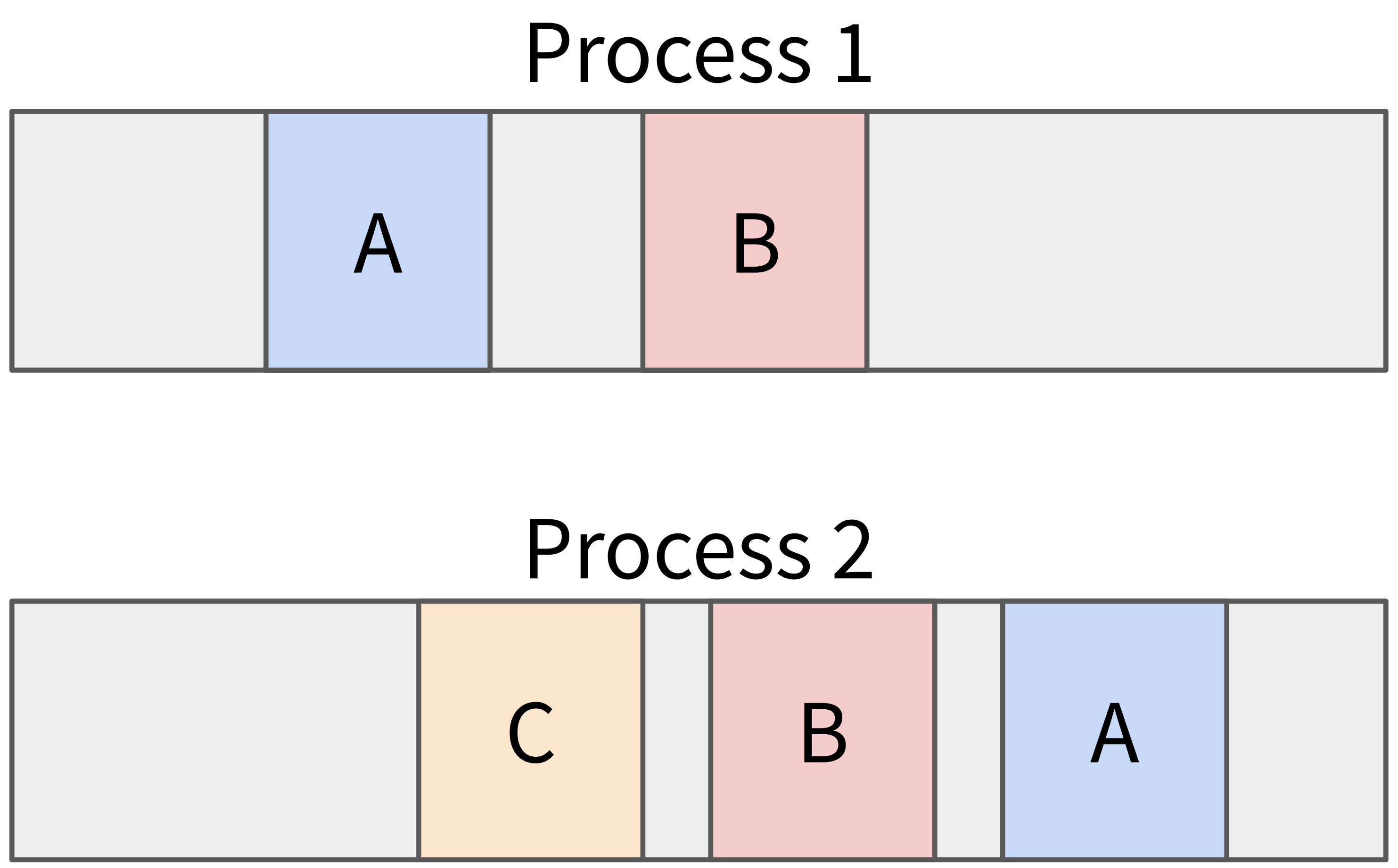
Persistent data should be operated on *directly* and *like memory*



Pointers must be valid *anywhere* in *any* thread's address space

Data-centric programming

Persistent data should be operated on *directly* and *like memory*

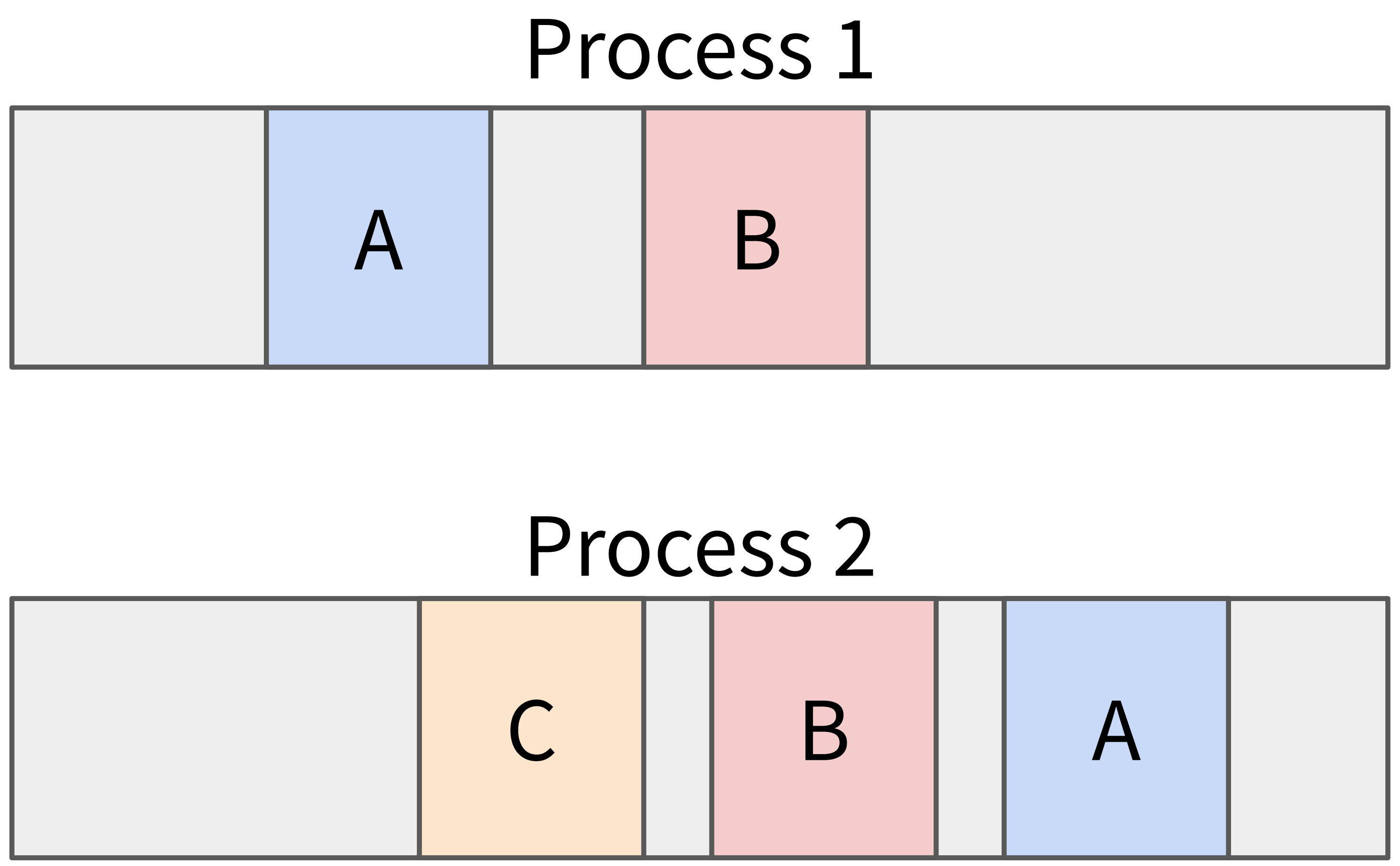


Pointers must be valid *anywhere* in *any* thread's address space

Pointers may be *cross-object*: referring to data within a different object

Data-centric programming

Persistent data should be operated on *directly* and *like memory*

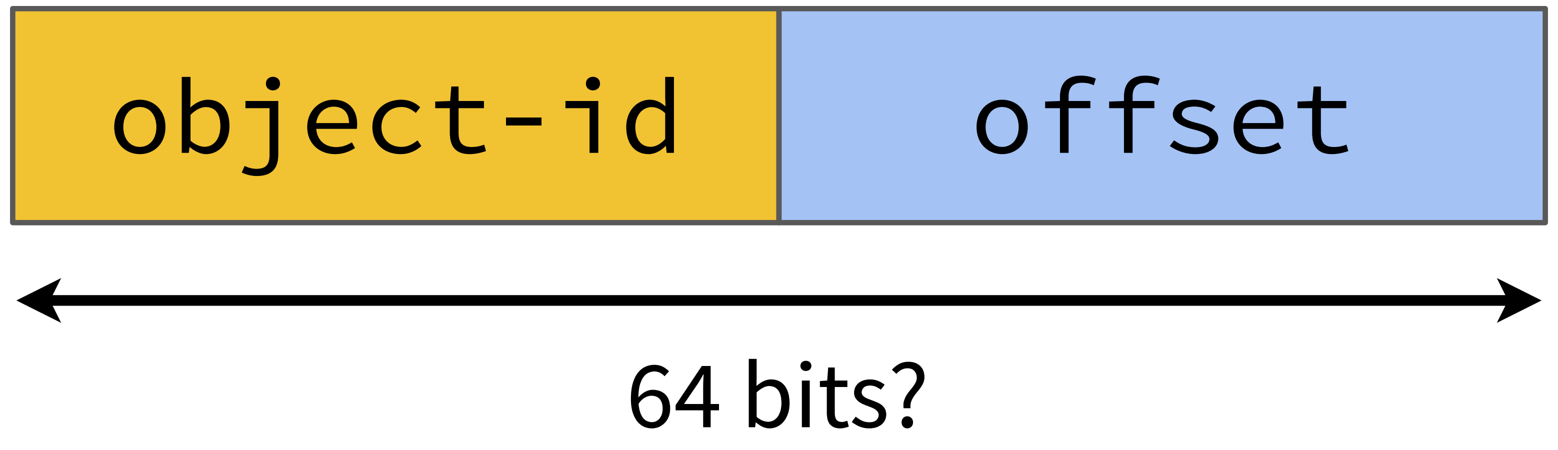


Pointers must be valid *anywhere* in *any* thread's address space

Pointers may be *cross-object*: referring to data within a different object

object-id	offset

Persistent pointers: format



Requirement: keep pointers 64-bits.

Avoids increasing hardware complexity and memory usage

Problem: object ID and offset are too big to fit

Persistent pointers: indirection

CENTER FOR RESEARCH IN STORAGE SYSTEMS

object-id	offset

64 bits

Pointers interpreted relative to the object in which they're stored!

Foreign Object Table

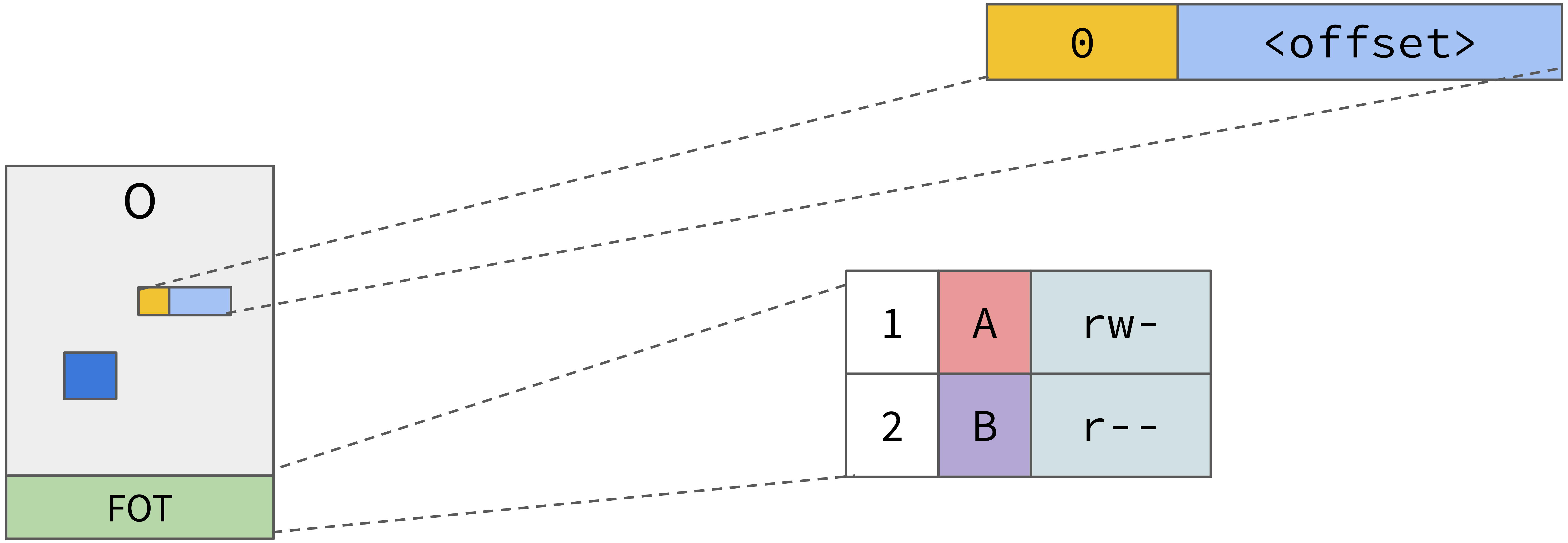
1	object ID	flags
2	object ID	flags

…

Object Layout

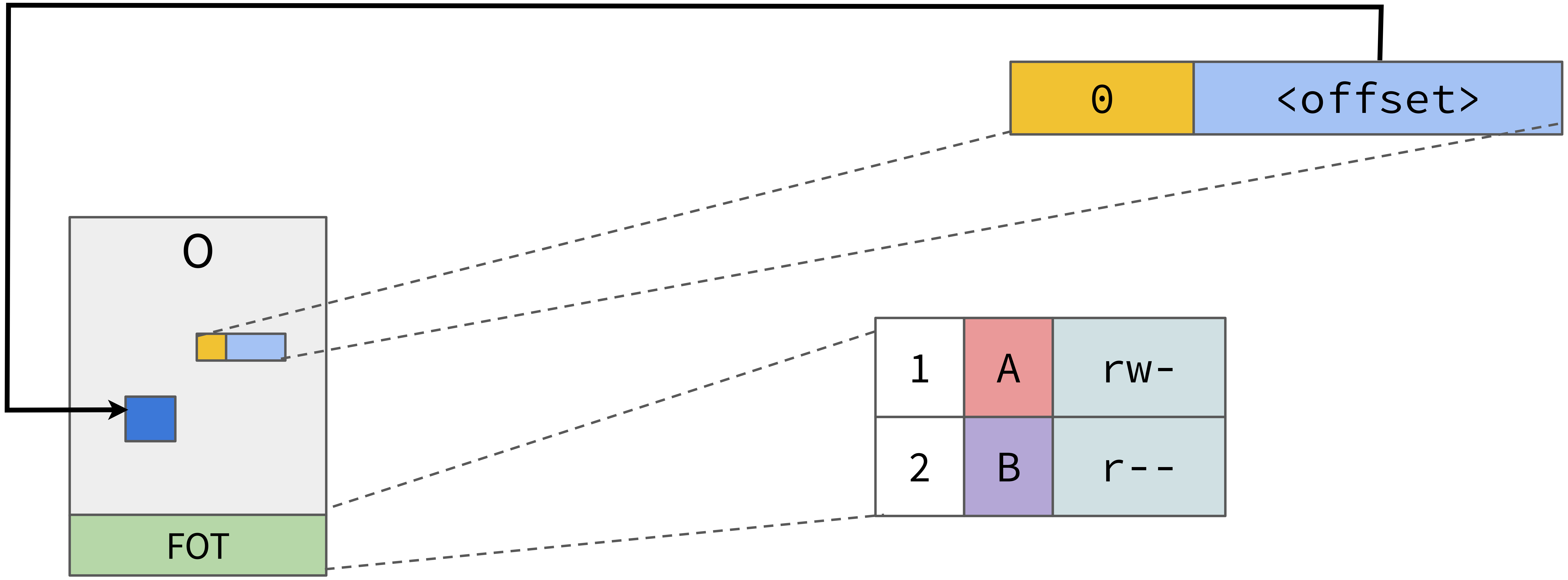
FOT	Data

Persistent pointers: translation



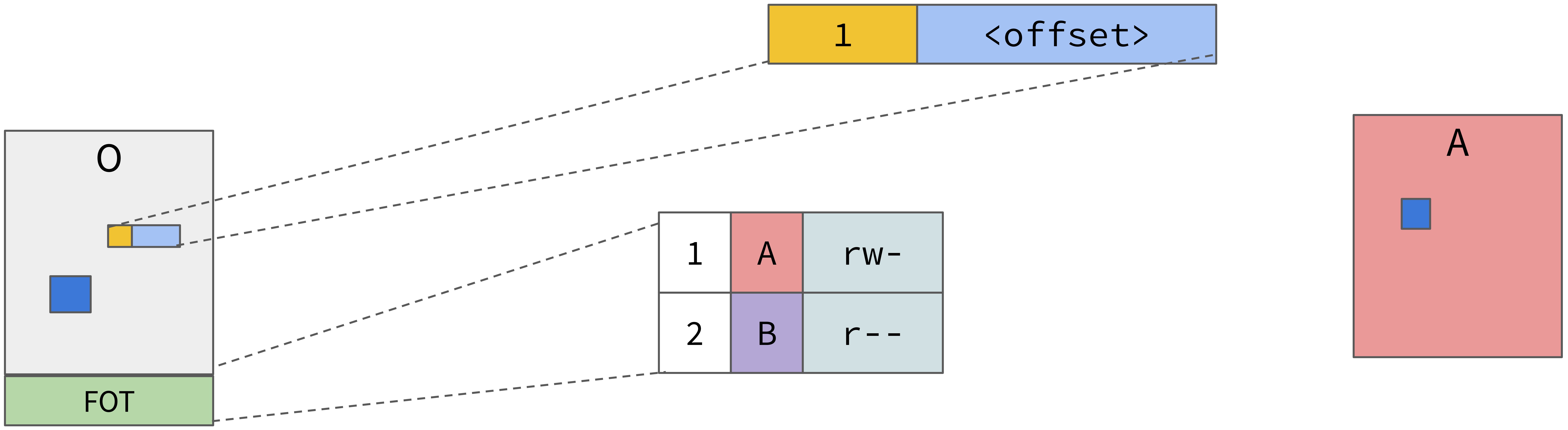
FOT entry of 0 means “self” pointer—points within the same object.

Persistent pointers: translation



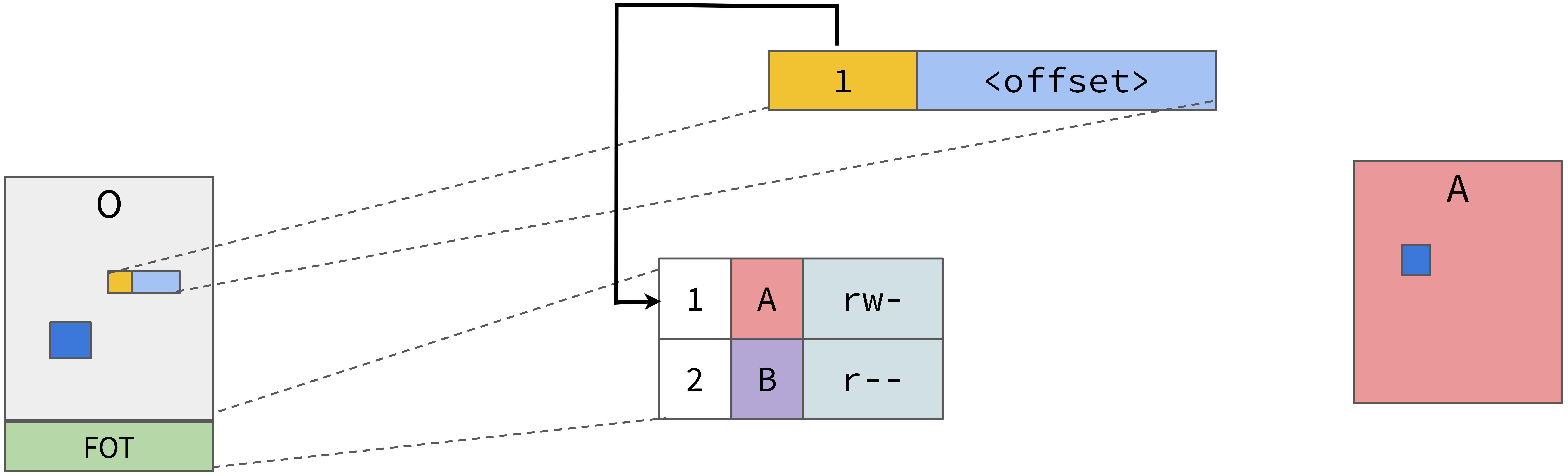
FOT entry of 0 means “self” pointer—points within the same object.

Cross-object persistent pointers



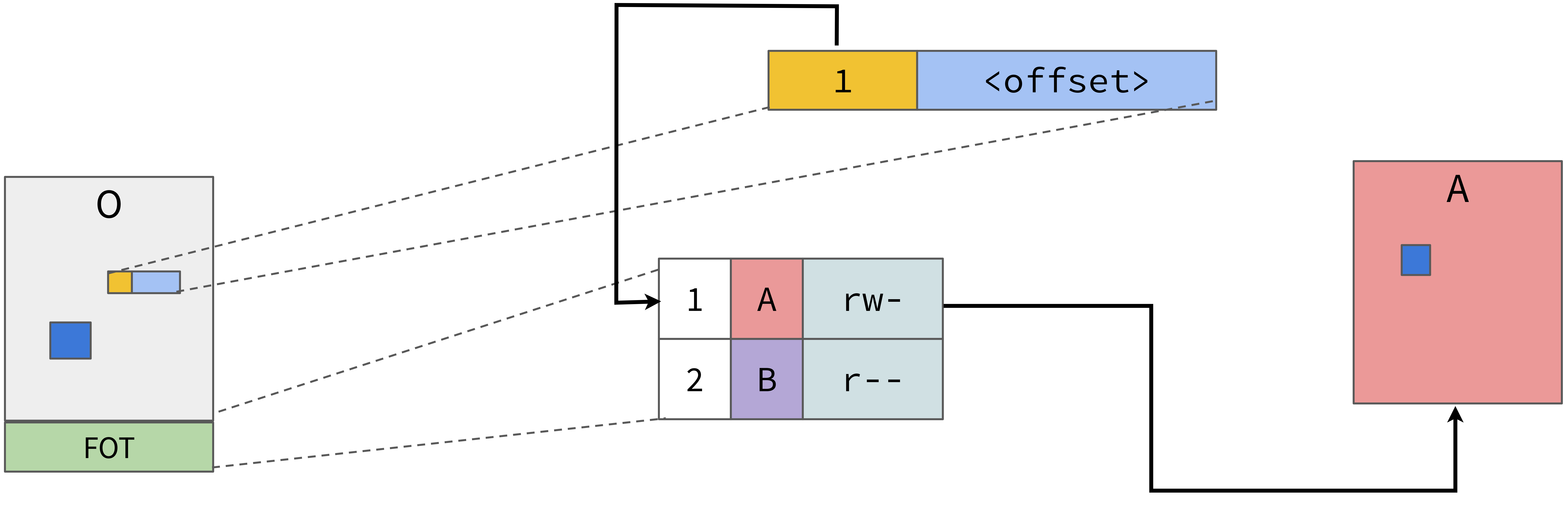
FOT entry of >0 means “cross-object”—points to a different object.

Cross-object persistent pointers



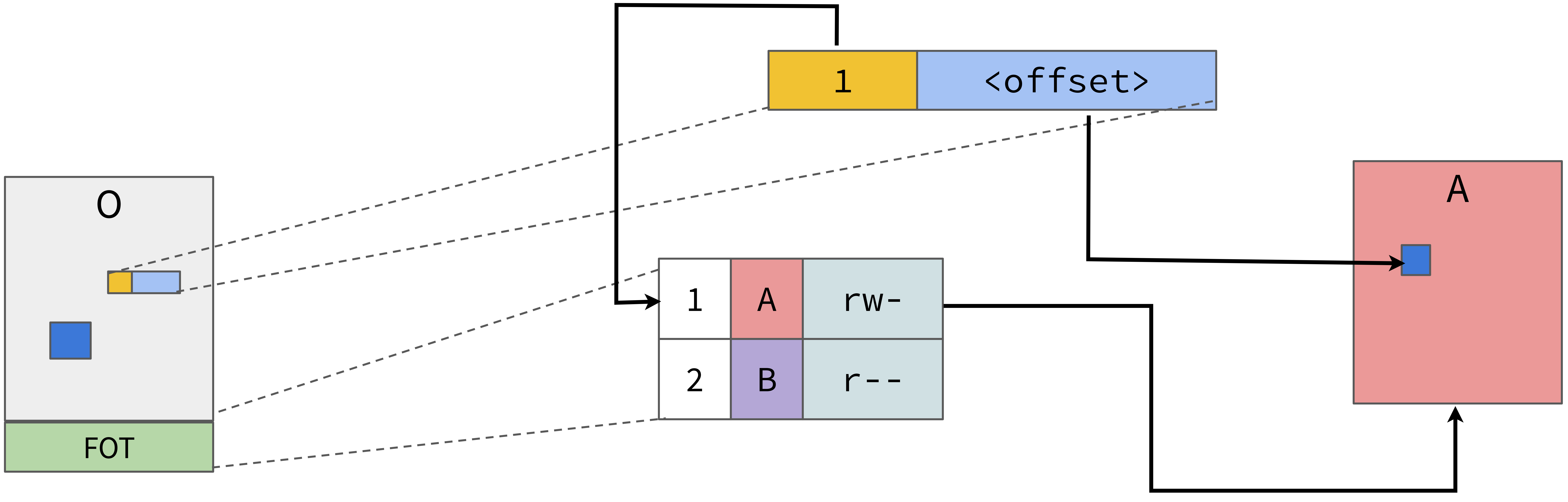
FOT entry of >0 means “cross-object”—points to a different object.

Cross-object persistent pointers



FOT entry of >0 means “cross-object”—points to a different object.

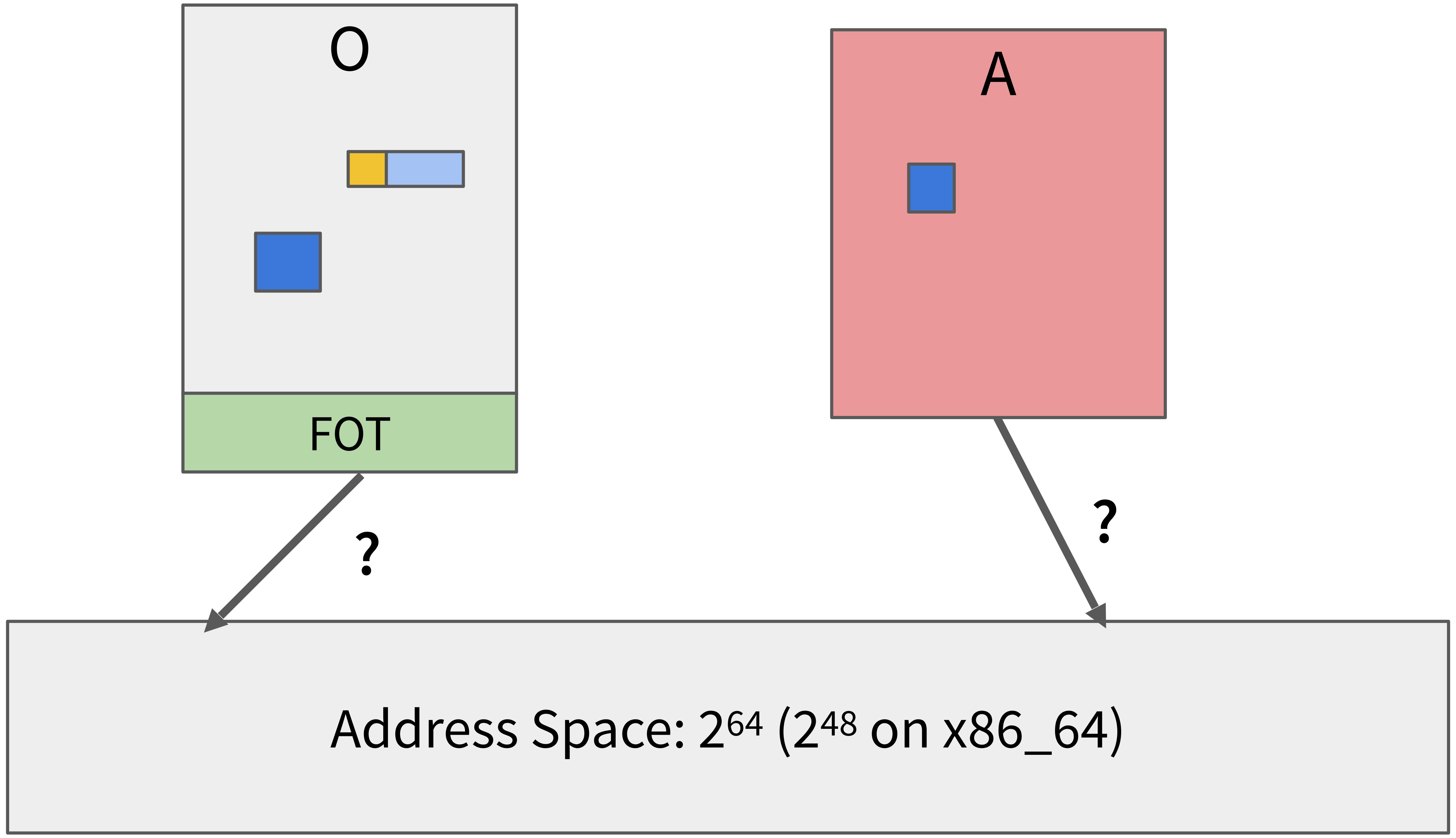
Cross-object persistent pointers



FOT entry of >0 means "cross-object"—points to a different object.

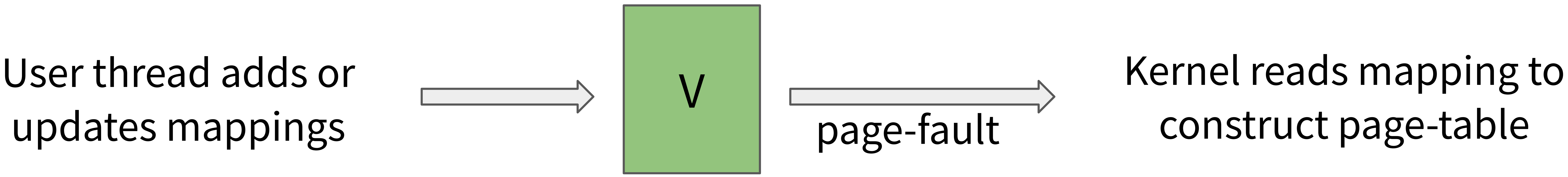
Persistent pointers: API

- Mapping should be transparent to applications
- Virtual address space abstraction does not fit with the object:offset model
- User-level LibOS handles address translation
  - Currently done as inline function calls (very fast)
  - Could be inserted directly by compiler



Ephemeral views of persistent objects

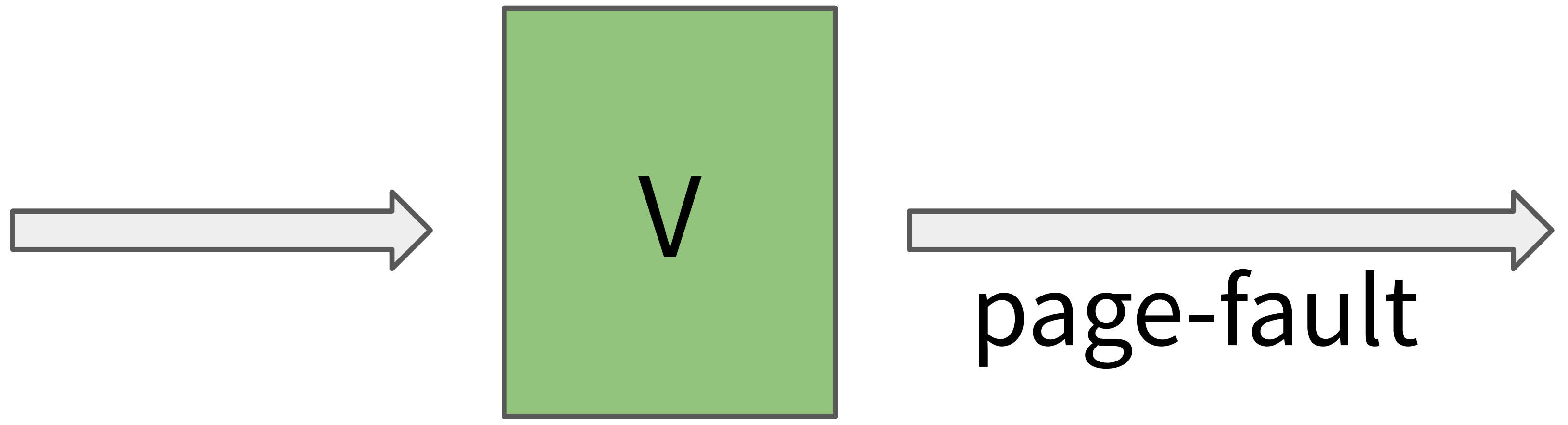
- A view allows threads to define their virtual address space layout
  - Thread requests objects at particular locations in a table shared with the OS: no system call!
  - Kernel maps in the objects on a page fault if access is allowed
  - Provides an ephemeral “window” to persistent objects with persistent pointers
- **Sharing table between user space and kernel space reduces system calls**



Views: implementation

A *view* allows user-space to define the virtual address space layout without a system call.

User thread adds or updates mappings



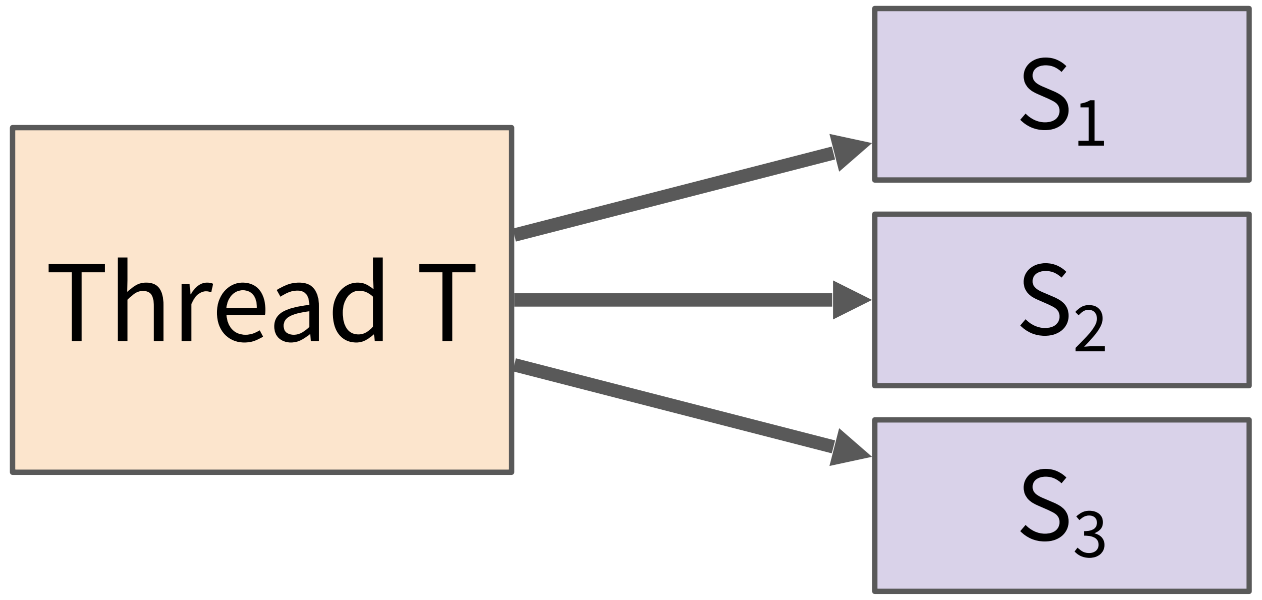
Kernel reads mapping to construct page-table

A view is like a page table that the kernel uses to construct a real page-table

slot #	target	flags
0	A	r-x
1	B	rw-
…		
m	V	rw-

Virtual address space of view object V

Security and access control

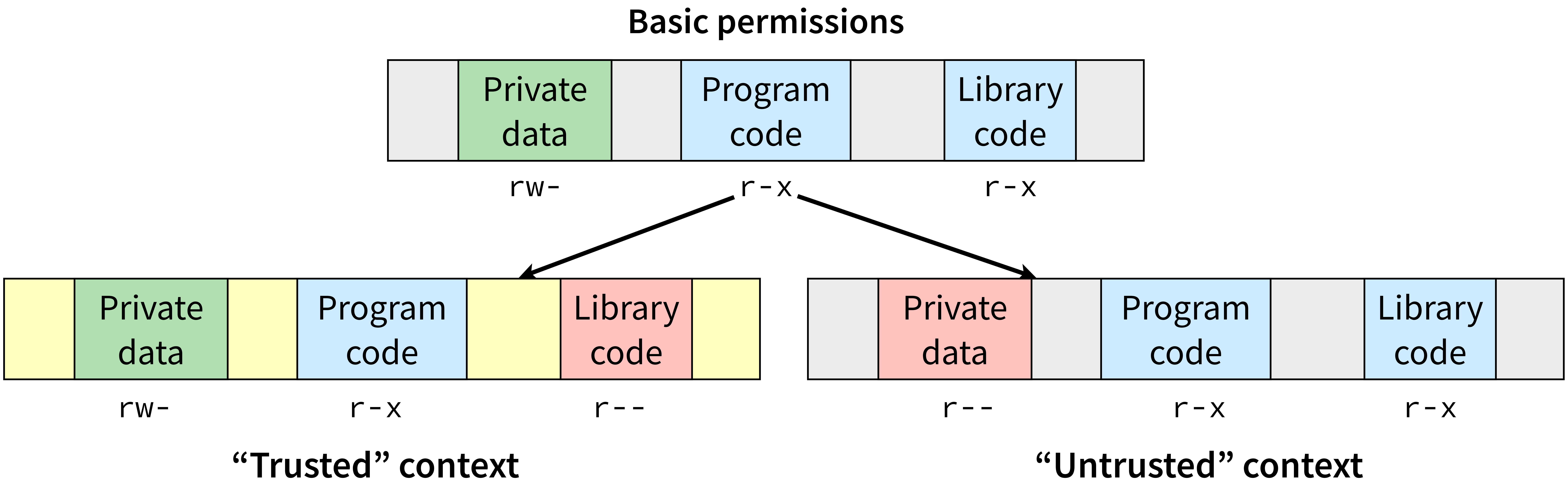


Threads run in
security contexts

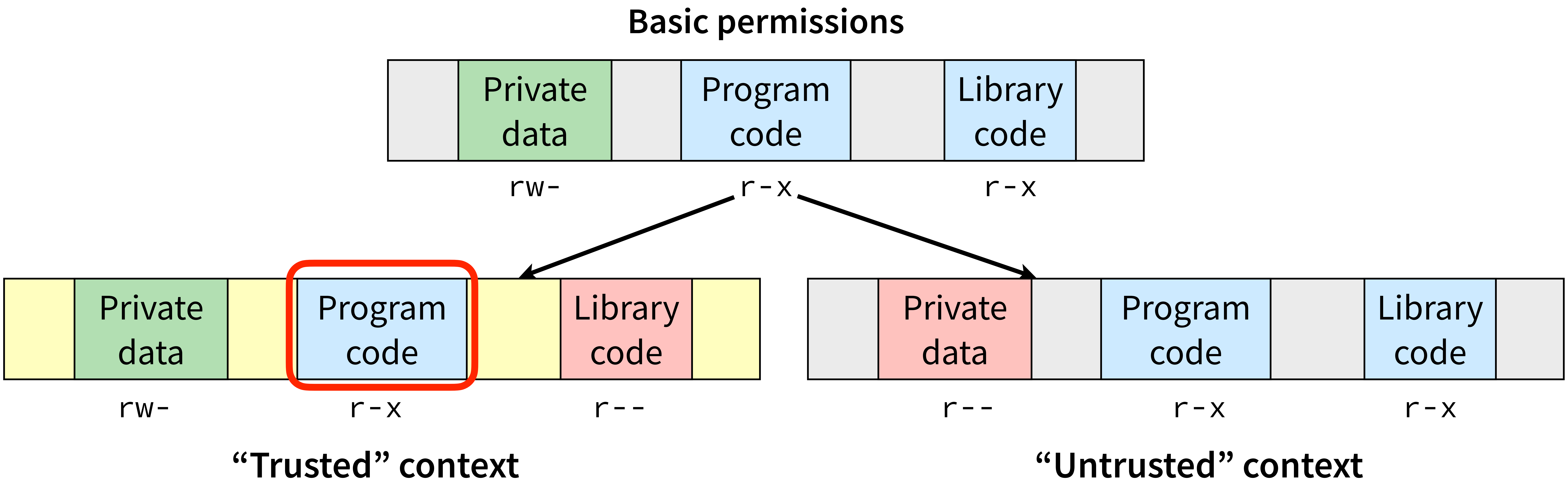
Access control per-object,
per security context

Threads can switch between security contexts

Why multiple security contexts?

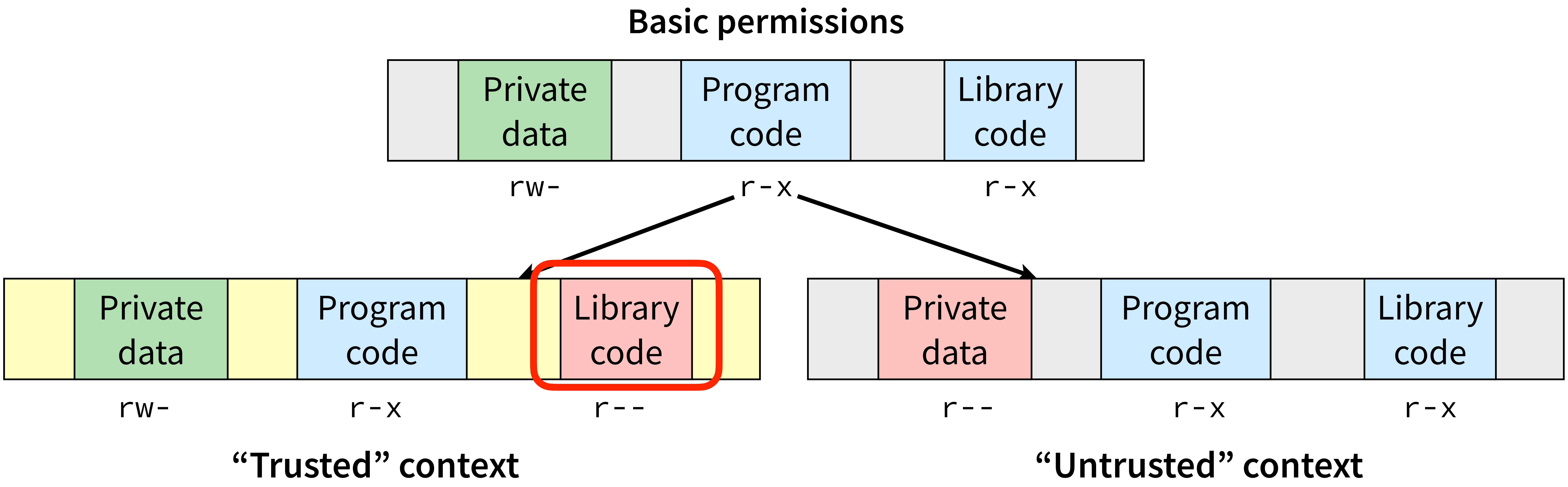


Why multiple security contexts?



Say we're running in the trusted context.

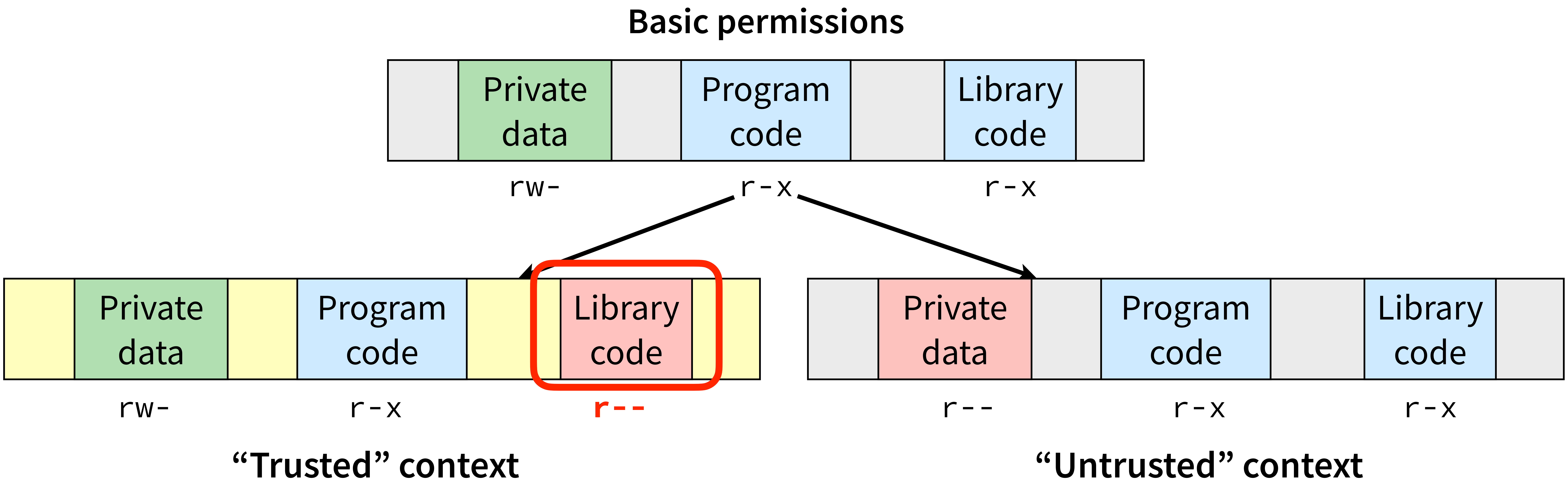
Why multiple security contexts?



Say we're running in the trusted context.

If we jump to library code, we'll cause a trap.
The kernel will then jump the thread over to the untrusted context.

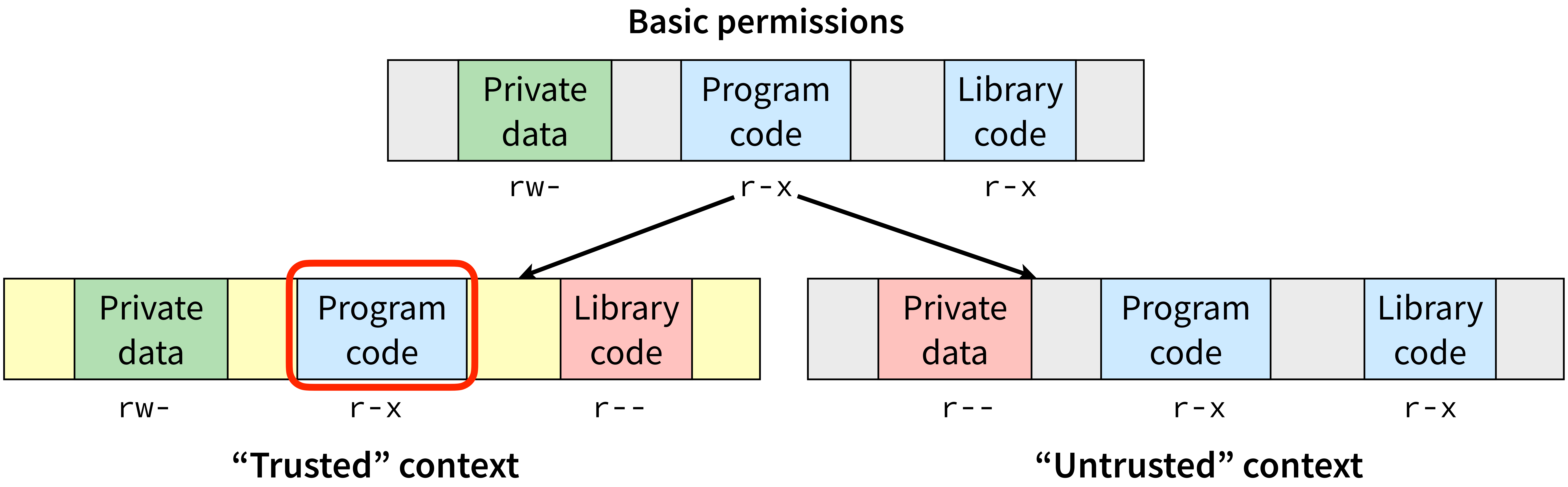
Why multiple security contexts?



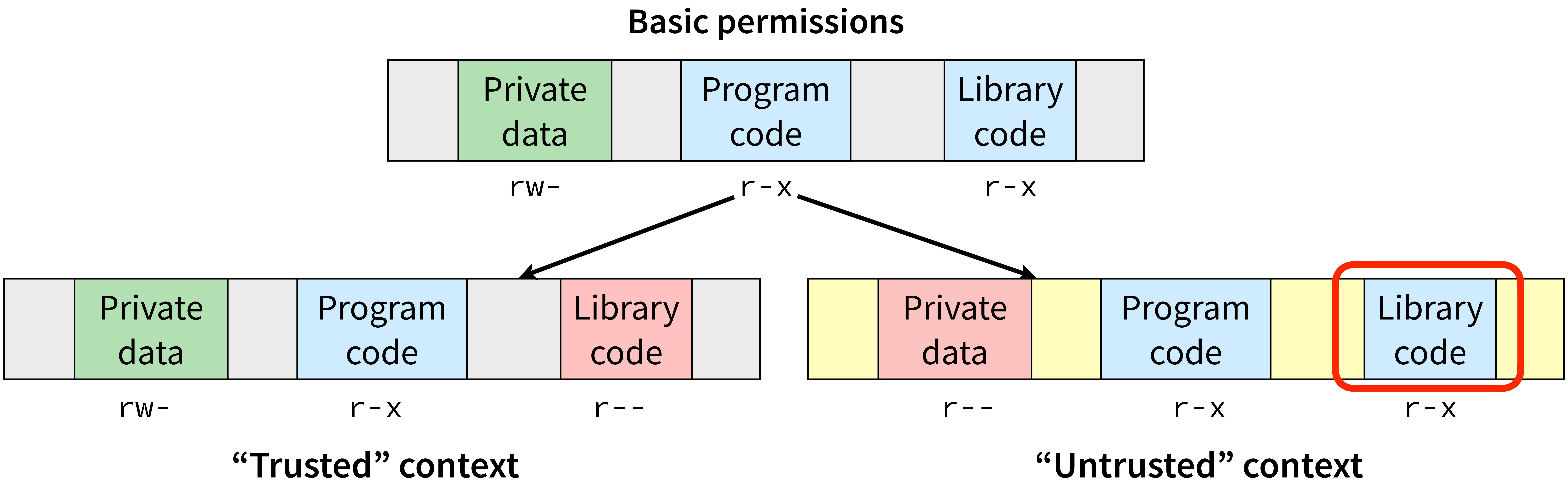
Say we're running in the trusted context.

If we jump to library code, we'll cause a trap.
The kernel will then jump the thread over to the untrusted context.

Example: trusted vs. untrusted contexts



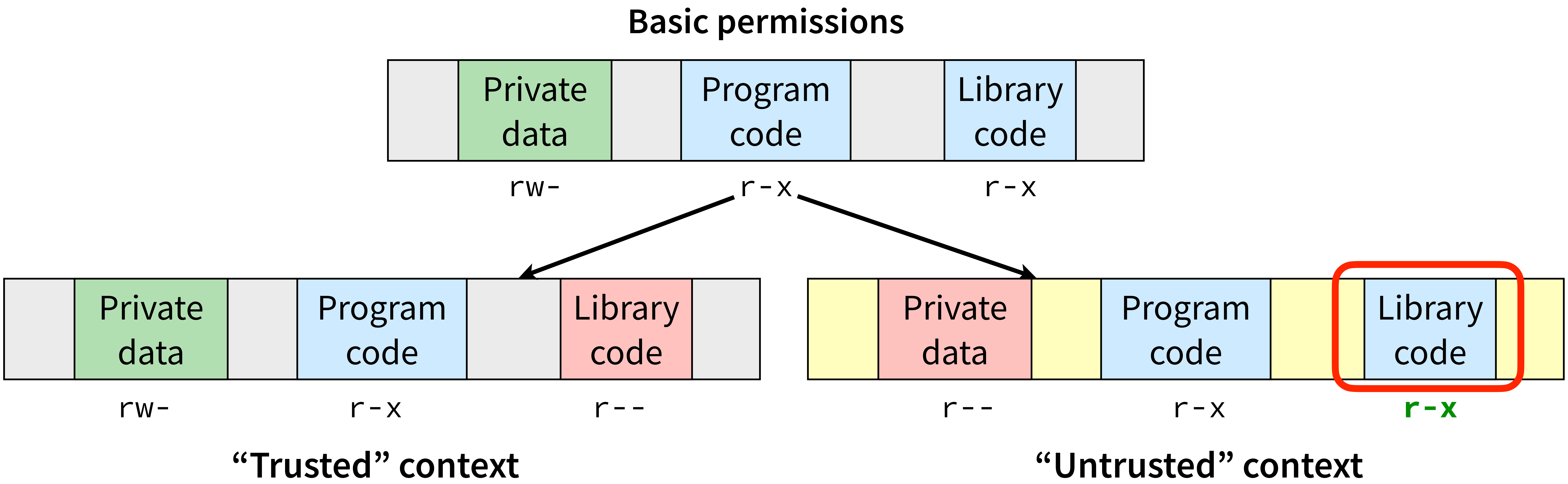
Example: trusted vs. untrusted contexts



Now, in the untrusted context, we cannot access the private data.

If we jump back to program code, and access private data, we'll get a trap.
...and switch back to "trusted".

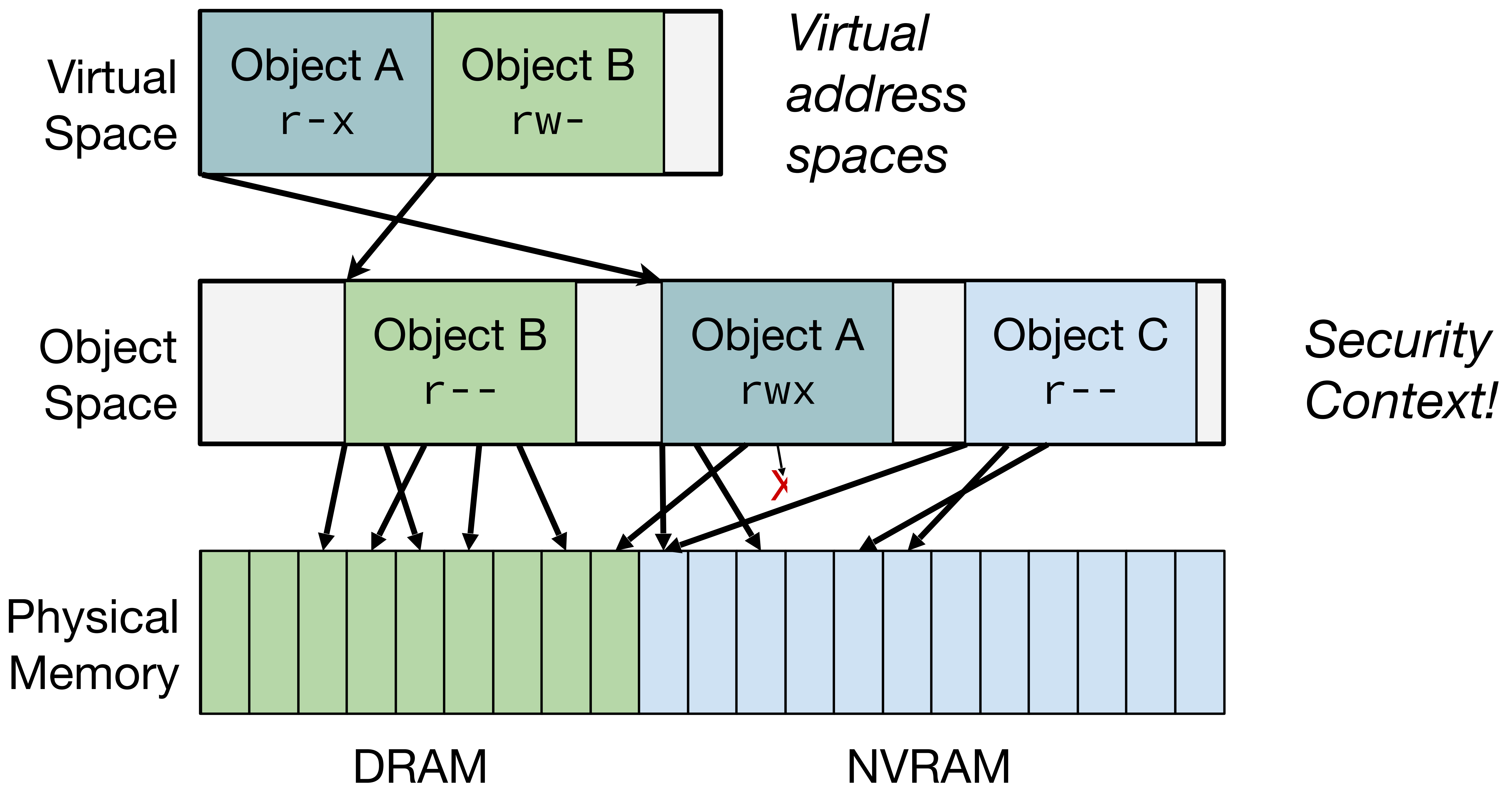
Example: trusted vs. untrusted contexts



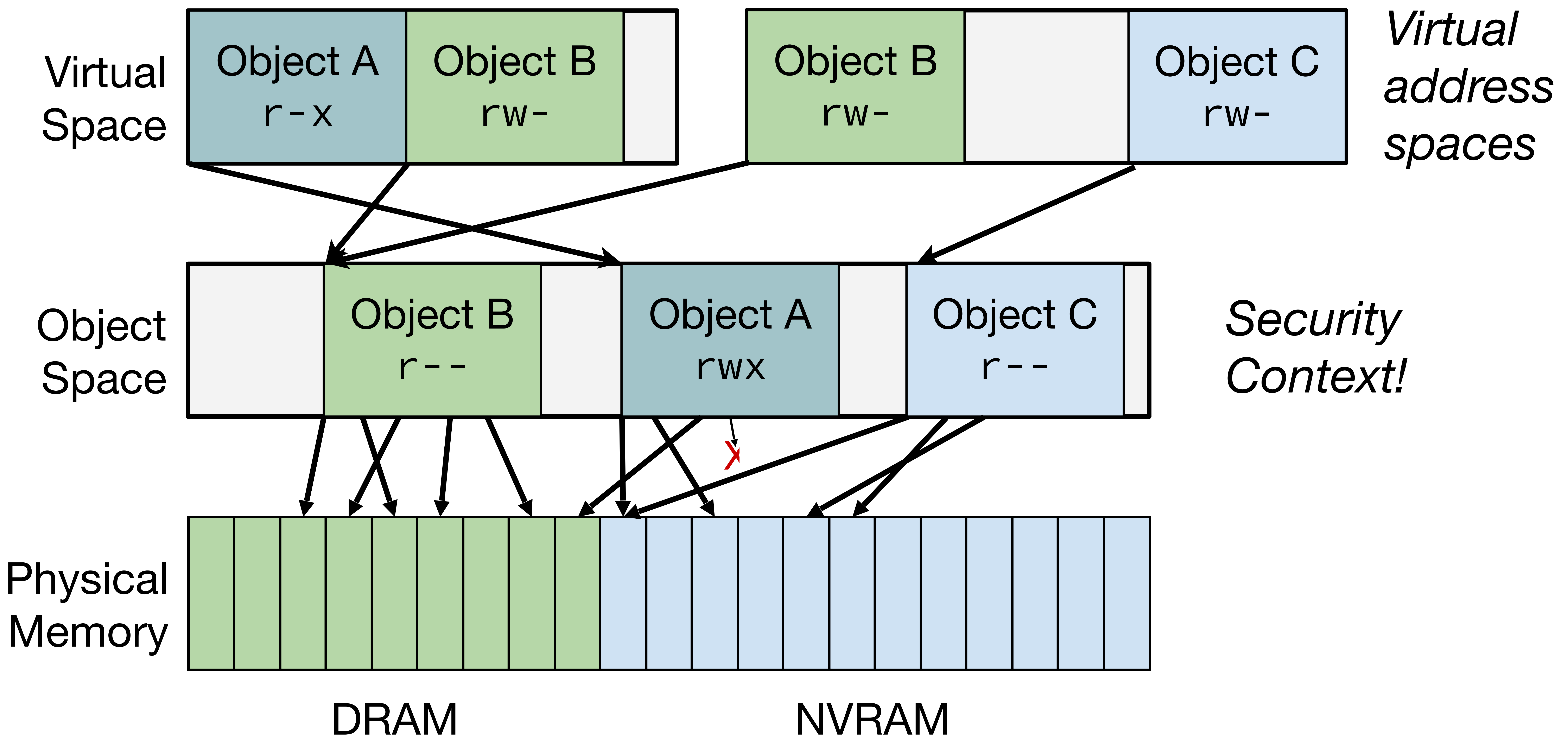
Now, in the untrusted context, we cannot access the private data.

If we jump back to program code, and access private data, we'll get a trap.
...and switch back to "trusted".

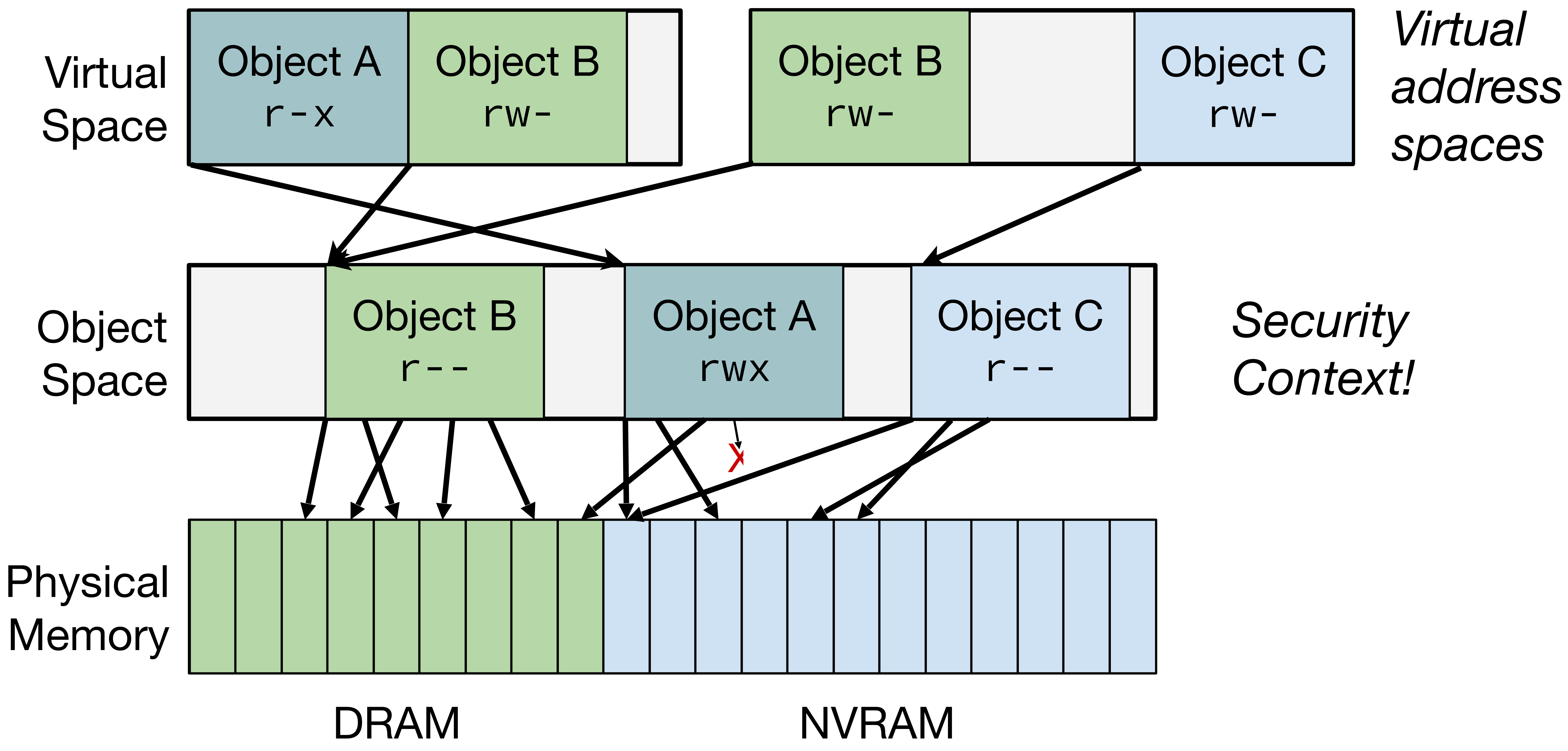
Implementing views with multi-level mapping



Implementing views with multi-level mapping



Implementing views with multi-level mapping



n+m page tables!
(instead of n×m)

Managing security in Twizzler

- **Users responsible for**
  - Setting up security policies
- **Kernel responsible for**
  - ***Validating*** security policies
  - Programming MMU to enforce policies
- **How can we make this a secure arrangement?**

User	√		
Kernel		√	√

Treat security policies as objects

- Security policies are contained in objects
- Access to objects controlled by security policies

➡ Access to security policies controlled by security policies!

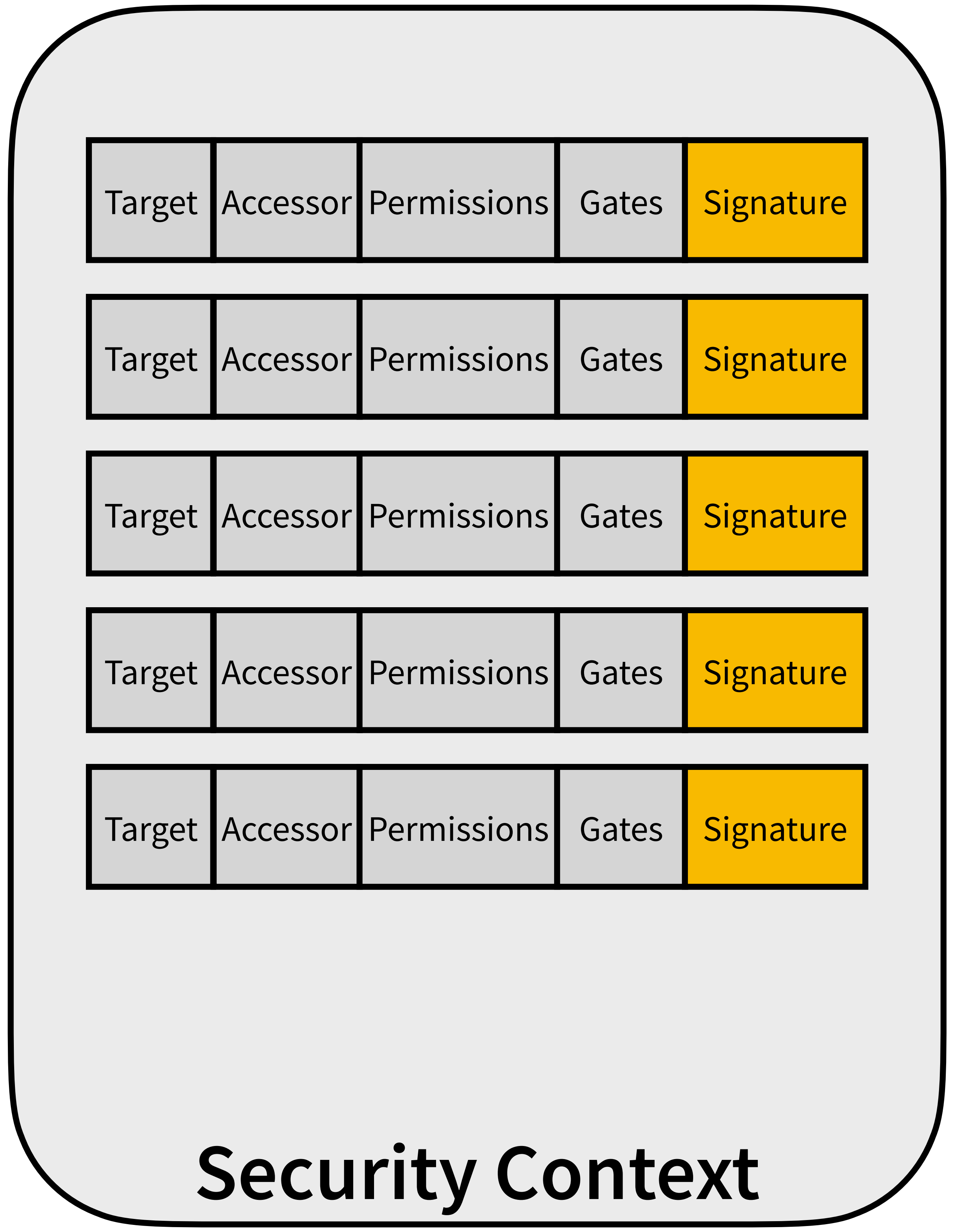
Security policies encoded in capabilities

Twizzler capability

Target	Accessor	Permissions	Gates	Signature

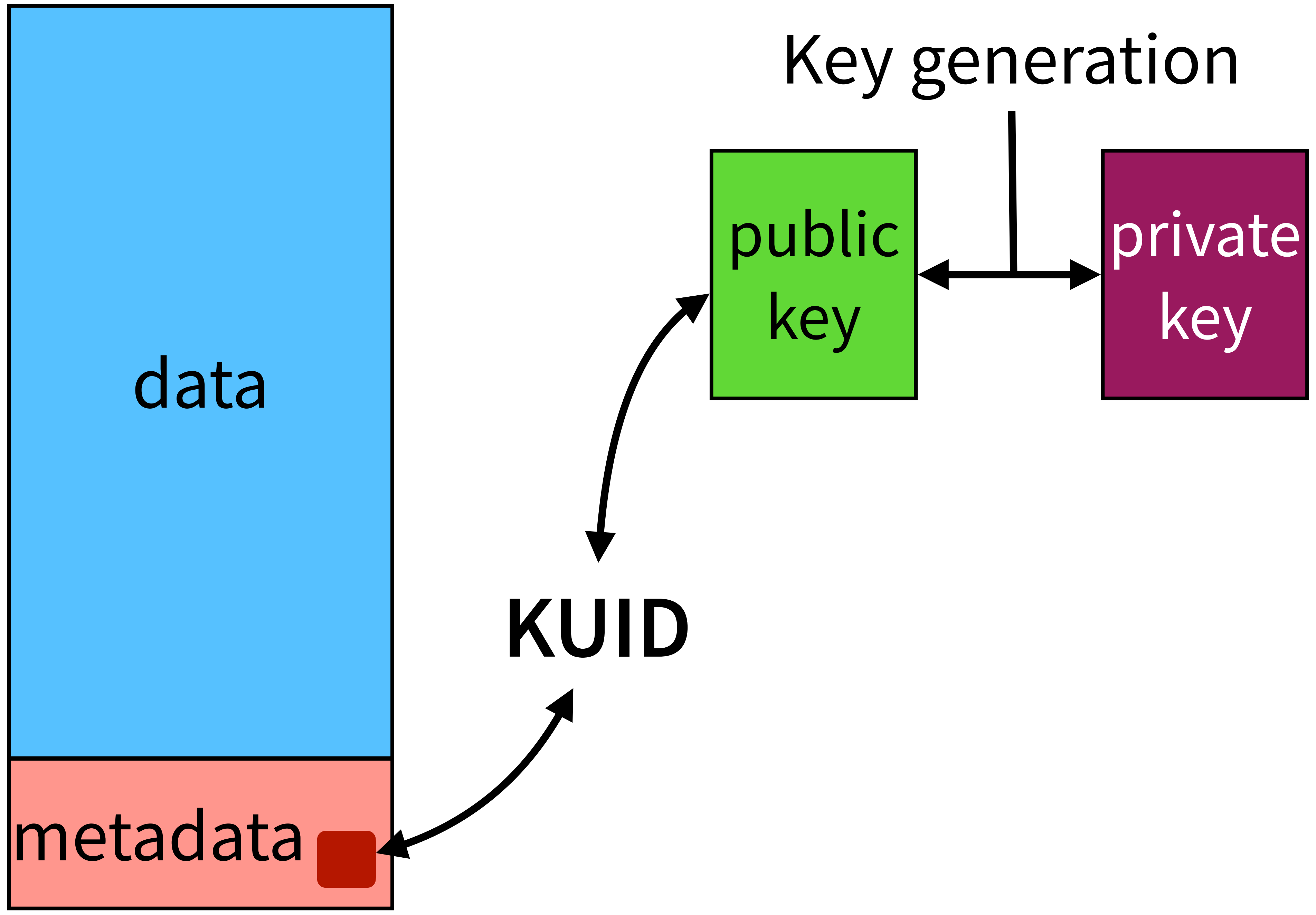
Security contexts in Twizzler

- A security context is an object containing capabilities
- A user has at least one security context
- Code objects may have their own security contexts



Objects and keys

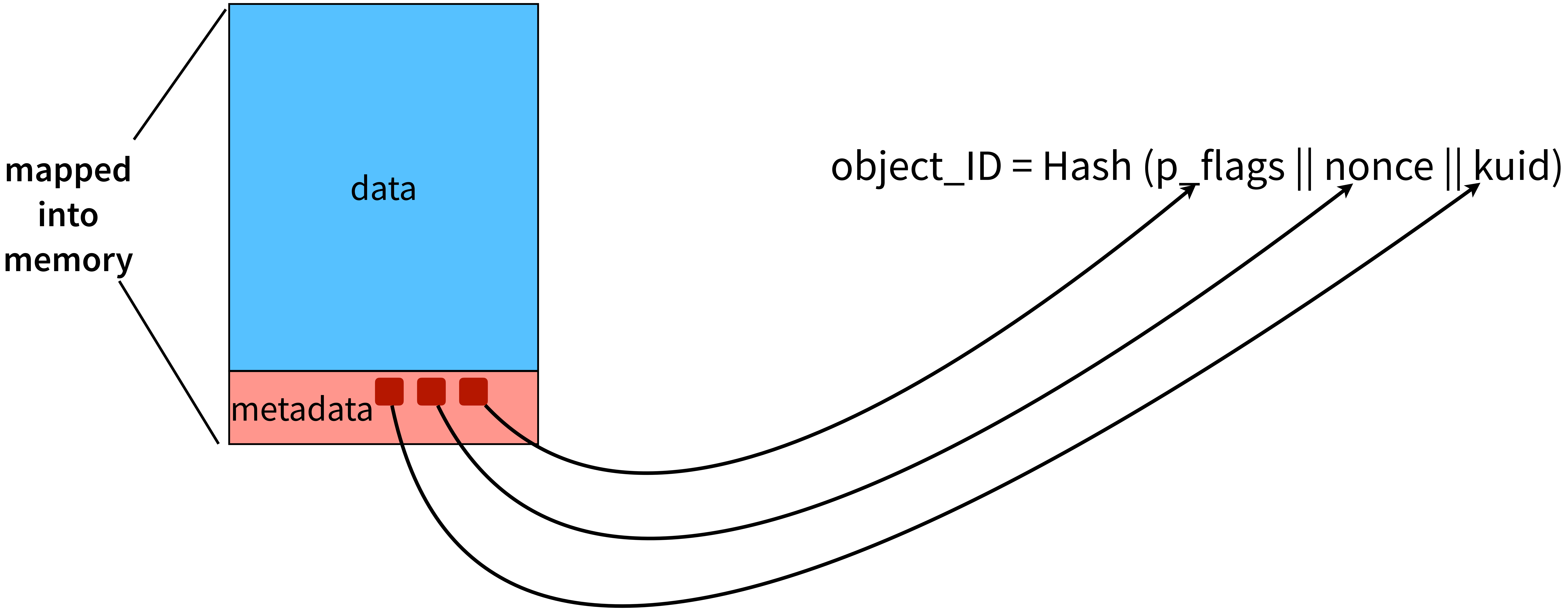
- Each object has a public-private key pair
  - Key pair need not be unique to the object
  - Example: user might have half a dozen key pairs
  - Example: “system” might have a single key pair
- Kernel can read public keys
- Capabilities signed by private keys
  - Private keys kept in objects with access control
  - Need not be stored in the clear on the system



Aren't public key ops expensive?

- **Public-key operations are indeed expensive (relatively)**
- **Mitigate the cost by**
  - Having the kernel cache results of PK operations (verifications)
  - Having "default" permissions encoded directly in the object: especially important for public code objects

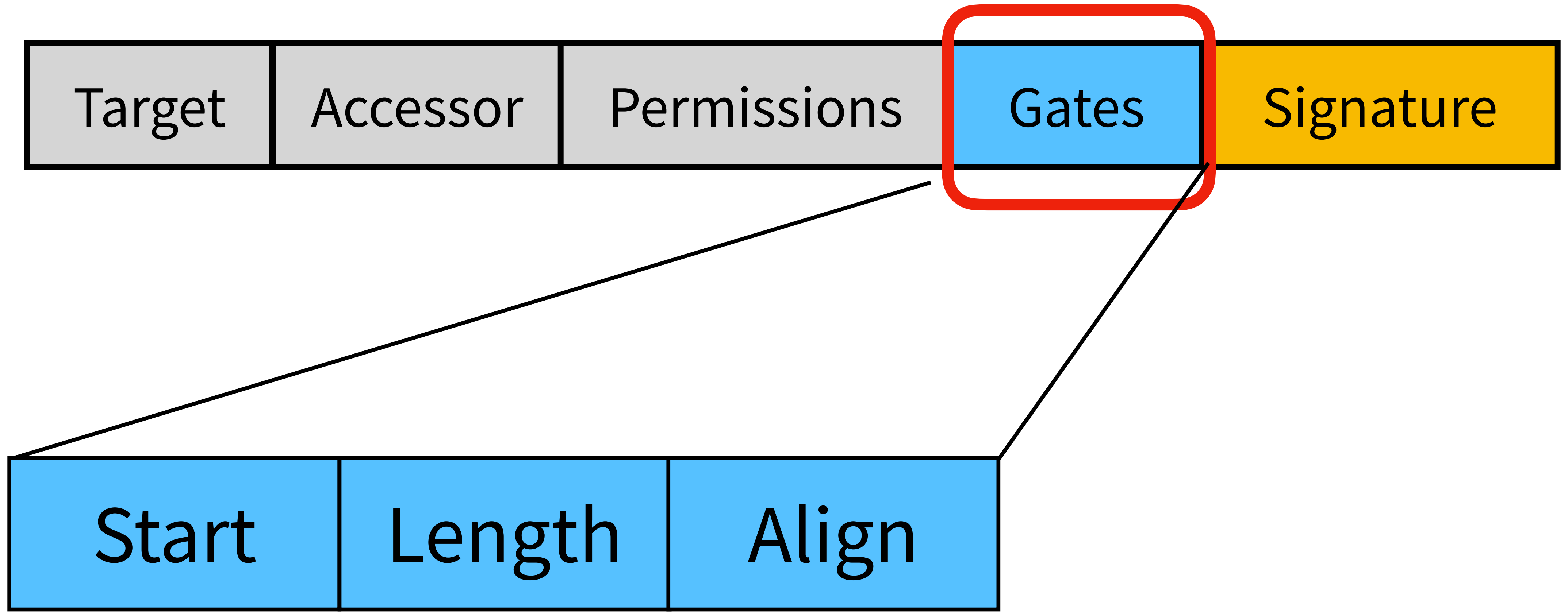
Object ID as self-signature



Use masks to limit permissions

- **Maximum permissions determined by union of**
  - Default object permissions
  - Permissions granted by a specific (signed) capability
- **Permissions *limited* by masks**
  - Security context can limit permissions to objects it could otherwise access: useful in preventing accidental (or malicious) accesses
  - This can be done per-object, or globally for a security context
    - Code library that can only read most objects
    - Exceptions for stack and perhaps heap

Gates



- **Object-level permissions OK for read & write**
- **Execute is different: limit "access points"**
- **<u>Gates</u> provide this limitation**
  - Specify start, length, alignment
  - Jump *into* object must meet these criteria
- **Use a *trampoline* for return from call to a different object**

Implications for security

CENTER FOR RESEARCH IN STORAGE SYSTEMS

- **Security can be specified by *users* without kernel intervention**
  - Capabilities are protected by cryptographic signatures
  - Private keys need not be accessible to the kernel
- **Kernel can *validate* signatures using public keys**
  - Public key identifiers generated by hashing as well (standard technique)
  - No need to even know *who* signed a capability: don't need to be local!
- **Users can ask for any privileges they want**
  - Kernel only grants those that it can verify using capabilities
  - Kernel programs the MMU to enforce these permissions

Further security issues

CENTER FOR RESEARCH IN STORAGE SYSTEMS

- **Delegation**
  - Principal assigns a capability to another principal that may not already have access
  - Assignment can limit further delegation
  - Assignment authenticated by signing with private key
- **Revocation**
  - Capabilities may be time-limited
  - Revocation by expiring capabilities

Ongoing research: distributed PM systems

- **Twizzler-style access works very well in distributed systems**
  - GUIDs are 128-bit, easily expandable to 256 bit without larger persistent pointers
  - Access to objects is transparent to object physical location
    - Cache object in local memory?
    - Send accesses to remote memory?
- **Security is scalable as well**
  - Capabilities can be verified by *any* kernel with the necessary public keys
  - Currently no way to ***guarantee*** that remote kernel is trustworthy
    - This is a very difficult problem
    - Straightforward to reject writes to local objects without accompanying capabilities

Conclusions

- **Persistent memory requires direct access with minimal OS involvement**
  - Accesses must go directly to/from PM
  - Kernel sets up the MMU and stays out of the way
- **Programming model must allow easy sharing in a scalable system**
  - Security is an important part of that
- **OS must enforce user-specified security**
  - Minimal *implicit* trust of security policies: rely on public-key encryption
  - Maximal flexibility for user-level specification of policies

Remember…

SSDs only reached their true potential
when we stopped treating them like fast disks
and optimized for how they work.

Persistent memory will only reach its true potential
when we treat it as a single-level persistent store that
supports direct byte-level access for computation and storage.

Questions?

CENTER FOR RESEARCH IN STORAGE SYSTEMS

Students

Allen Aboytes
Daniel Bittman
Barbara Moretto Dama
Vishal Shrivastav
Michael Usher

Faculty

Peter Alvaro
Darrell D. E. Long
Ethan L. Miller
Robert Soule

Industry

Pankaj Mehra

https://twizzler.io

Research sponsors: