Karlsruhe Institute of Technology

Microkernel Construction
I.9 – Security

Lecture Summer Term 2017
Wednesday 15:45-17:15 R 131, 50.34 (INFO)

Jens Kehne | Marius Hillenbrand
Operating Systems Group, Department of Computer Science

Is your system secure?

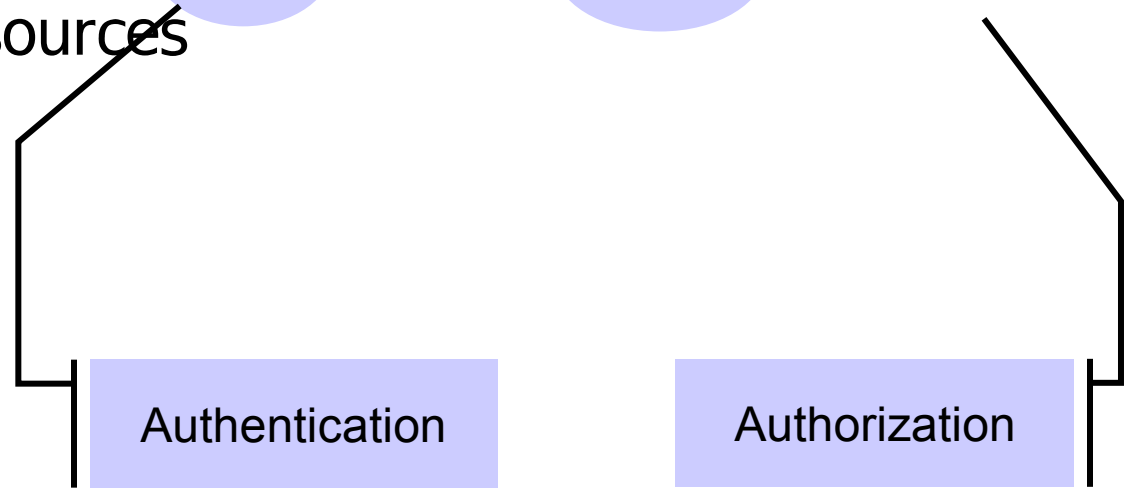
Security: A condition that results from the establishment and maintenance of protective measures that ensure a state of inviolability from hostile acts or influences. [Wikipedia]

Security Defined by Policy

- Examples
  - All users have access to all objects
  - Physical access to servers is forbidden
  - Users only have access to their own files
  - Users have access to their own files, group access files, and public files (UNIX)

Security Policy

- Specifies who has what type of access to which resources



All Access is via IPC

- What microkernel mechanisms are needed for security?
  - How do we authenticate?
  - How do we perform authorization?
  - How do we implement arbitrary security policies?
  - How do we enforce arbitrary security policies?

Authentication

- Unforgeable endpoint identifiers
  - Thread ID of sender returned by kernel
  - Capabilities generated by kernel
  - Thread identifiers can be mapped to
    - Tasks
    - Users
    - Groups
    - Machines
    - Domains
  - Authentication is outside the microkernel – any policy can be implemented

Authorization

- Servers implement objects; clients access objects via IPC
- Servers receive unforgeable client identities from the IPC mechanism
  - Servers can implement arbitrary access control policy
- No special mechanisms needed in the microkernel

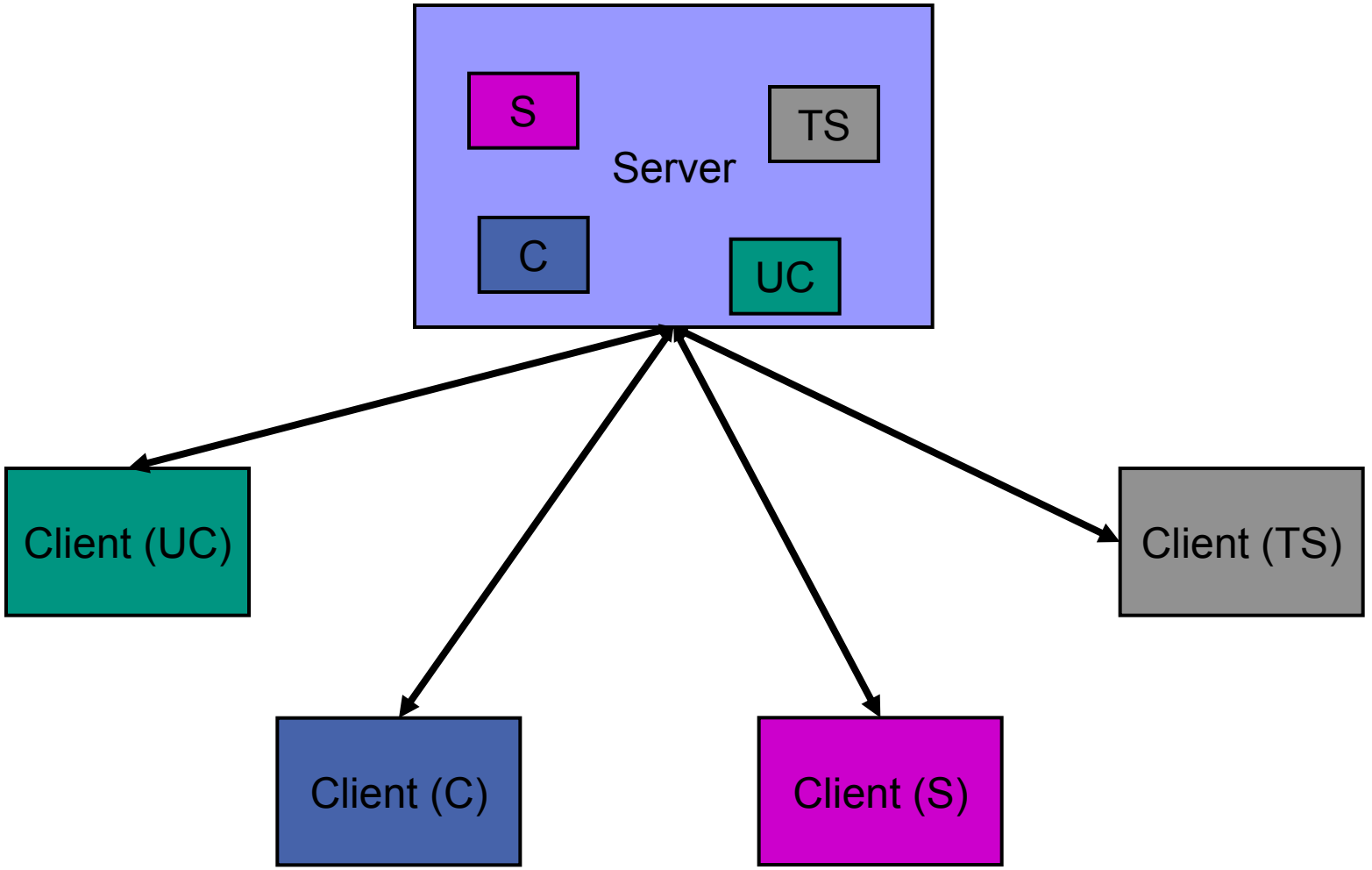
Is this really true???

Example Policy
Multi Level Security (MLS) – Confidentiality

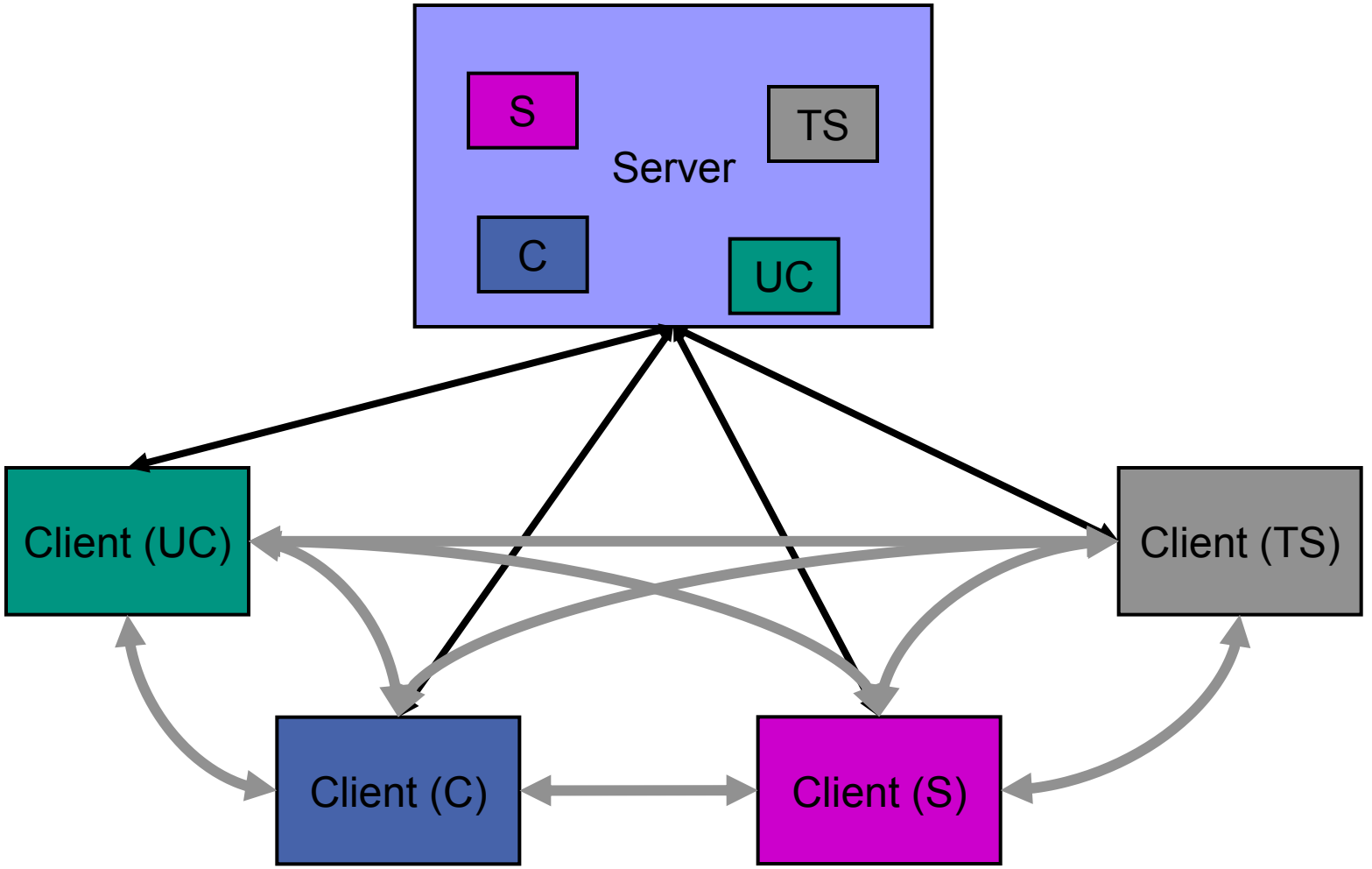
- Assign security levels to objects
  - Top Secret, Secret, Classified, Unclassified
    - TS > S > C > UC
- Assign security levels to subjects (users)
  - Top Secret, Secret, Classified, Unclassified

- Subject S can read object O iff
  - level (S) ≥ level (O)
- Subject S can write (append to) object O iff
  - level (S) ≤ level (O)

Example Policy
Multi Level Security (MLS) – Confidentiality



Problem

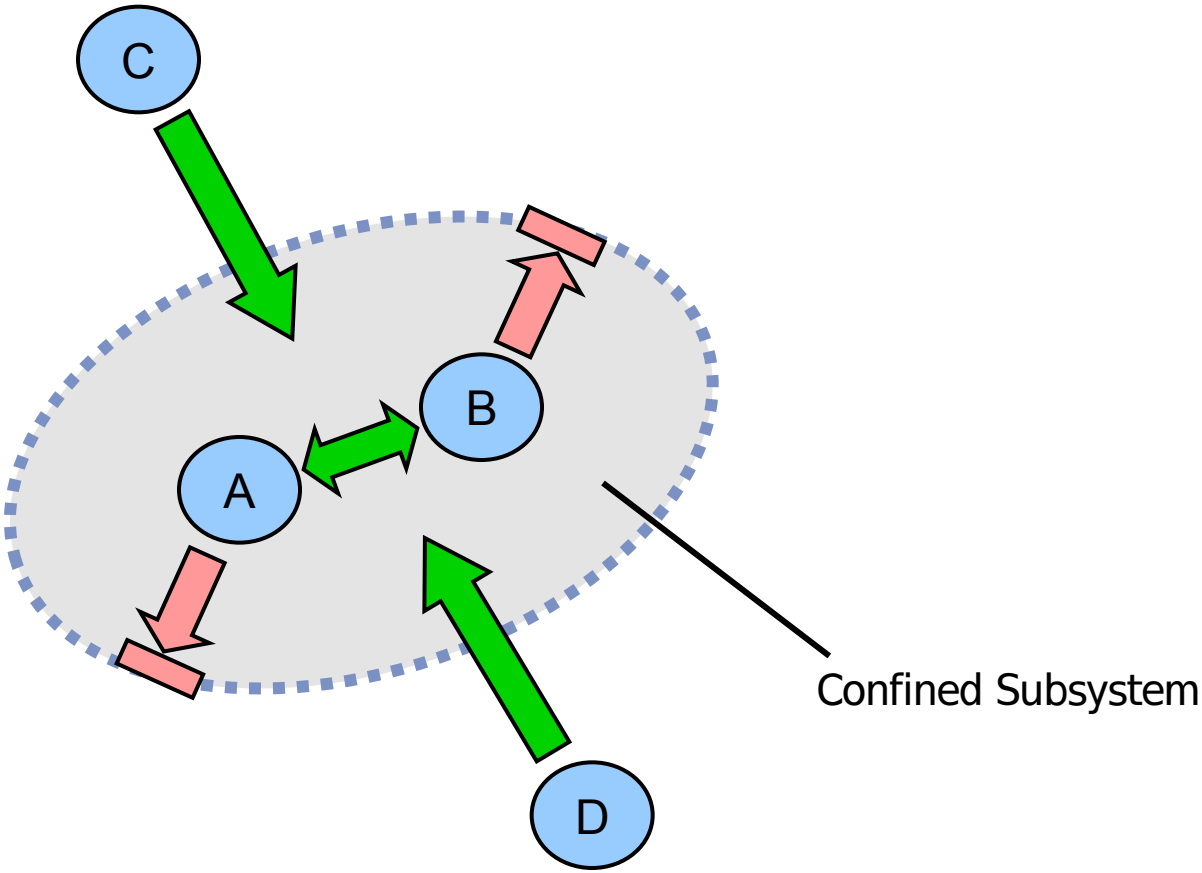


Conclusion

To control information flow we must control communication.

- We need mechanisms to not only implement a policy – we must also be able to enforce a policy
- Mechanism must be flexible enough to implement and enforce all relevant security policies

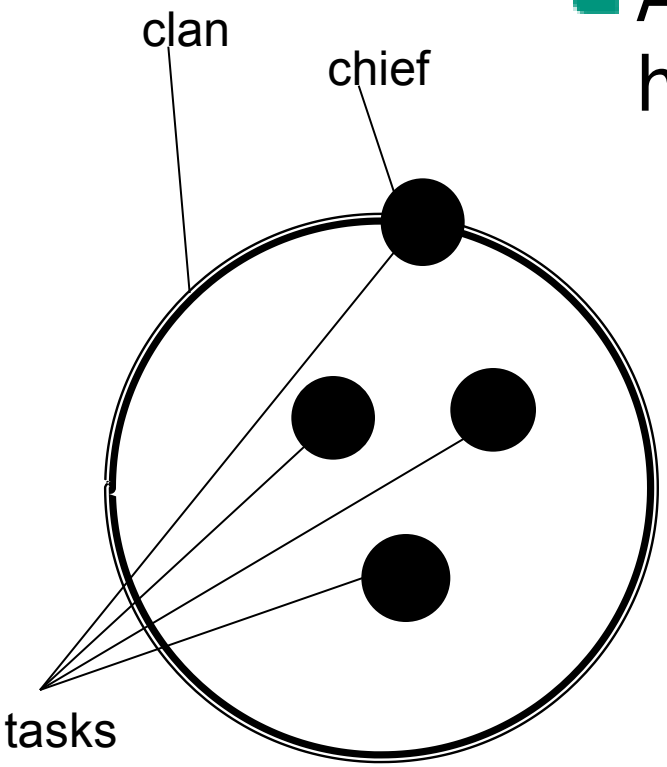
Confinement



CLANS & CHIEFS

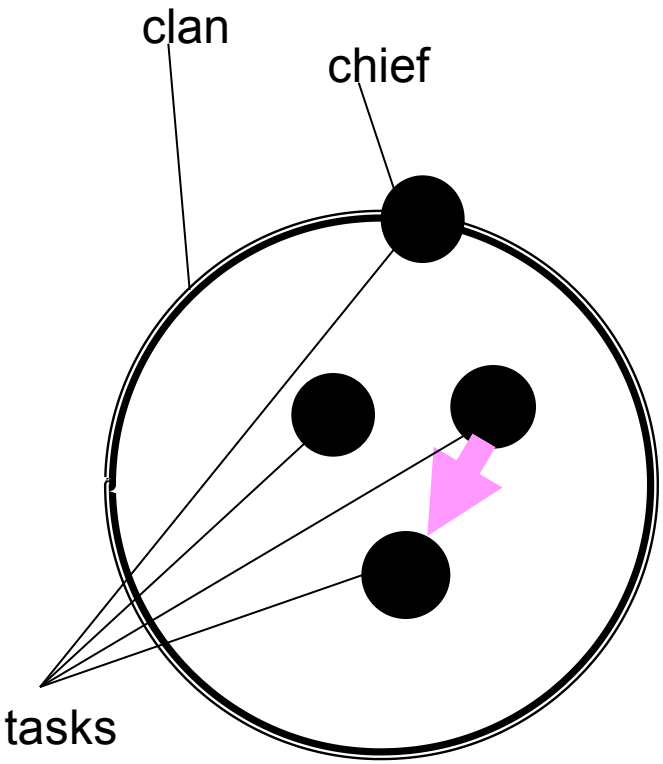
The Traditional L4 Approach

Clans & Chiefs



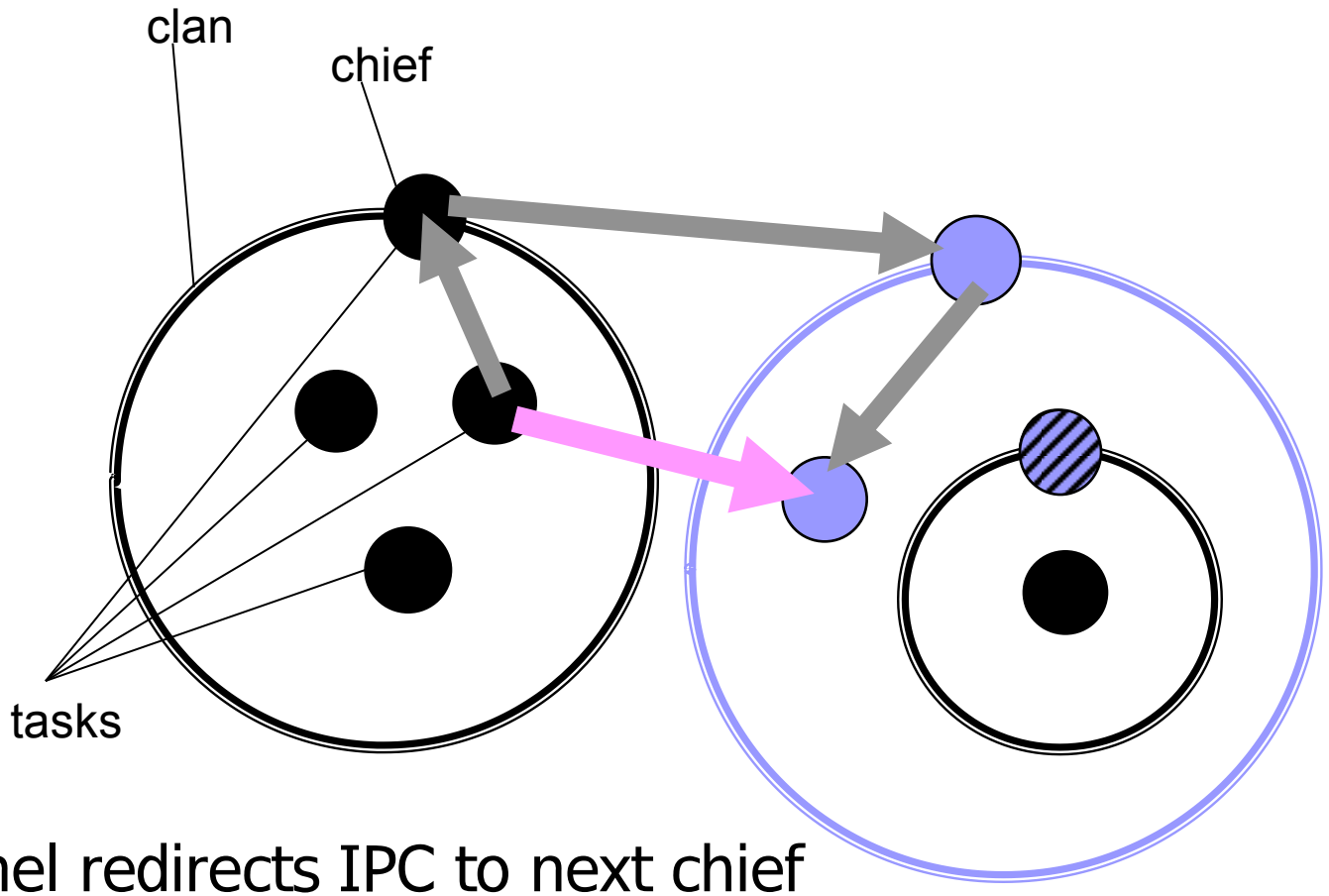
- A clan is a set of tasks headed by a chief task

Intra-Clan IPC



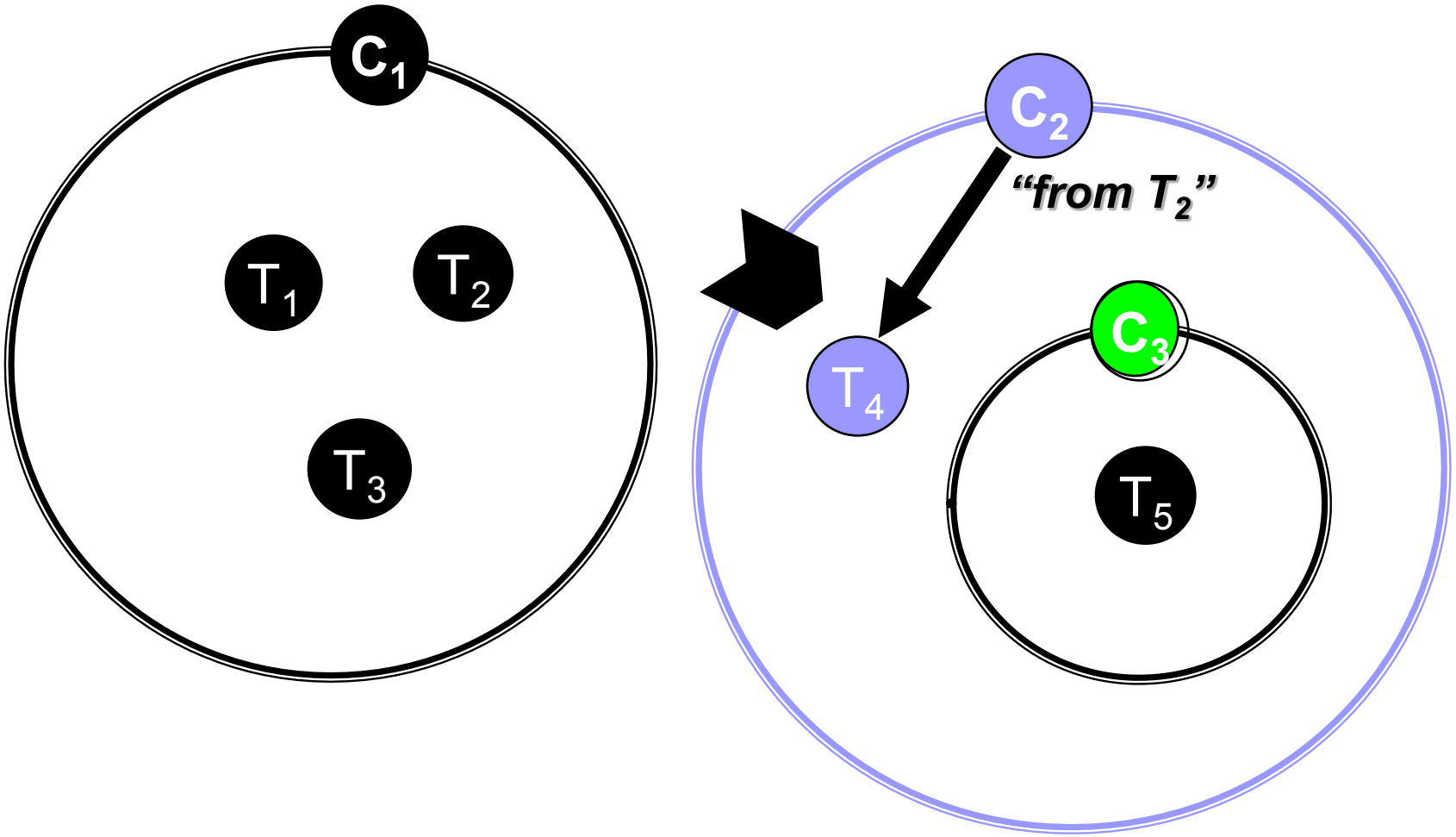
- Direct IPC by microkernel

Inter-Clan IPC

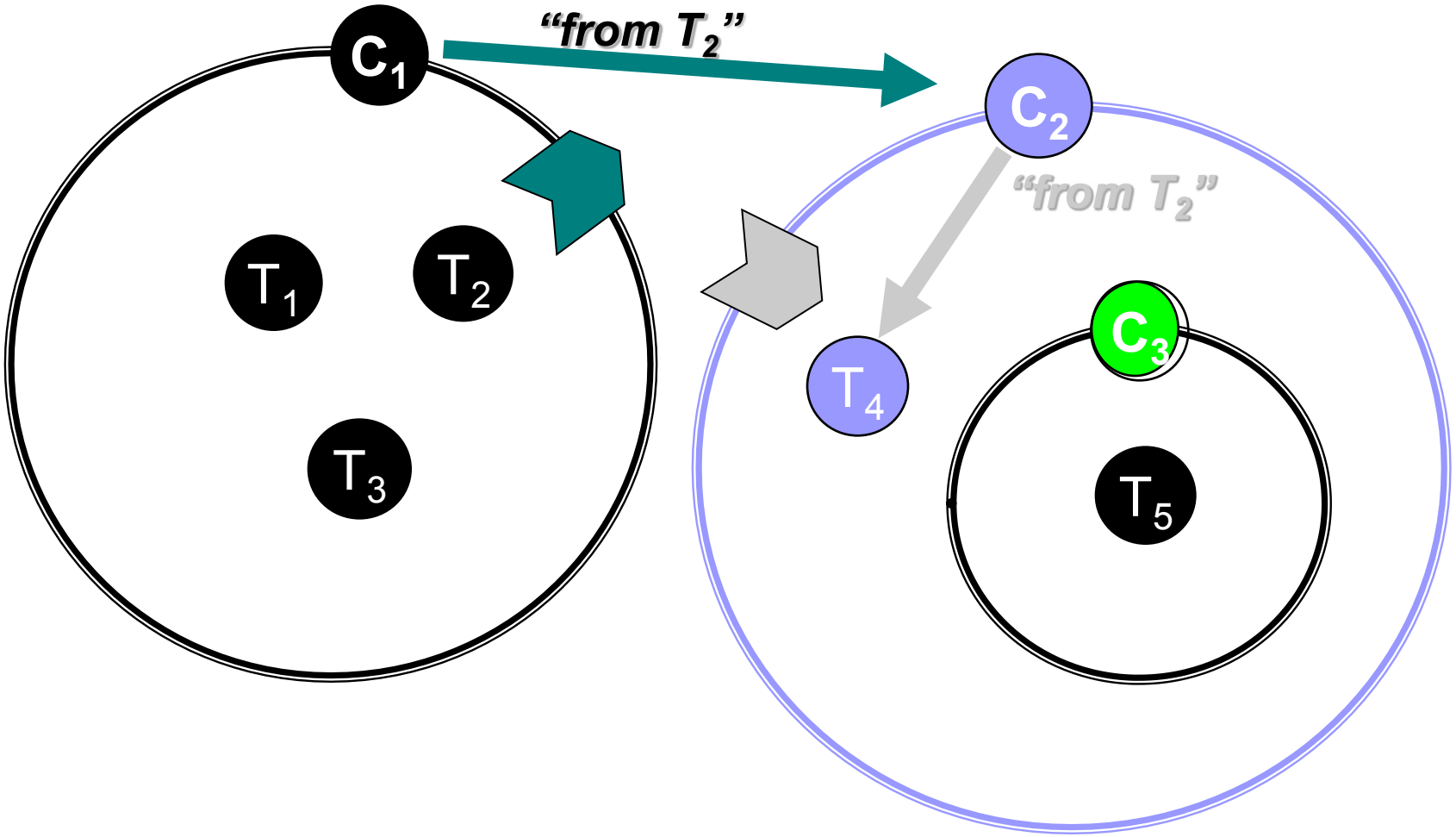


- Microkernel redirects IPC to next chief
- Chief (user task) can forward IPC or modify or ...

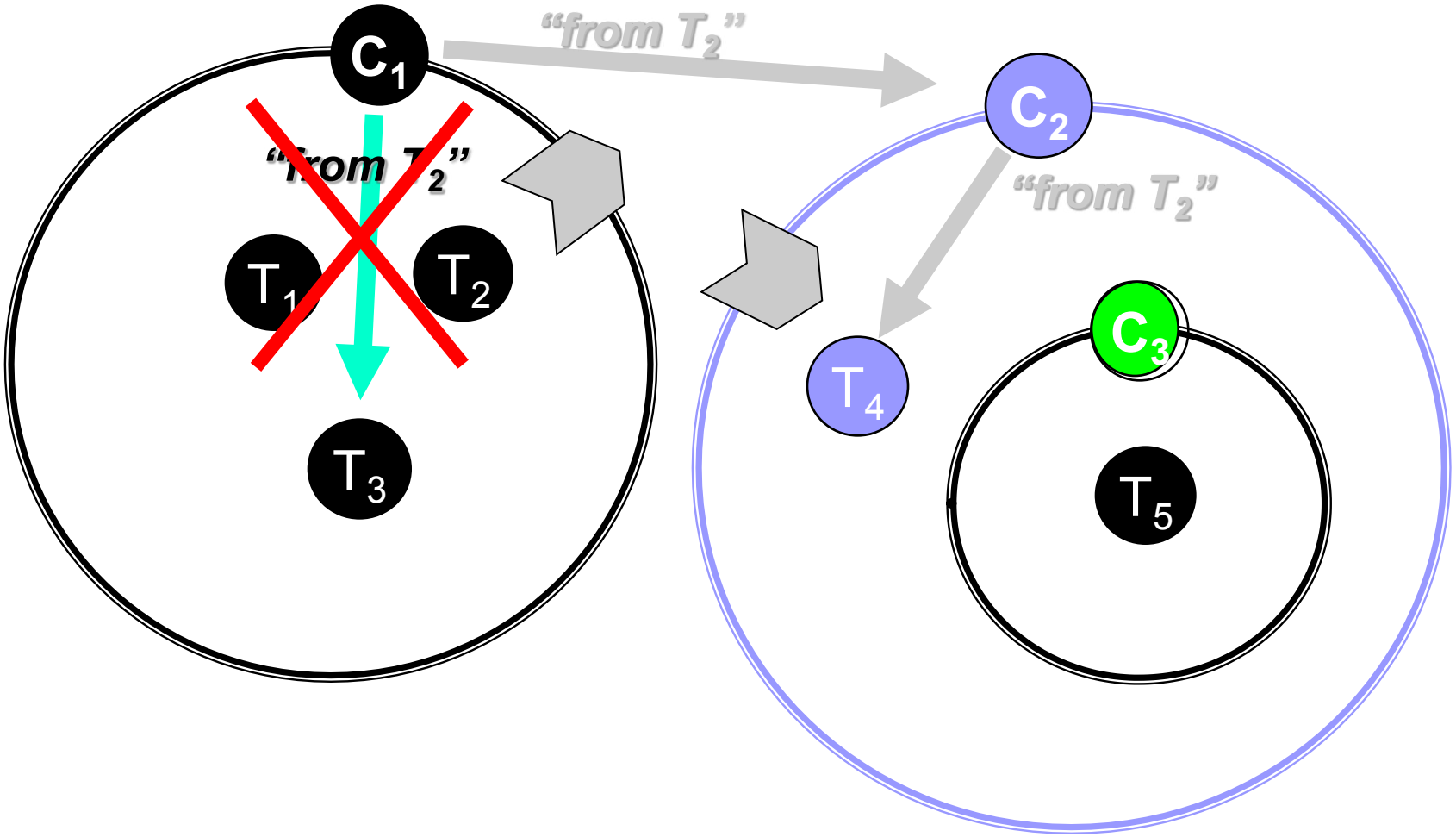
Direction-Preserving Deceiving



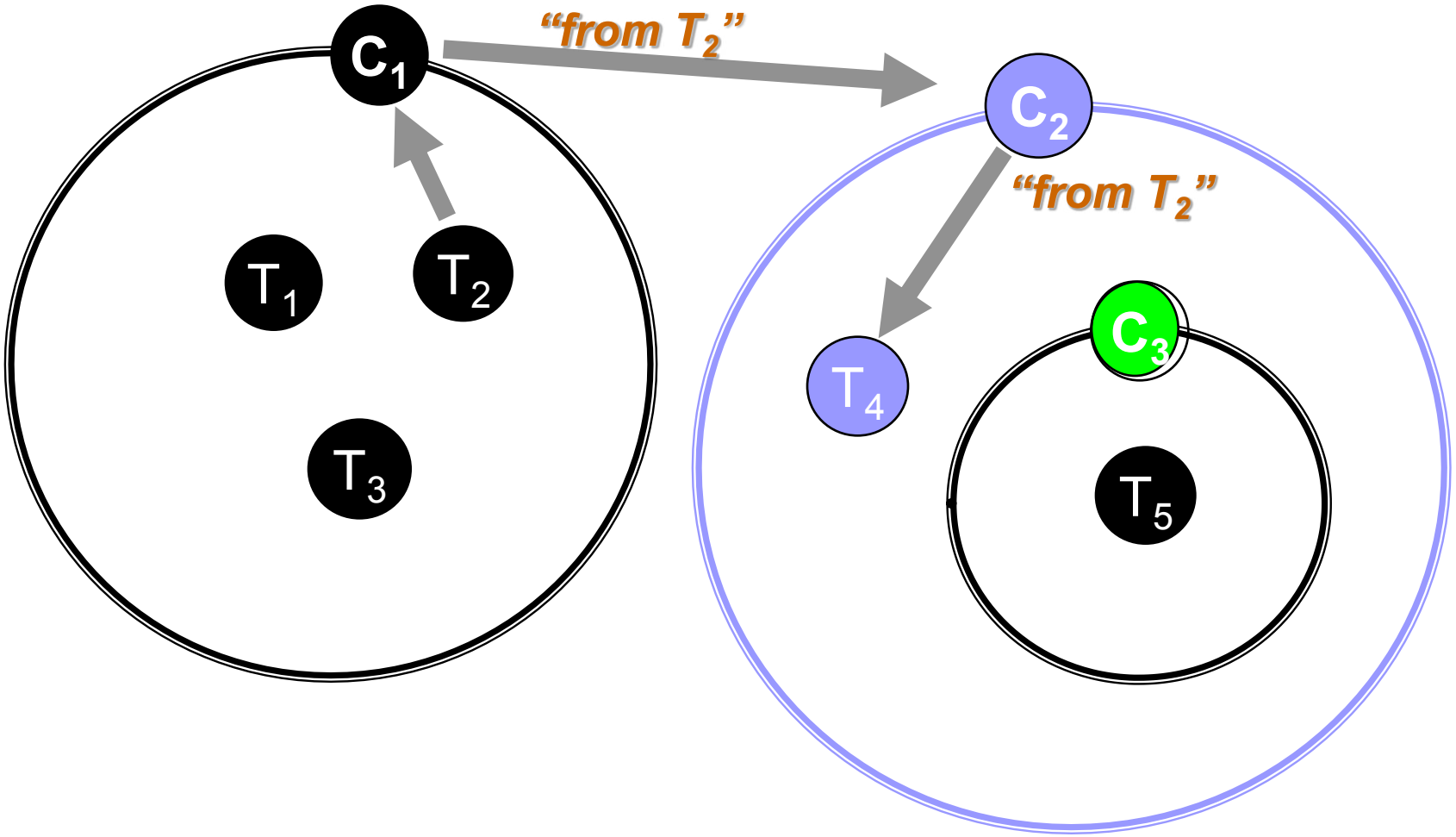
Direction-Preserving Deceiving



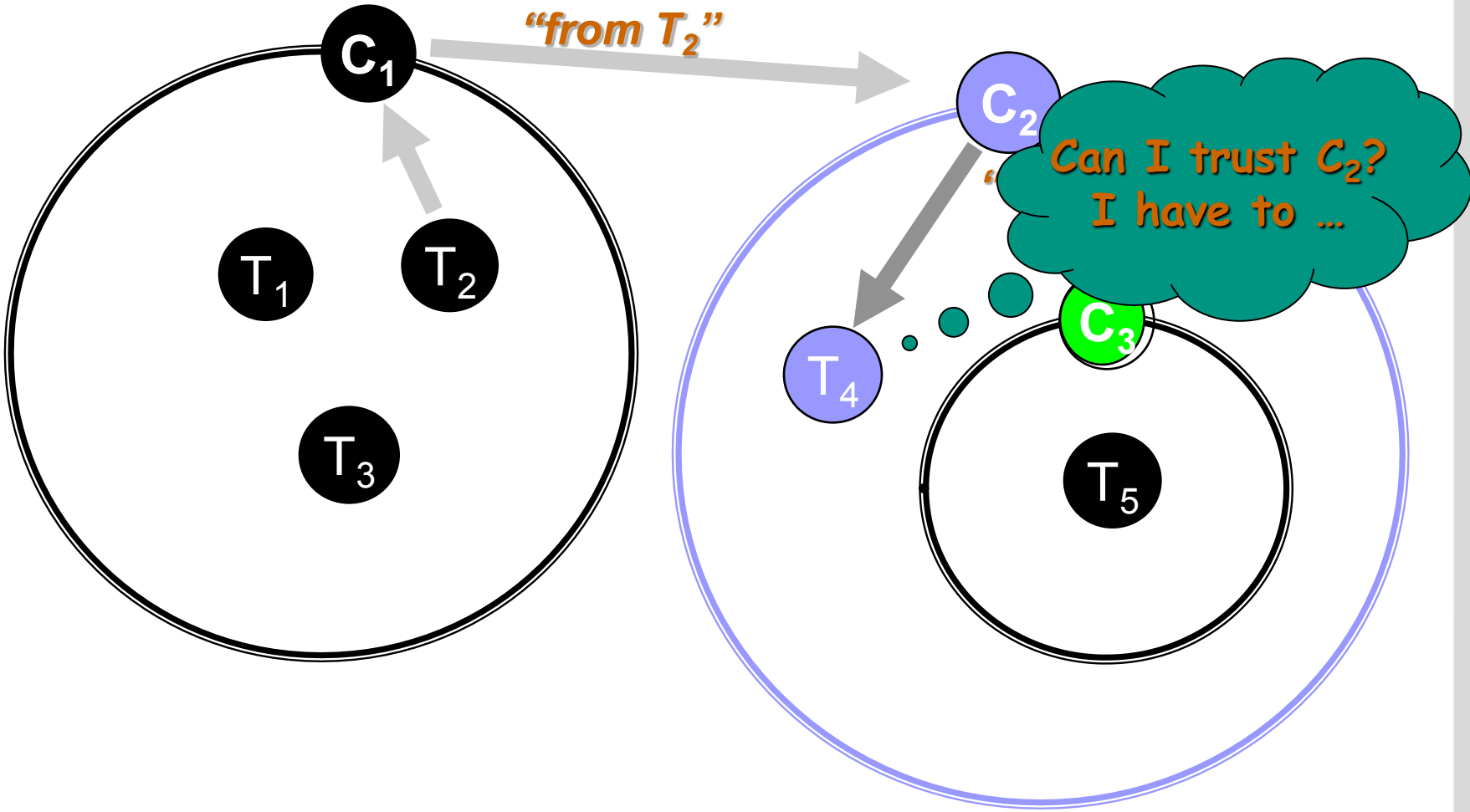
Direction-Preserving Deceiving



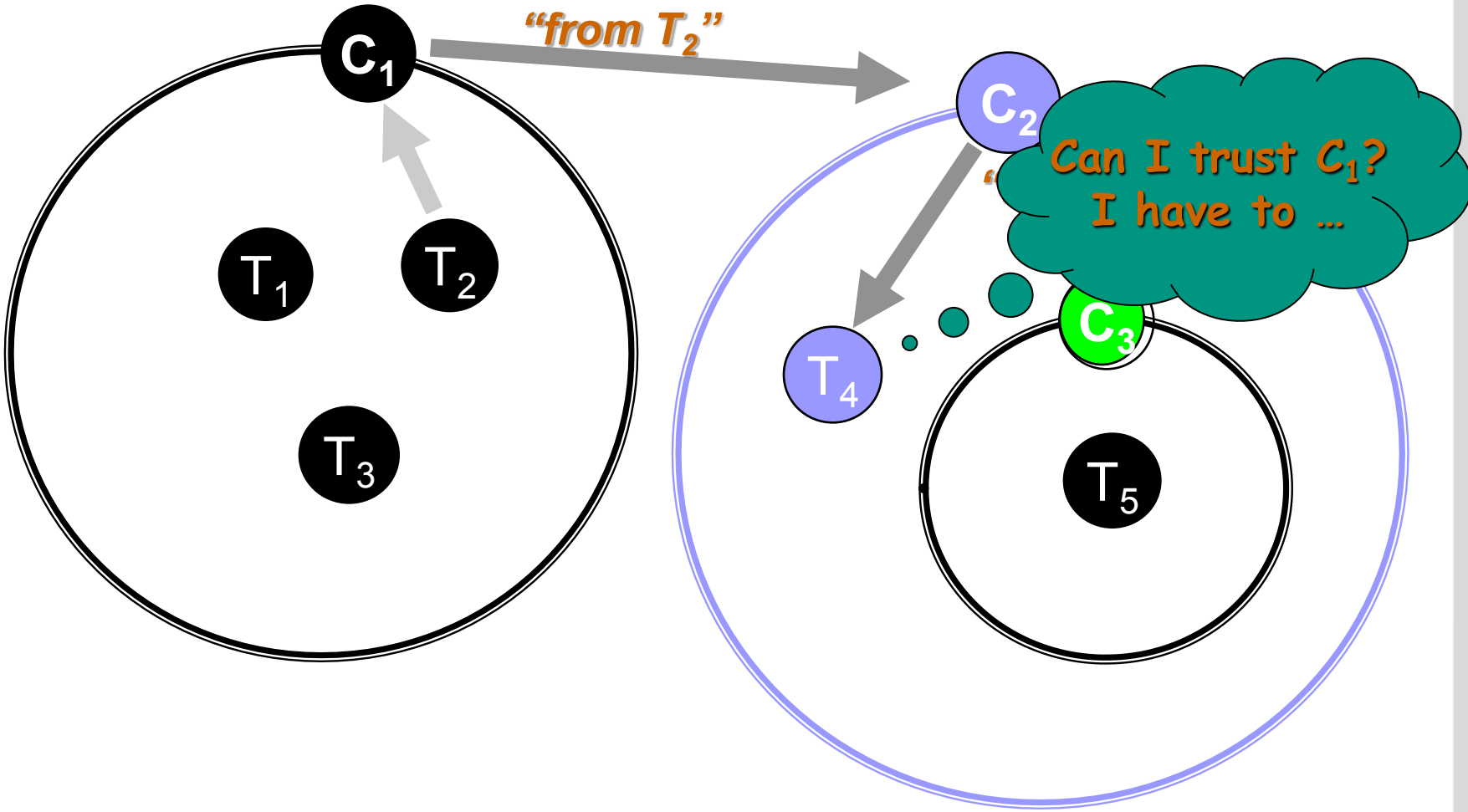
Direction-Preserving Deceiving



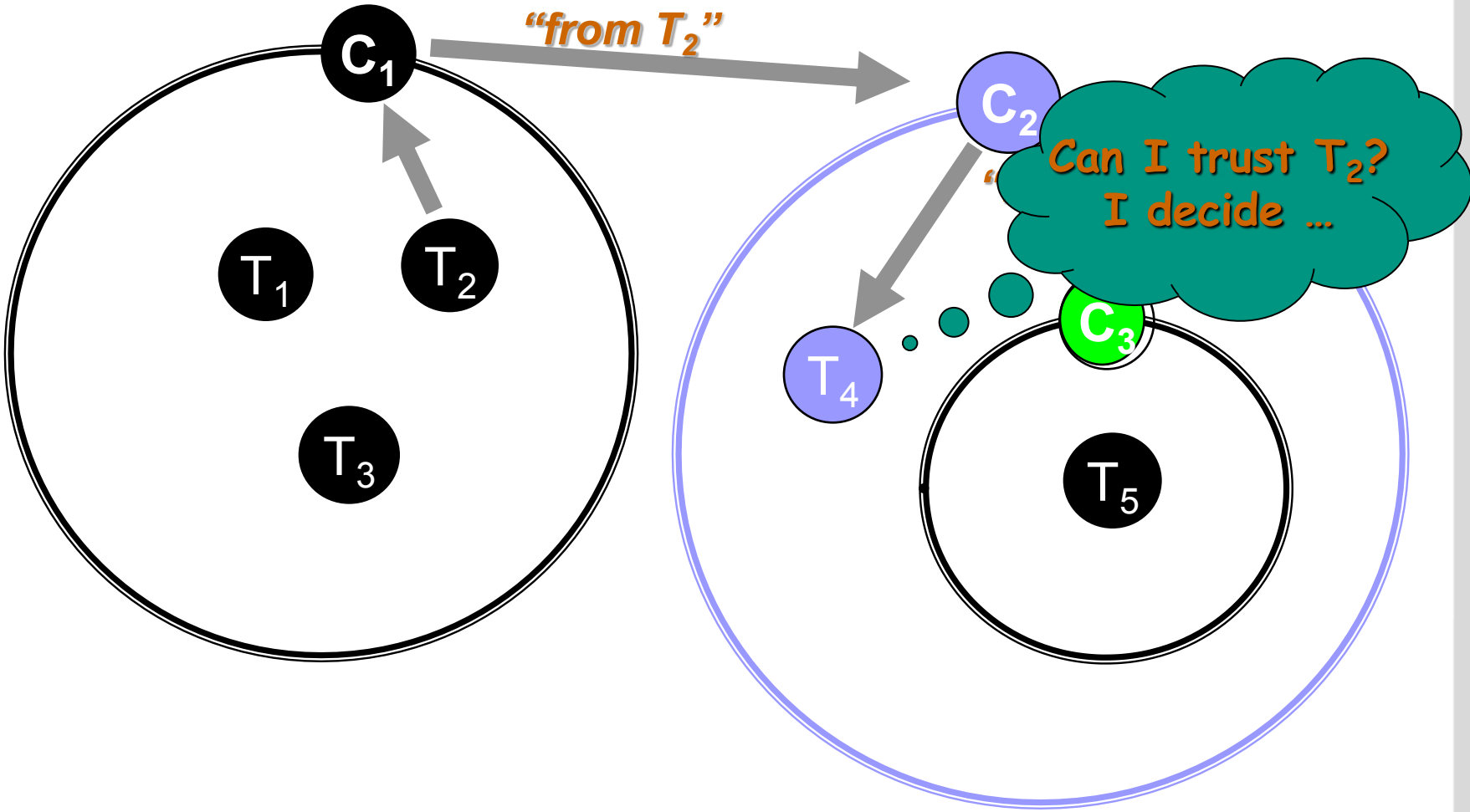
Direction-Preserving Deceiving



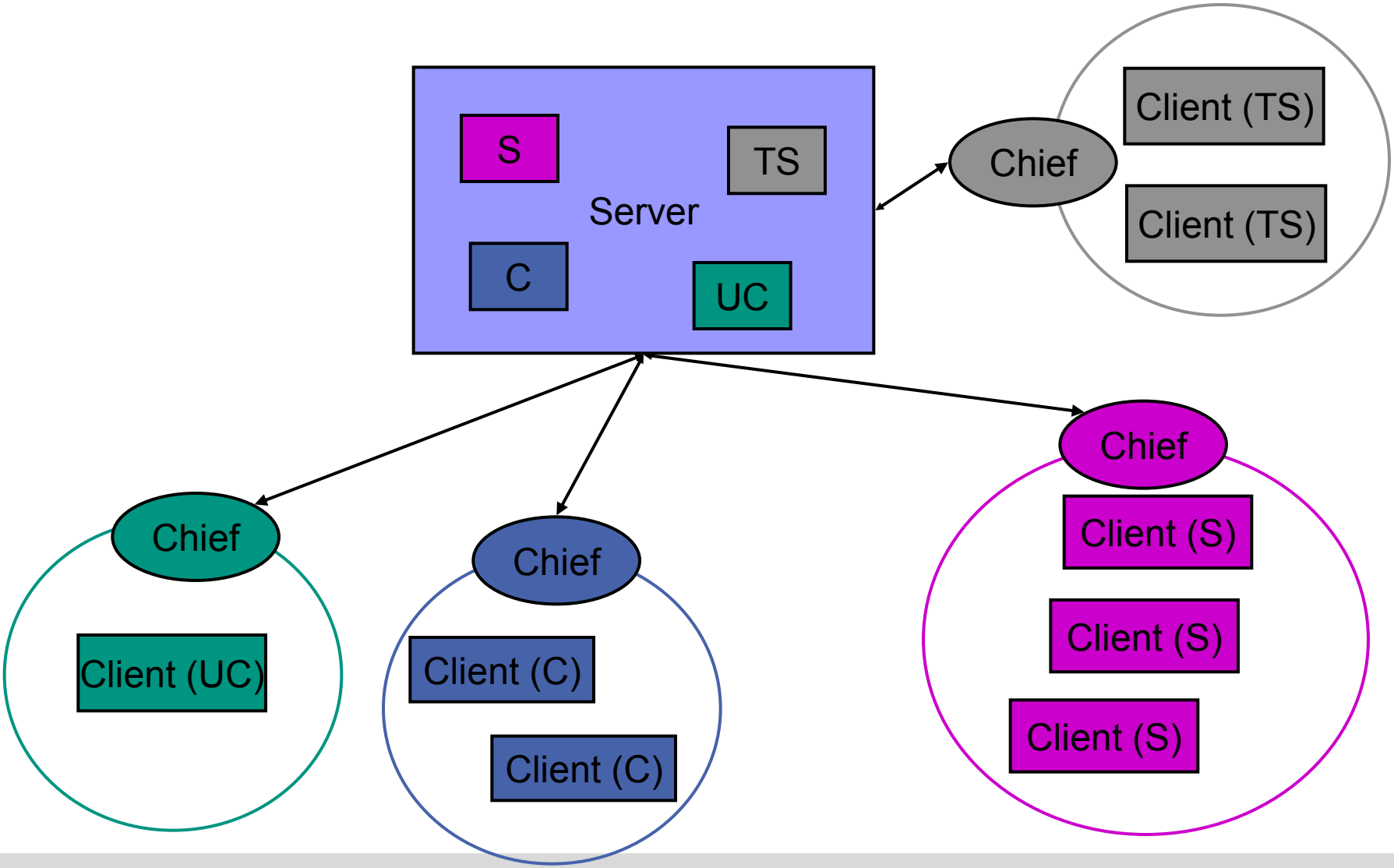
Direction-Preserving Deceiving



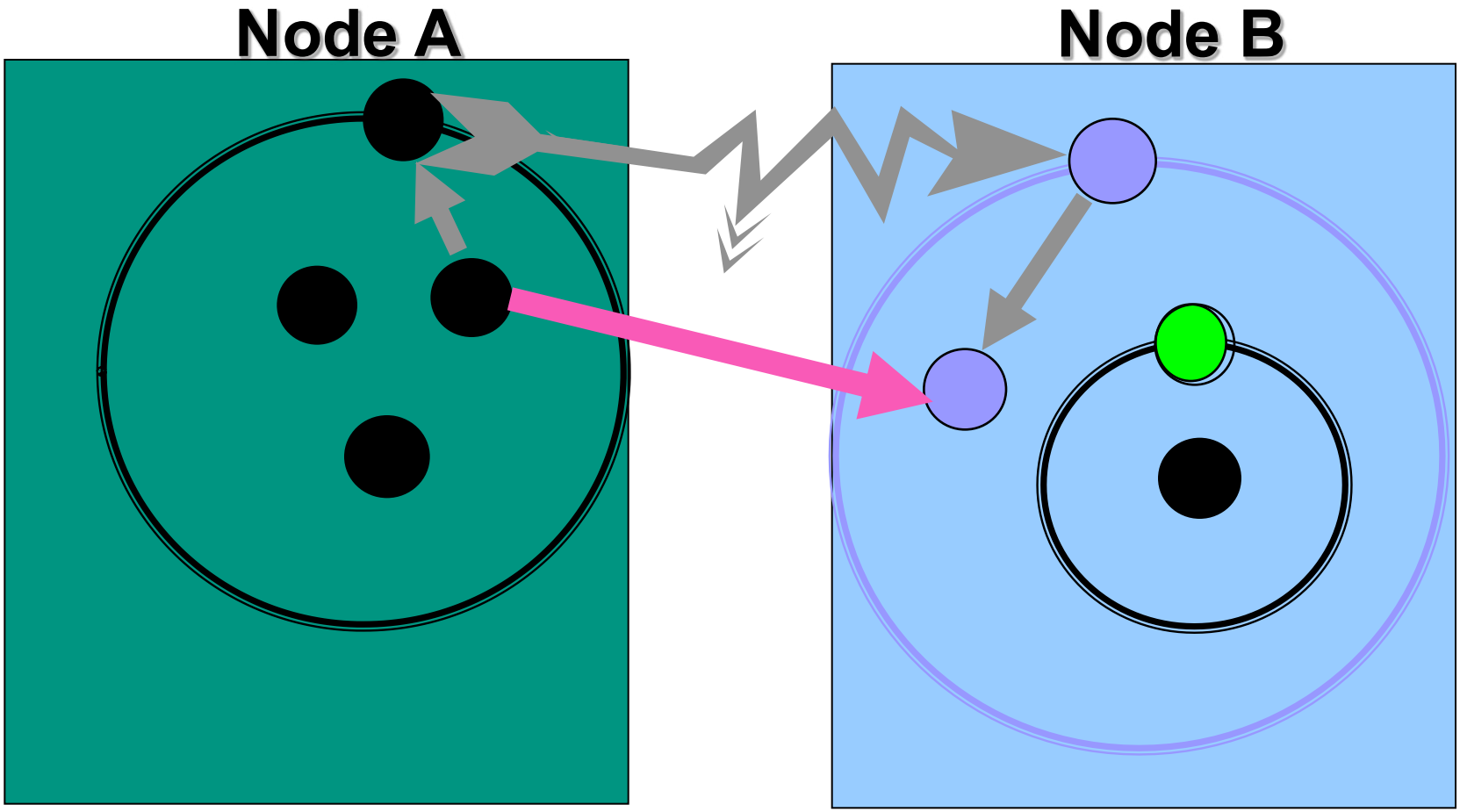
Direction-Preserving Deceiving



Secure System Using Clans & Chiefs



Remote IPC

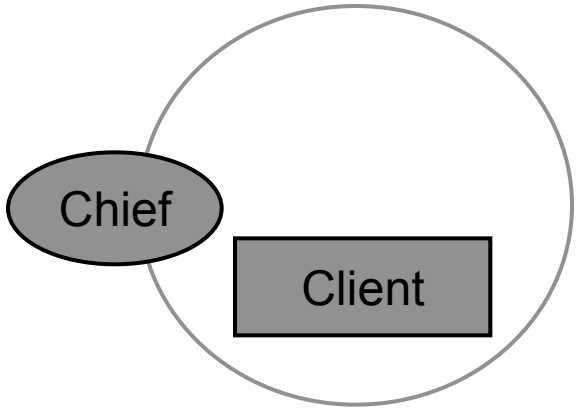


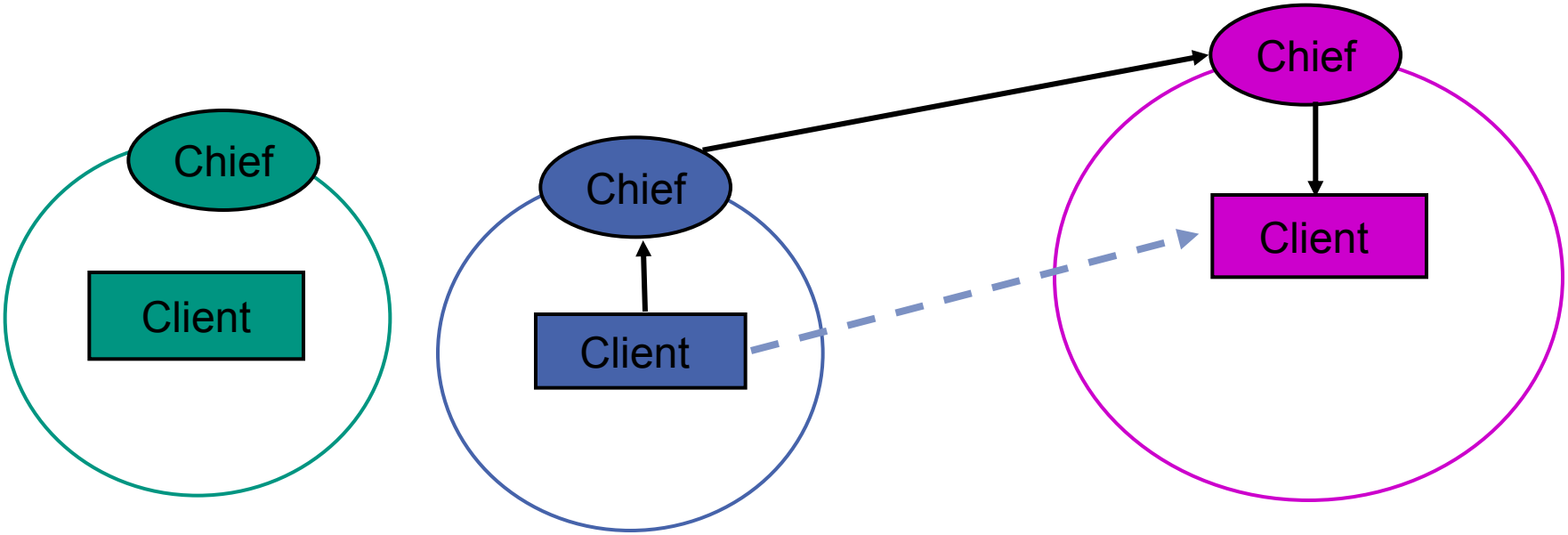
Problems with Clans & Chiefs

- Static
  - A chief is assigned when task is started
    - If we might want to control IPC, we must always assign a chief
- General case requires many more IPCs
  - Every task has its own chief

The Most General System Configuration

- Even if a pair could communicate freely we still require 3 IPCs where one would suffice



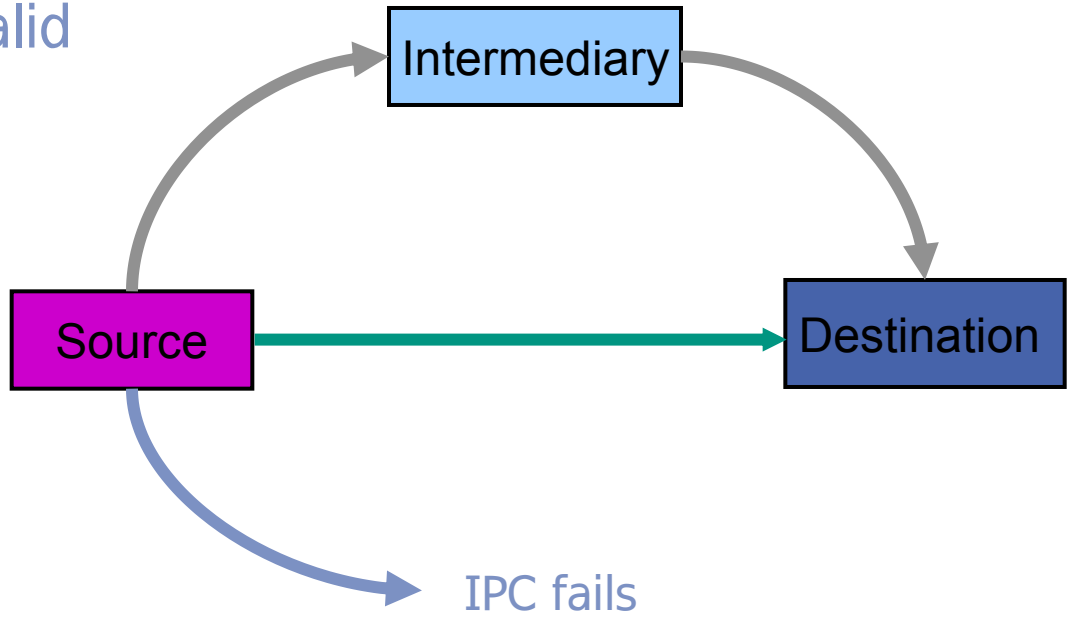


GENERIC IPC REDIRECTION

Flexibility and Dynamic Reconfiguration

IPC Redirection

- For each source and destination we actually deliver to X, where X is one of
  - Destination
  - Intermediary
  - Invalid

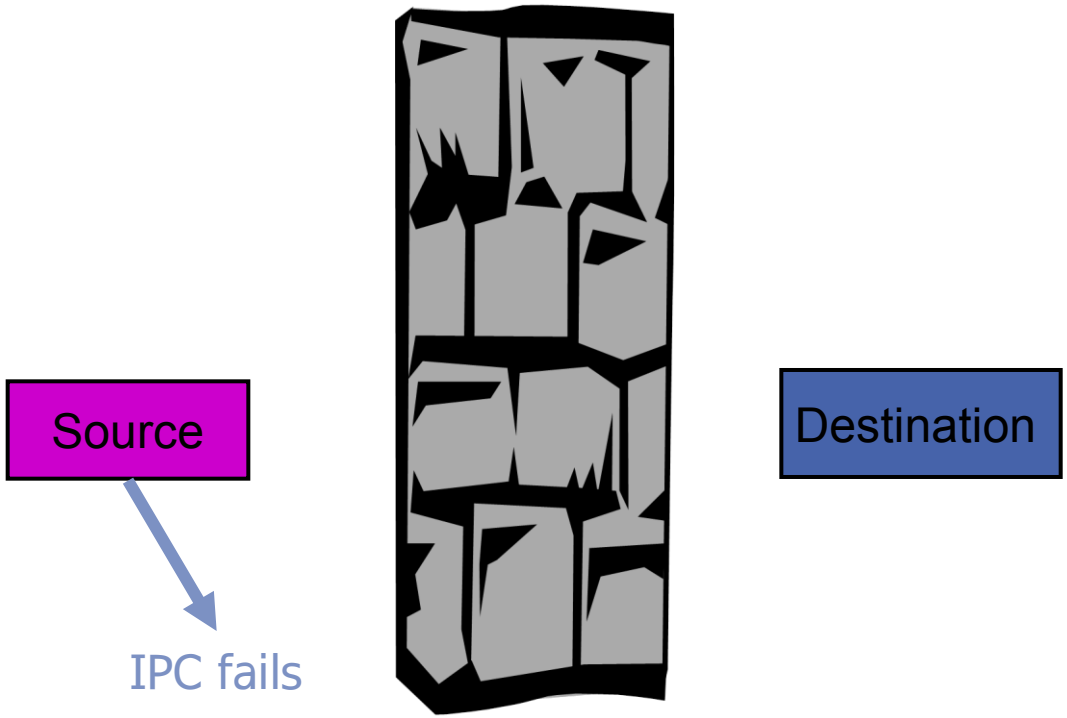


IPC Redirection

- If X = Destination
  - We have a fast path when source and destination can communicate freely

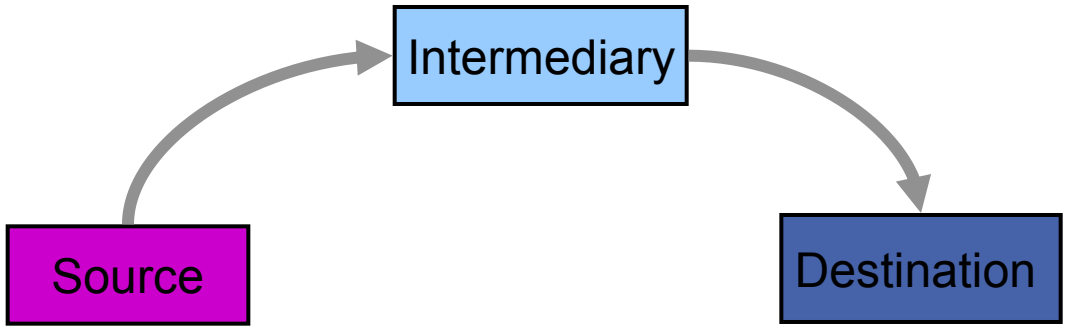
IPC Redirection

- If X = Invalid
  - We have a barrier that prevents communication completely



IPC Redirection

- If X = Intermediary
  - Enforce security policy
    - Monitor, analyze, reject, modify each IPC
  - Audit communication
  - Debug

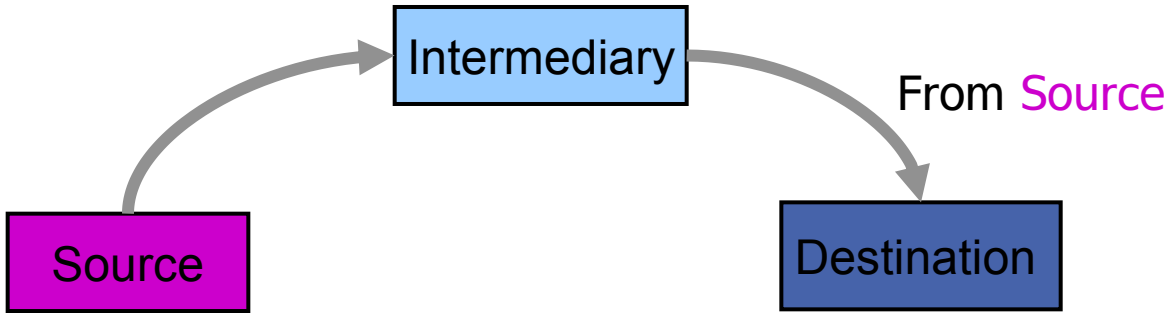


Deception

- Intermediaries must be able to deceive the destination into believing the intermediary is the original source
- An intermediary (I) can impersonate a source (S) in IPC to a destination (D)
  - I [S] ➡ D
    - If Redirection (S, D) = I, or
    - Redirection (S, D) = X and I [X] ➡ D (recursive)

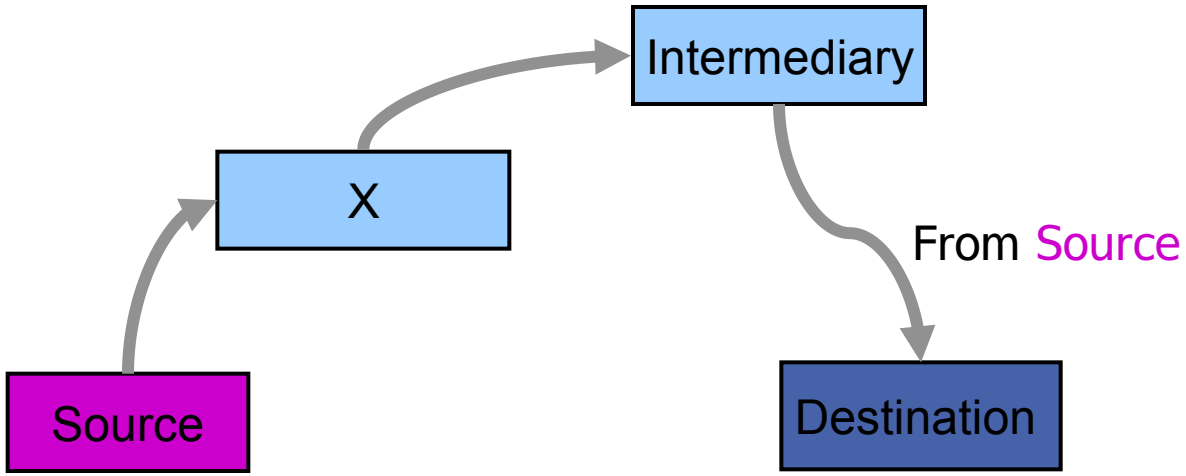
Deception: Case 1

- I [S] ➡ D if Redirection (S, D) = I

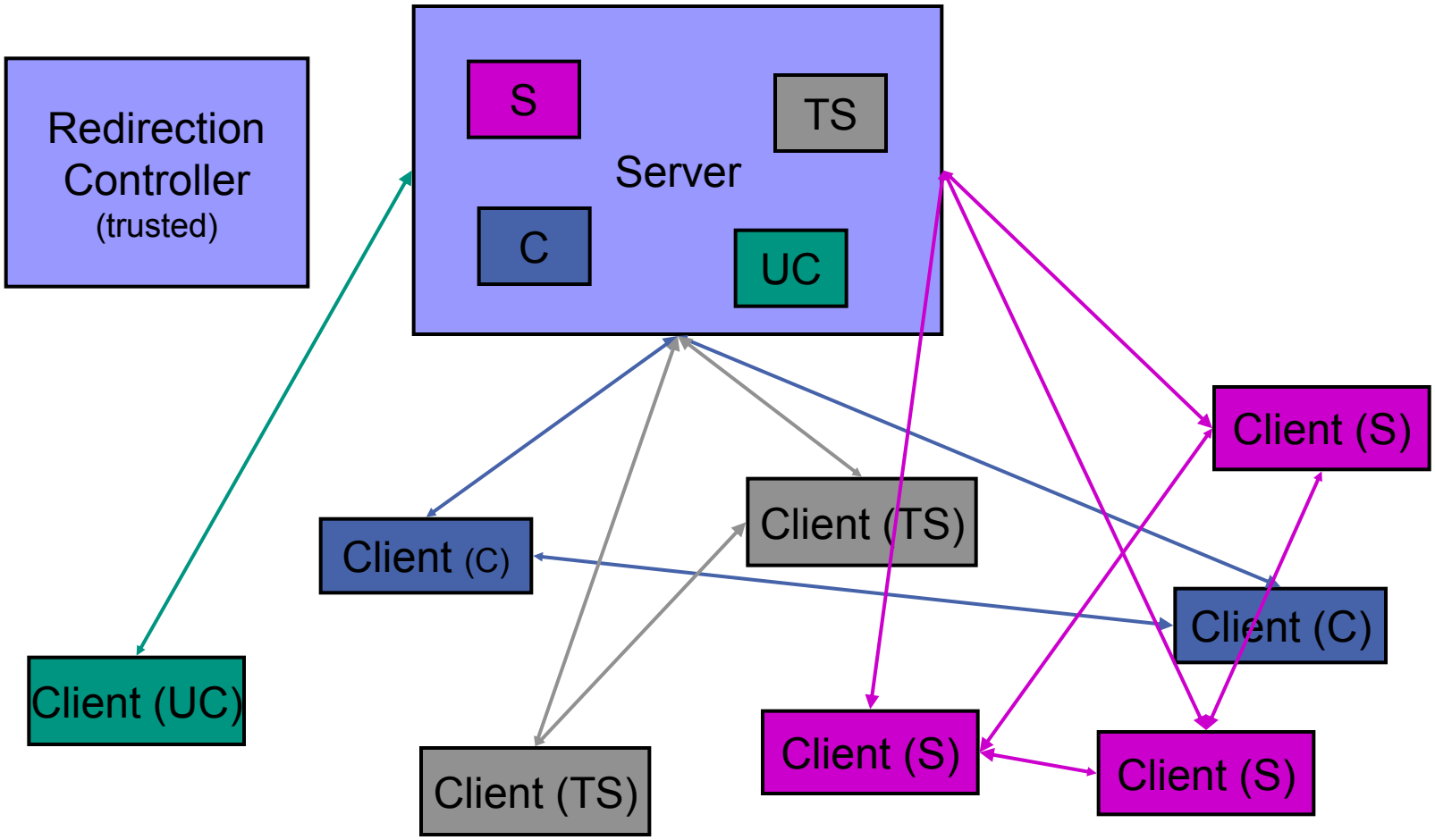


Deception: Case 2

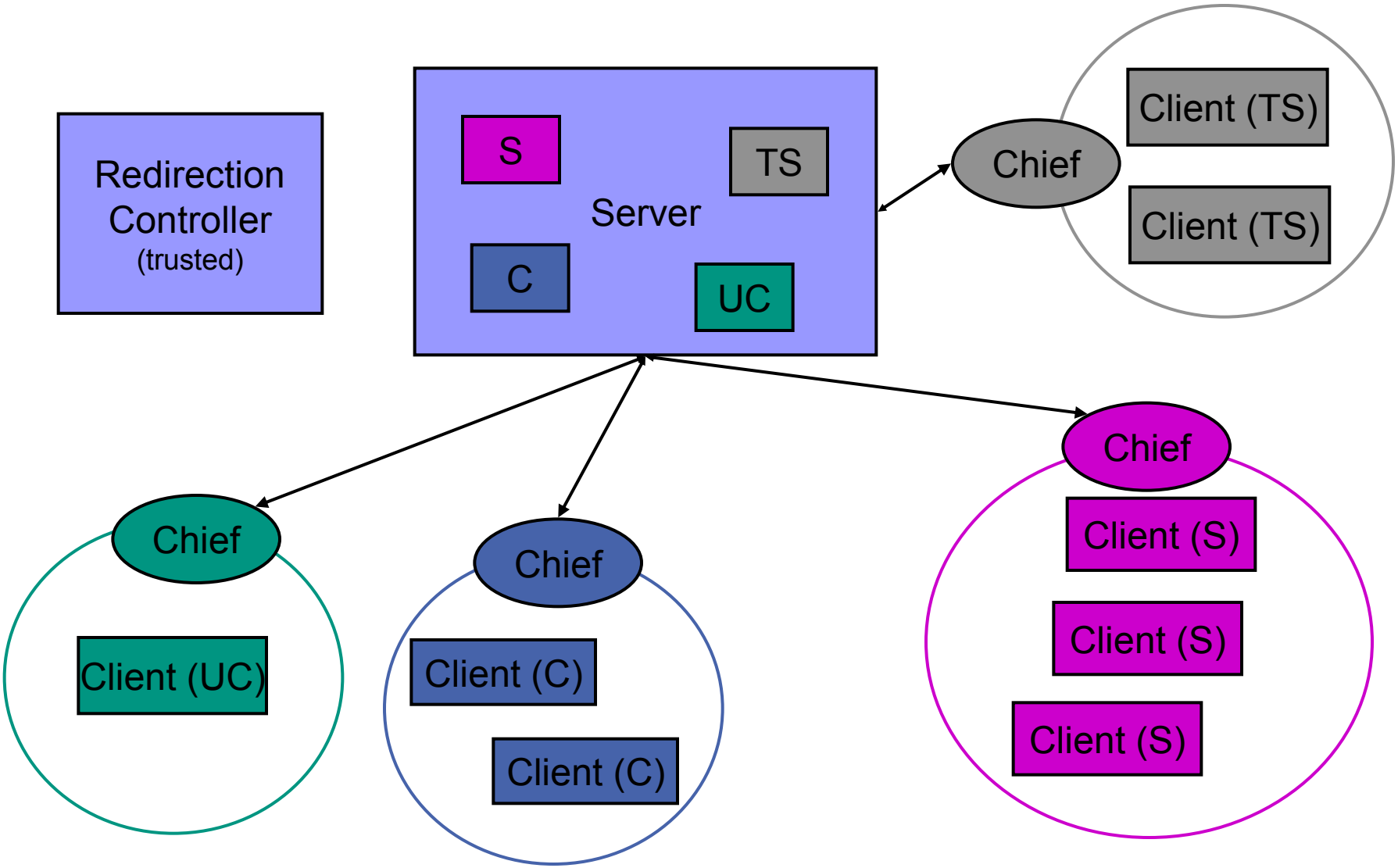
- I [S] ➡ D if Redirection (S, D) = X,
  and I [X] ➡ D (recursive)



Secure System Using IPC Redirection

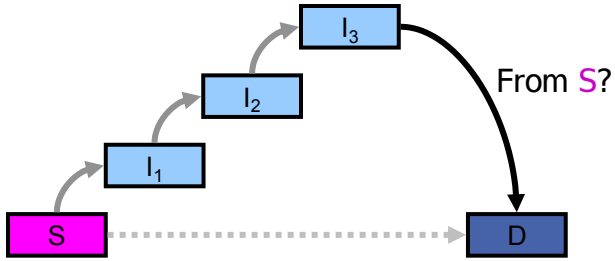


Clans & Chiefs Using IPC Redirection

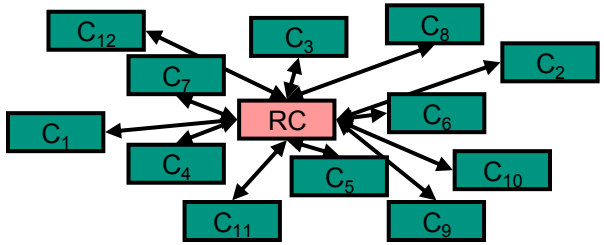


General IPC Redirection Issues

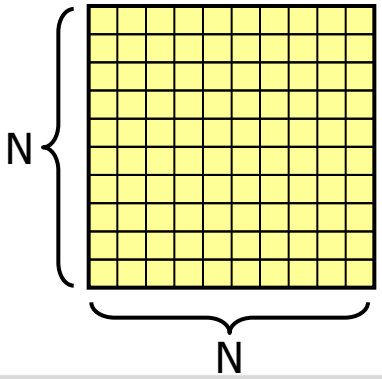
- Recursive operation
  - Can be expensive



- Centralized controller
  - Possible bottleneck



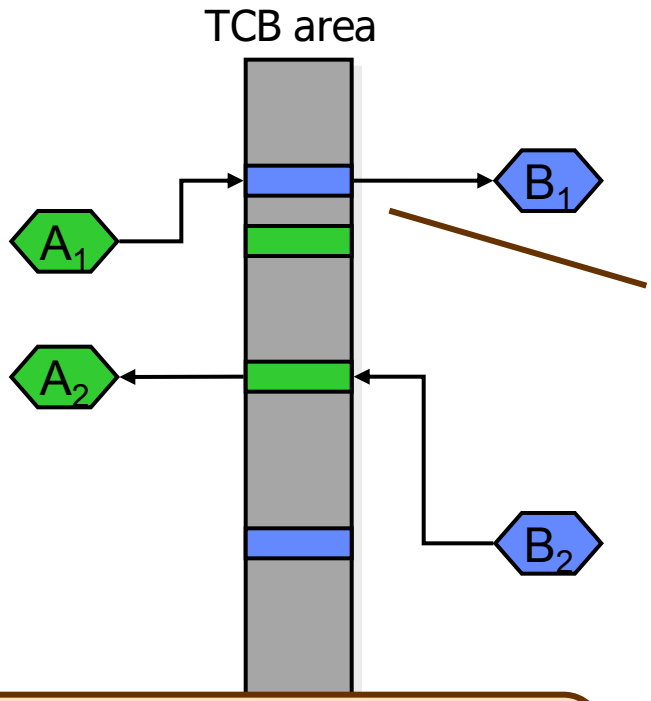
- Massive redirection structures
  - N×N array (N = num. threads)



CAPABILITIES

Decentralized IPC Management

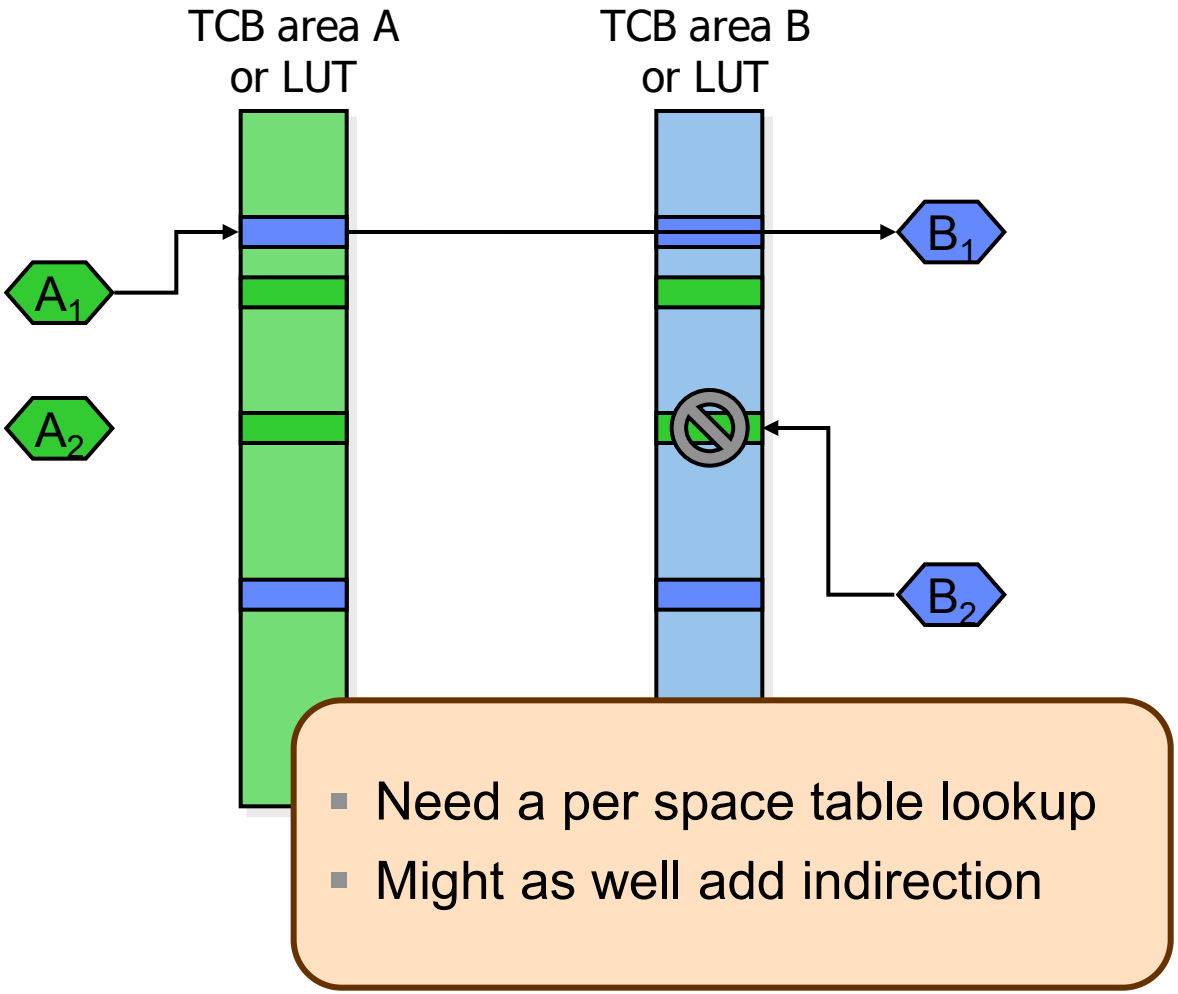
Communication Spaces
Current Model: Single Global Space



- Direct naming using global thread ID
- Extremely fast (no indirection)

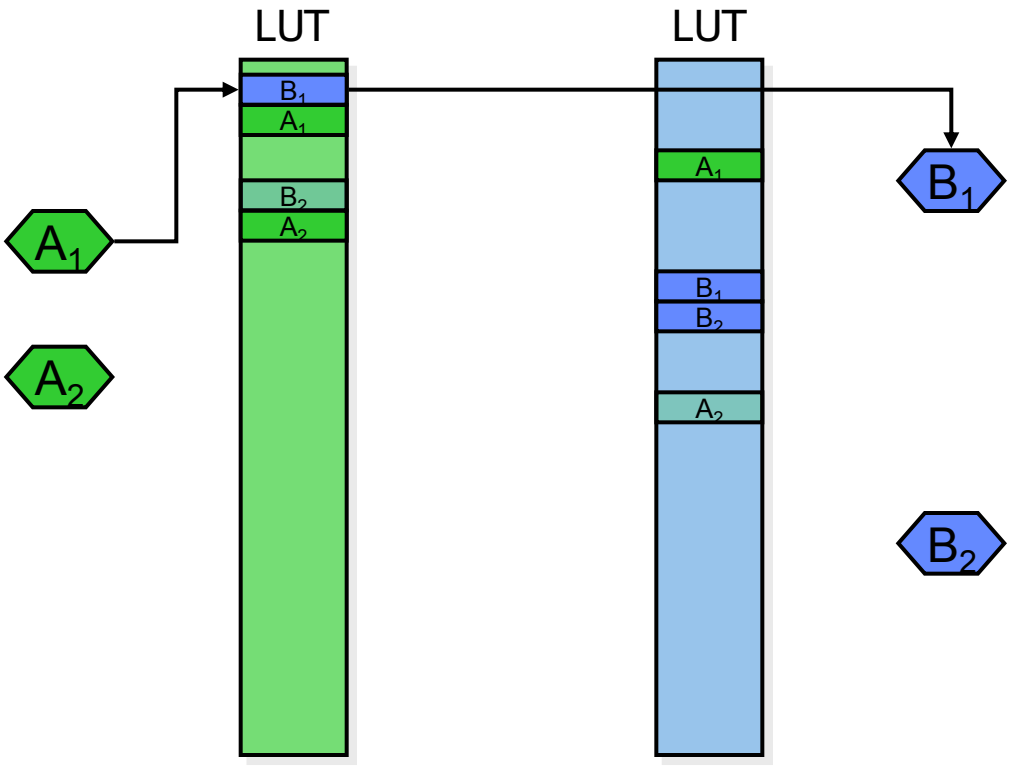
Must be able to restrict access rights on a per address space basis

Communication Spaces
Possible Solution: Per Space Access Rights

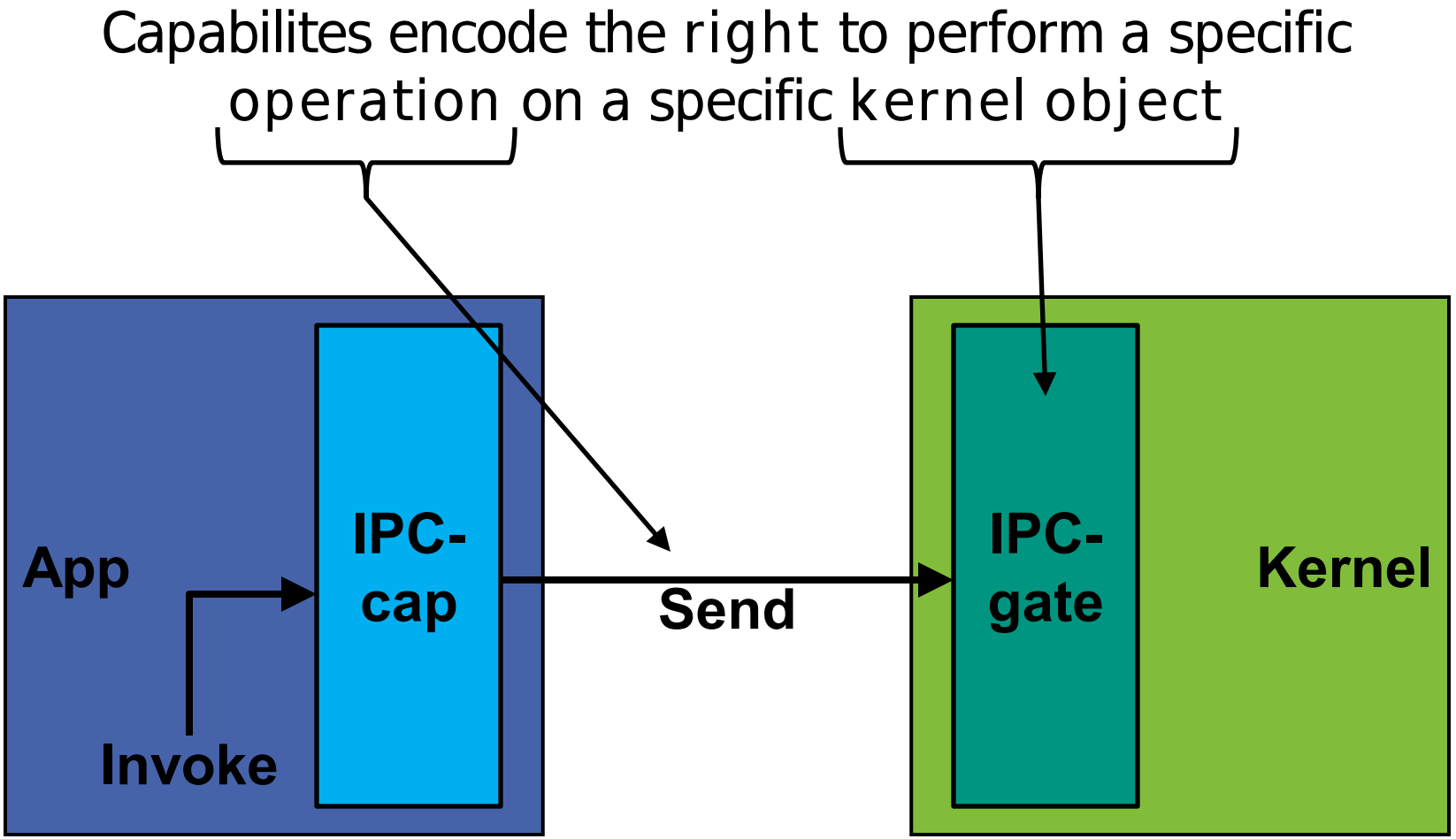


- Need a per space table lookup
- Might as well add indirection

Communication Spaces
Better Solution: Per Space Capability Array



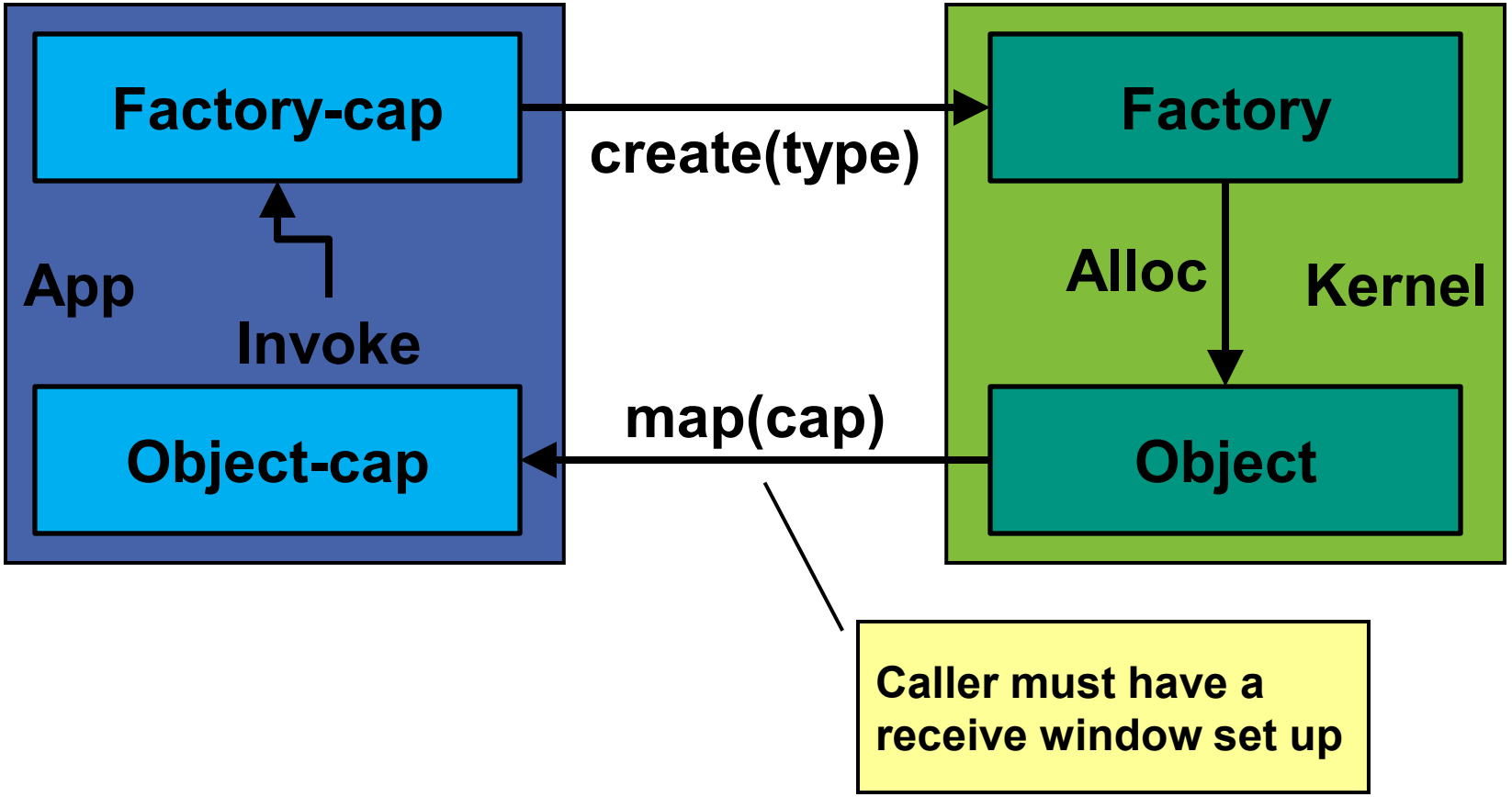
Capabilites



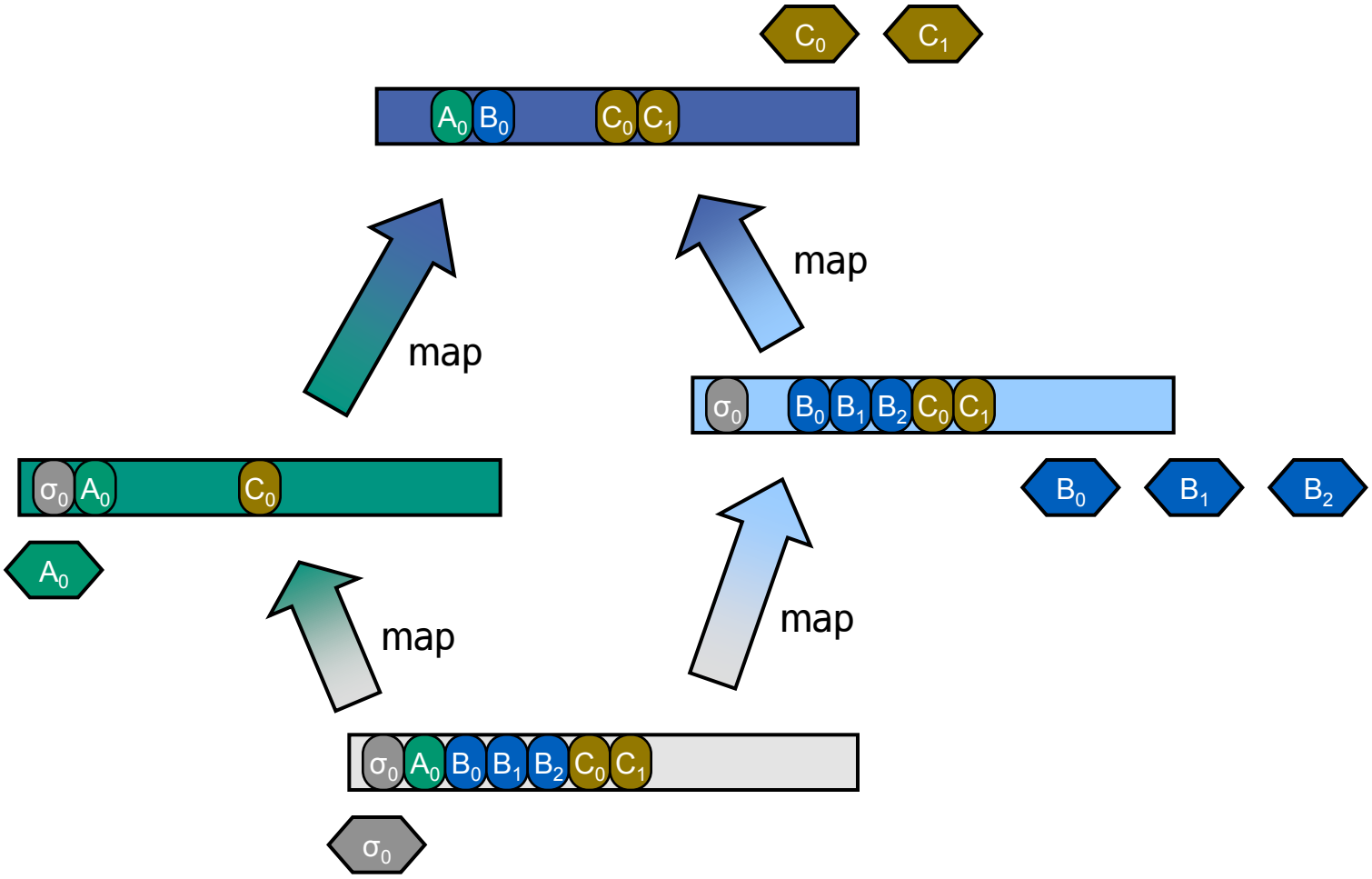
Capability properties

- Capabilities contain
  - Pointer to a kernel object
  - Access rights
- Capabilities live in kernel space
  - Not directly accessible to user
  - Referenced by index in per-AS capability array
- Capabilities provide:
  - Fine-grained access control
  - Local naming (name = idx in capability array)
    - Index has no meaning in other ASes!

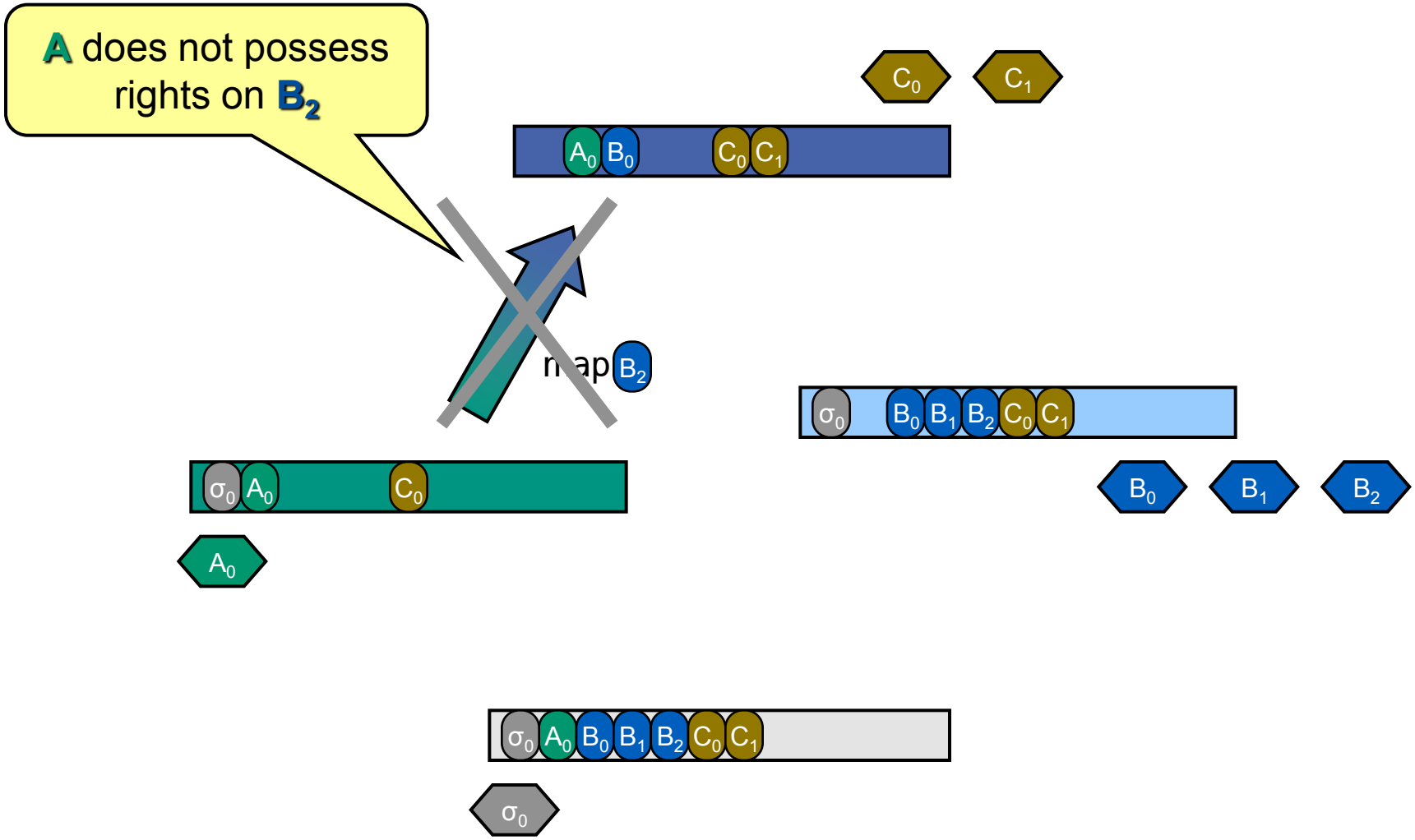
Creating capabilites



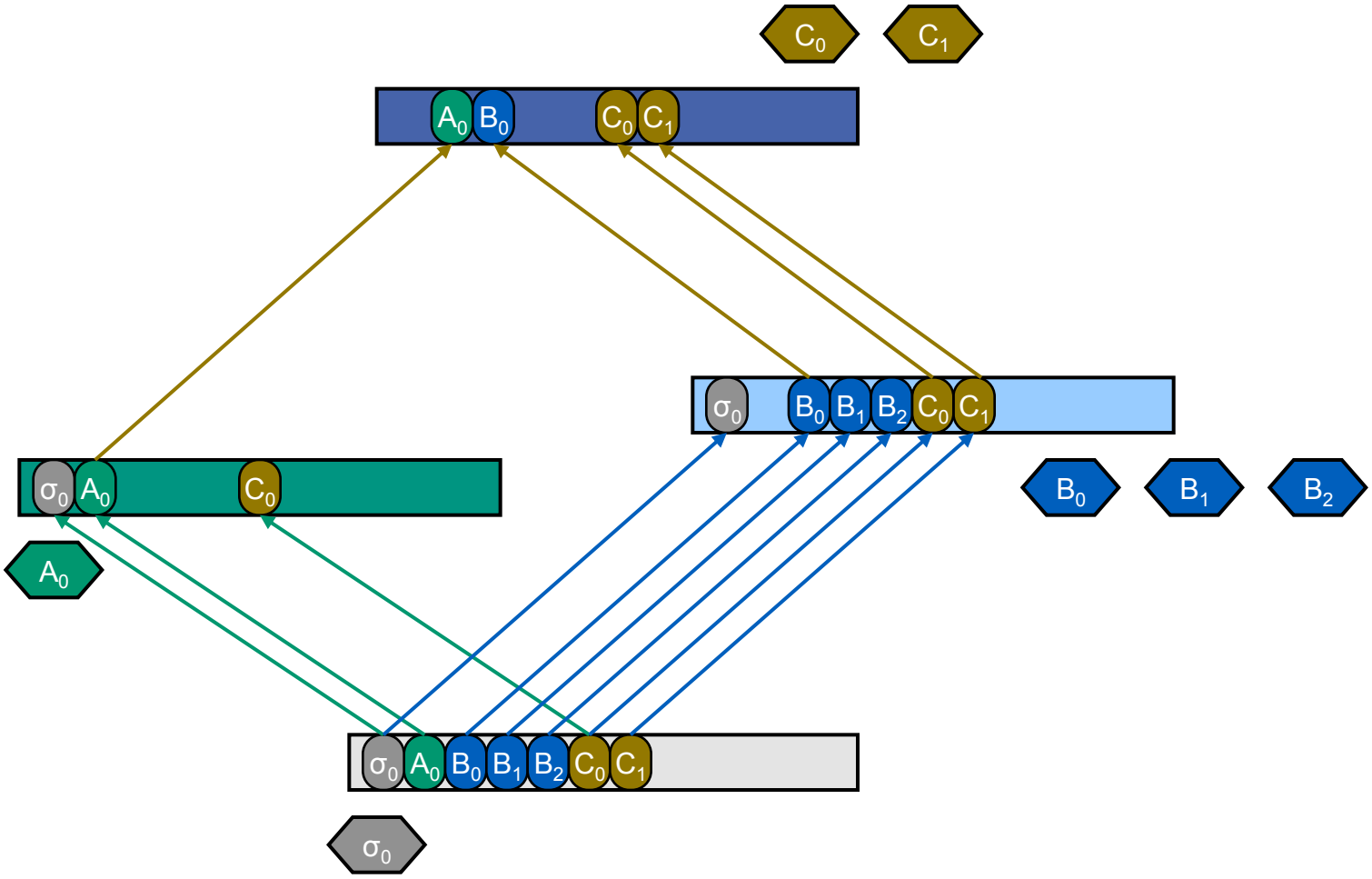
Communication Spaces with capabilites



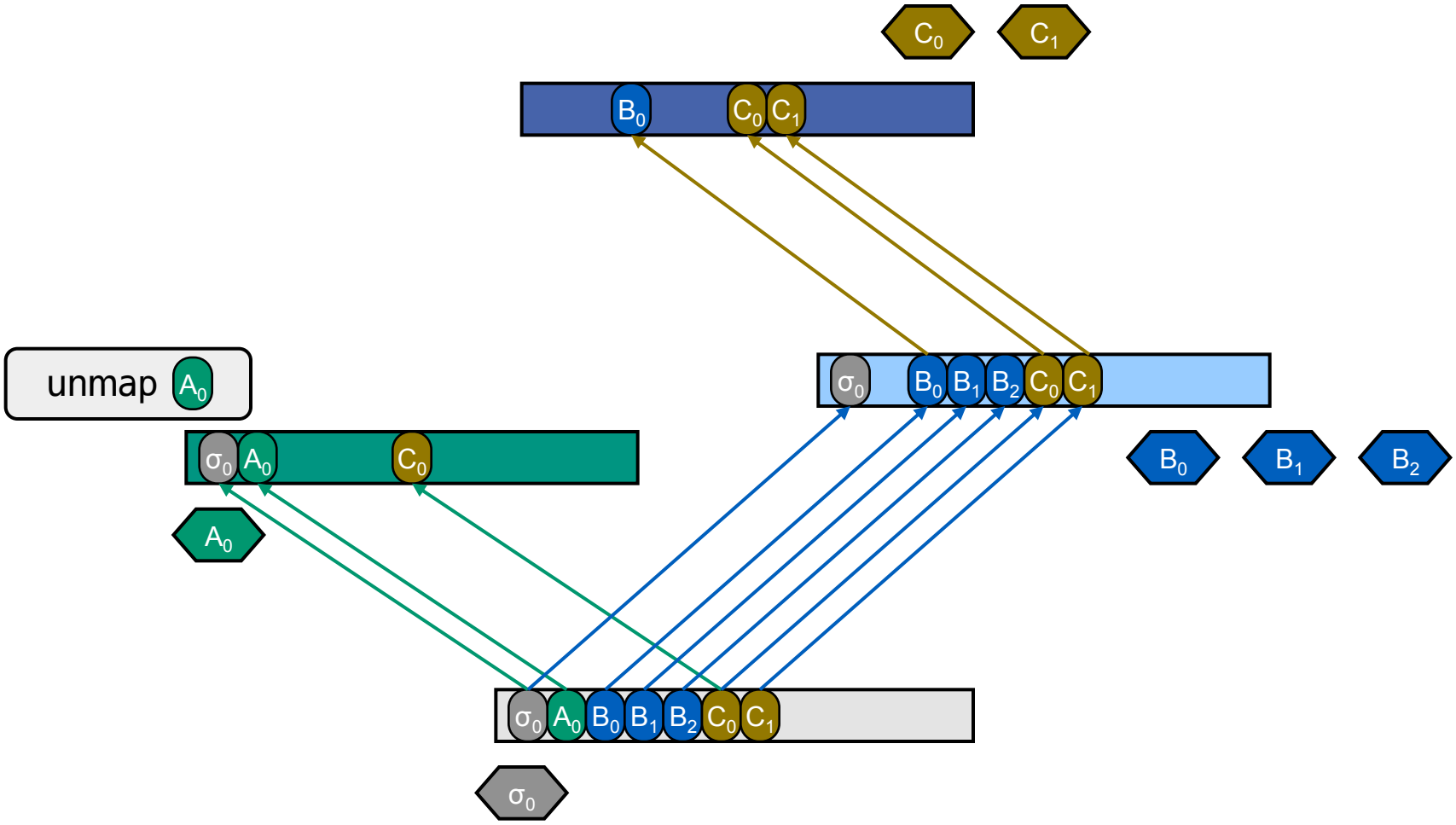
Mapping Communication Rights



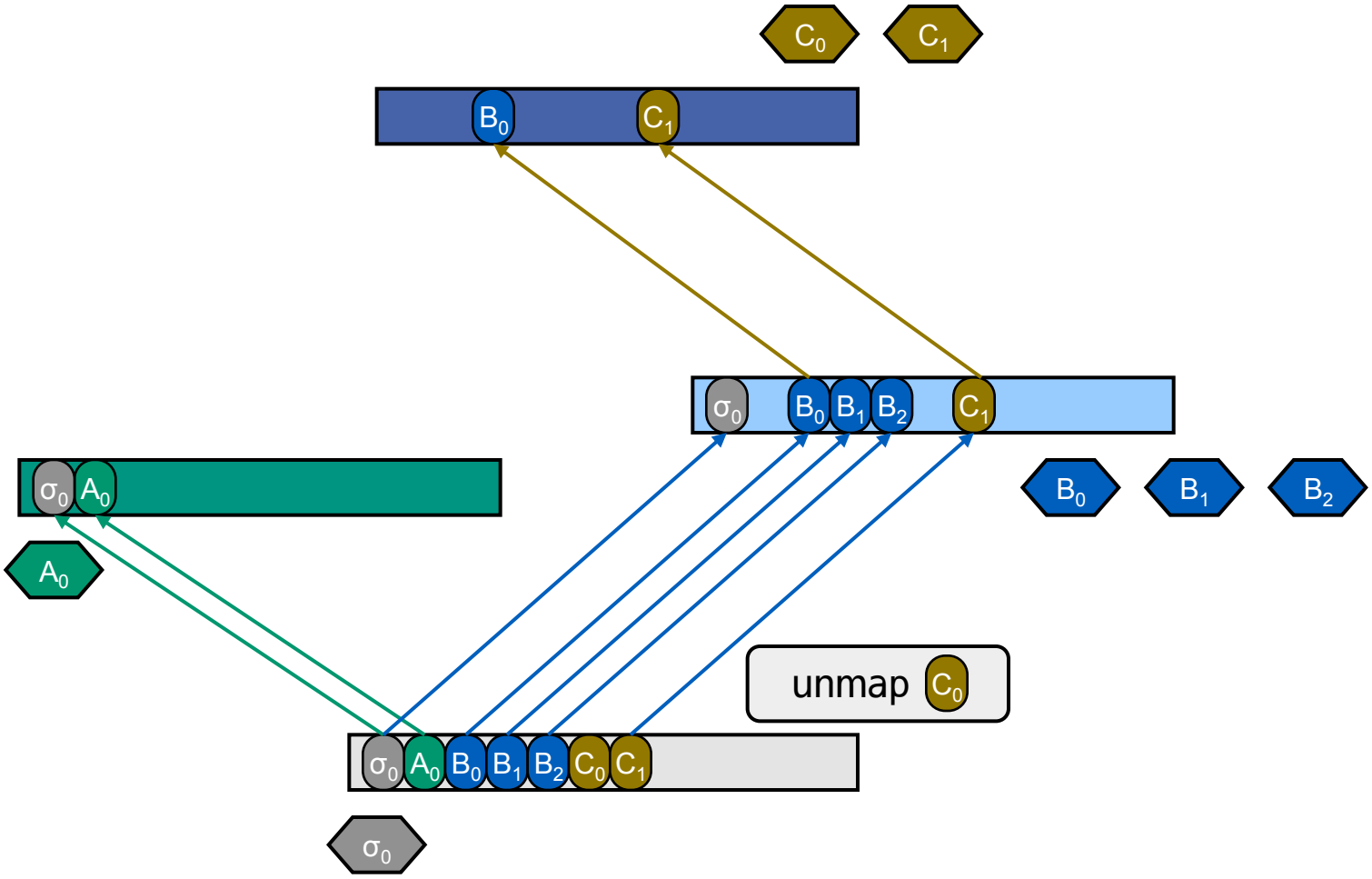
Revoking Communication Rights



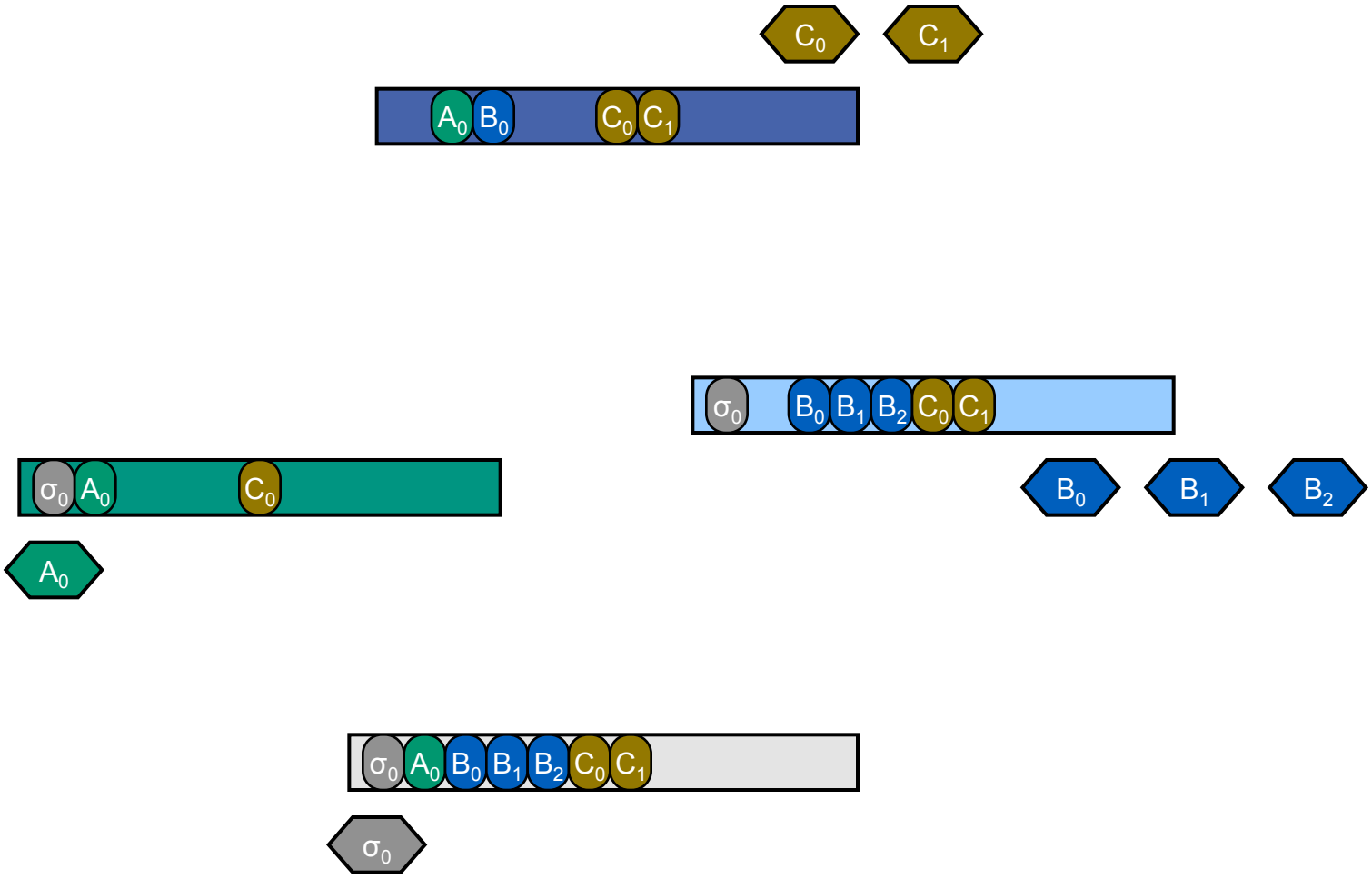
Revoking Communication Rights



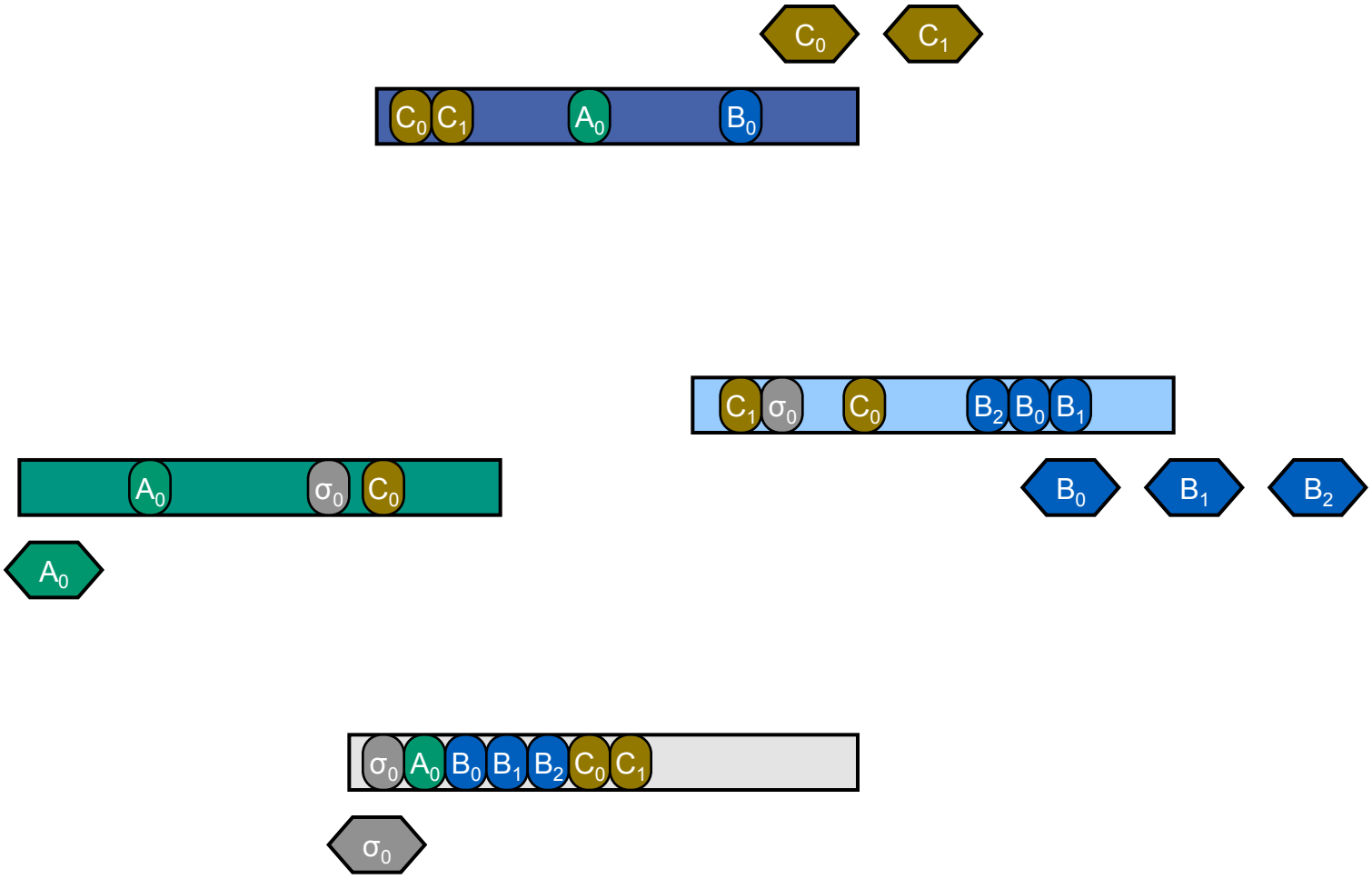
Revoking Communication Rights



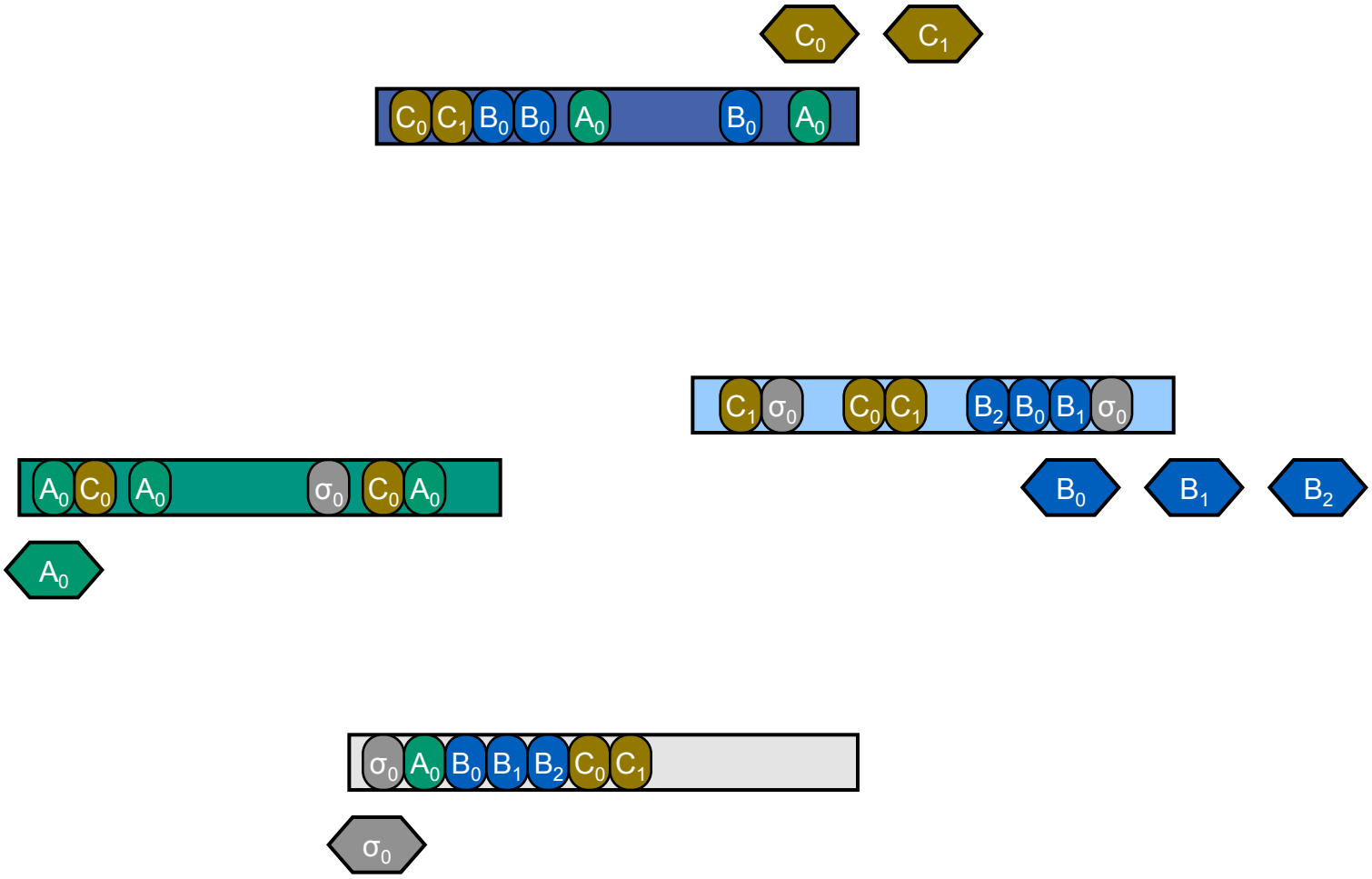
Virtual Communication Spaces
Arbitrary Thread ID Layout



Virtual Communication Spaces
Arbitrary Thread ID Layout

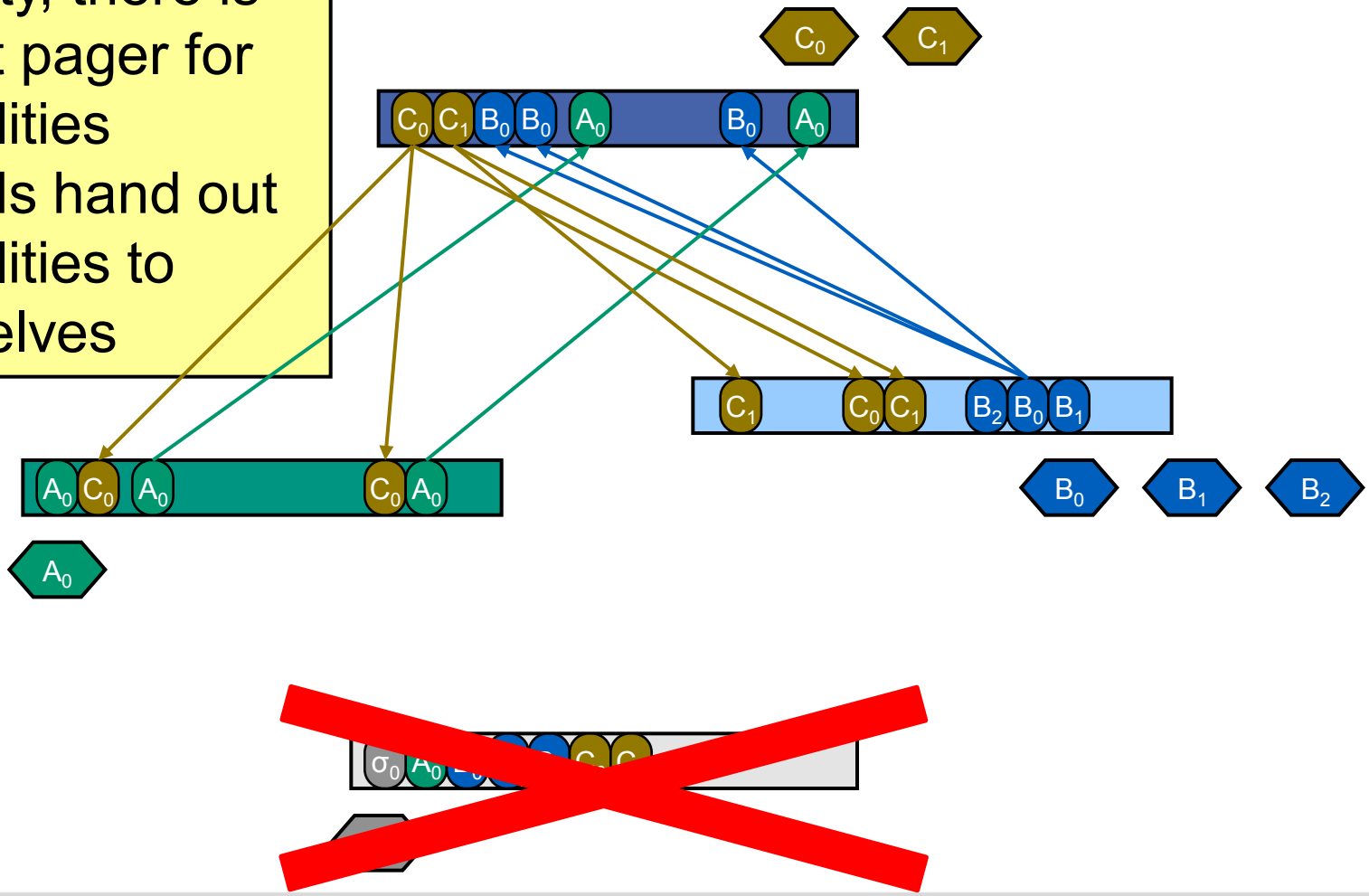


Virtual Communication Spaces
Arbitrary Thread ID Layout



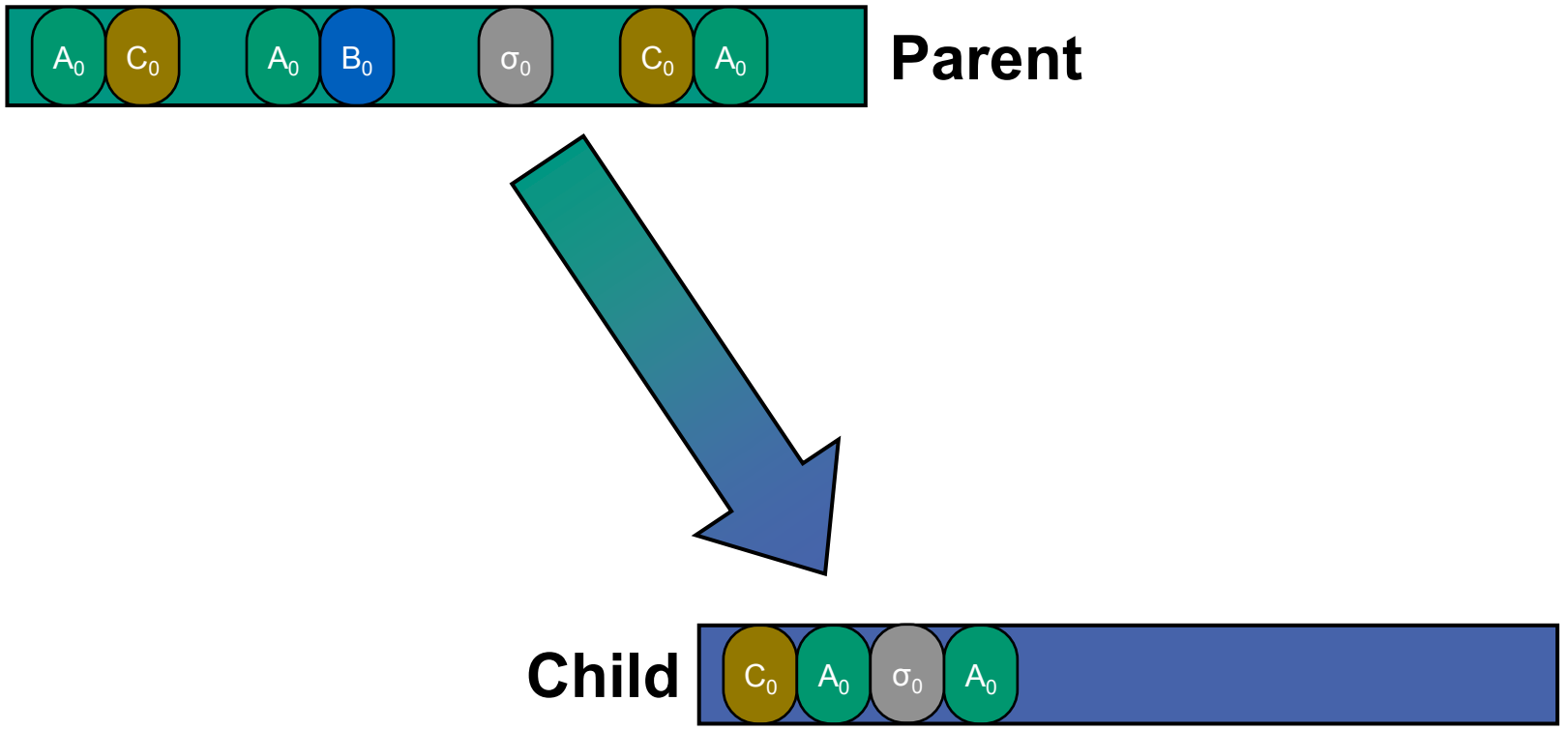
Virtual Communication Spaces
Decentralized access control

- In reality, there is no root pager for capabilities
- Threads hand out capabilities to themselves

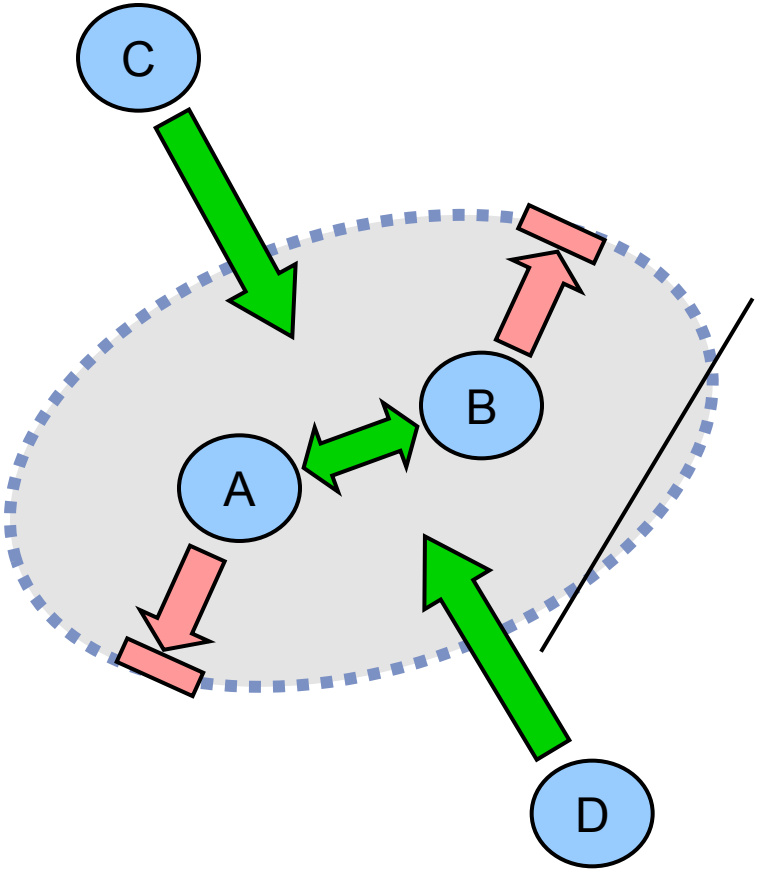


Bootstrapping

- Parent fills child's capability array on launch
- Parent receives thread capability for child



Confinement (revisited)



- Can map communication rights
- Can map writable memory
- Need ability to restrict such mappings
  - Restricted via map-right

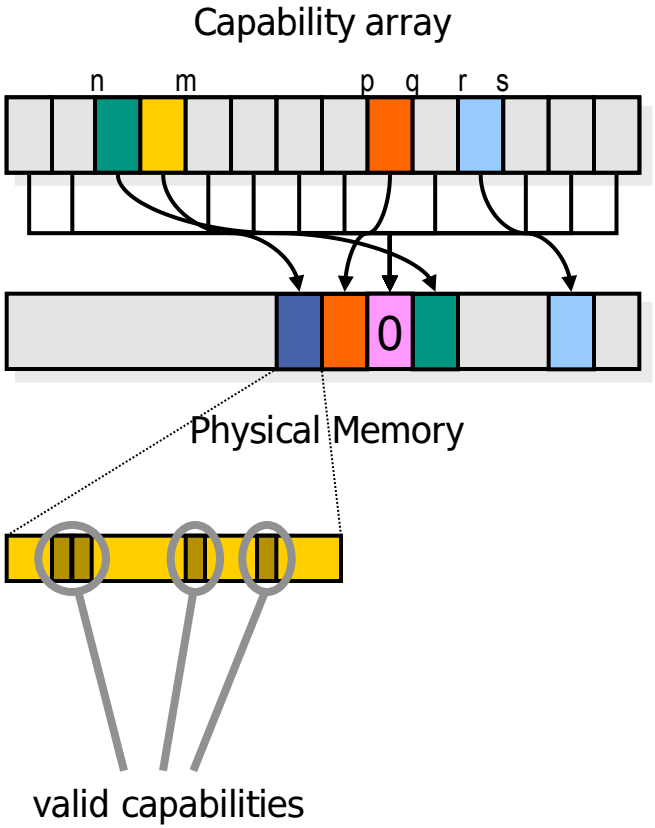
Capabilites: Implications on IPC Performance

- Need table lookup (indirection) to find destination thread
  - Table lookup needed anyway to check rights
- Implications of indirection for TCB lookup
  - – One more cache line access per IPC
  - \+ Smaller TLB footprint (sometimes, cf. mkc-03-aslayout)
    - TLBs usually smaller than caches
    - TLB misses often more expensive than cache misses

Capability array

- Lookup on each IPC invocation
  - Must be extremely efficient
  - Avoid any excess indirection
  - Indirection increases
    - Cache footprint
    - Number of direct and/or indirect cache misses
- Implemented via Virtual Linear Array (VLA)
  - Lookup into dedicated virtual memory area
  - Area with a valid mapping backed by dedicated page frame
  - Area with no valid mapping backed by zero page
    - All read accesses return zero
    - Cf. 0-mapping trick

Capability Array



- Implemented via Virtual Linear Array (VLA)
  - Lookup into dedicated virtual memory area
  - Area with a valid mapping backed by dedicated page frame
  - Area with no valid mapping backed by zero page
    - All read accesses return zero
    - Cf. 0-mapping trick

All Access is via IPC (revisited)

- What microkernel mechanisms are needed for security?
  - How do we authenticate?
    - Sender's ID revealed on IPC
    - Sender ID is unforgeable

All Access is via IPC (revisited)

- What microkernel mechanisms are needed for security?
  - How do we authenticate?
  - How do we perform authorization?
    - Give thread rights to communicate via mappings
    - Revoke rights to communicate via unmap
    - Individual servers can decide on fine grained policies

All Access is via IPC (revisited)

- What microkernel mechanisms are needed for security?
  - How do we authenticate?
  - How do we perform authorization?
  - How do we implement arbitrary security policies?
    - Authorization performed completely in user-level

All Access is via IPC (revisited)

- What microkernel mechanisms are needed for security?
  - How do we authenticate?
  - How do we perform authorization?
  - How do we implement arbitrary security policies?
  - How do we enforce arbitrary security policies?
    - Any communication requires the appropriate communication right

KERNEL SECURITY

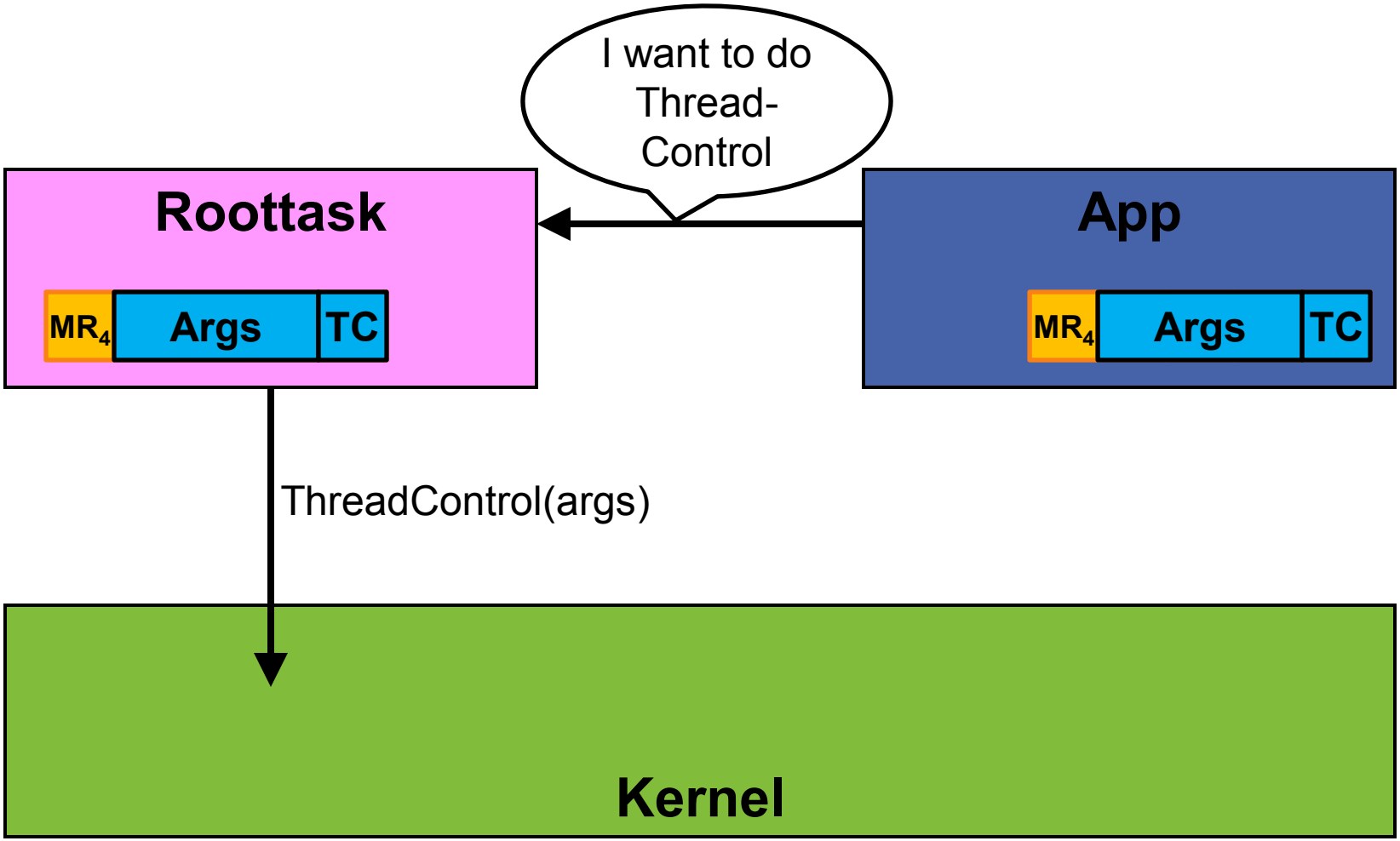
How to secure system calls and kernel resources

Problems with the kernel

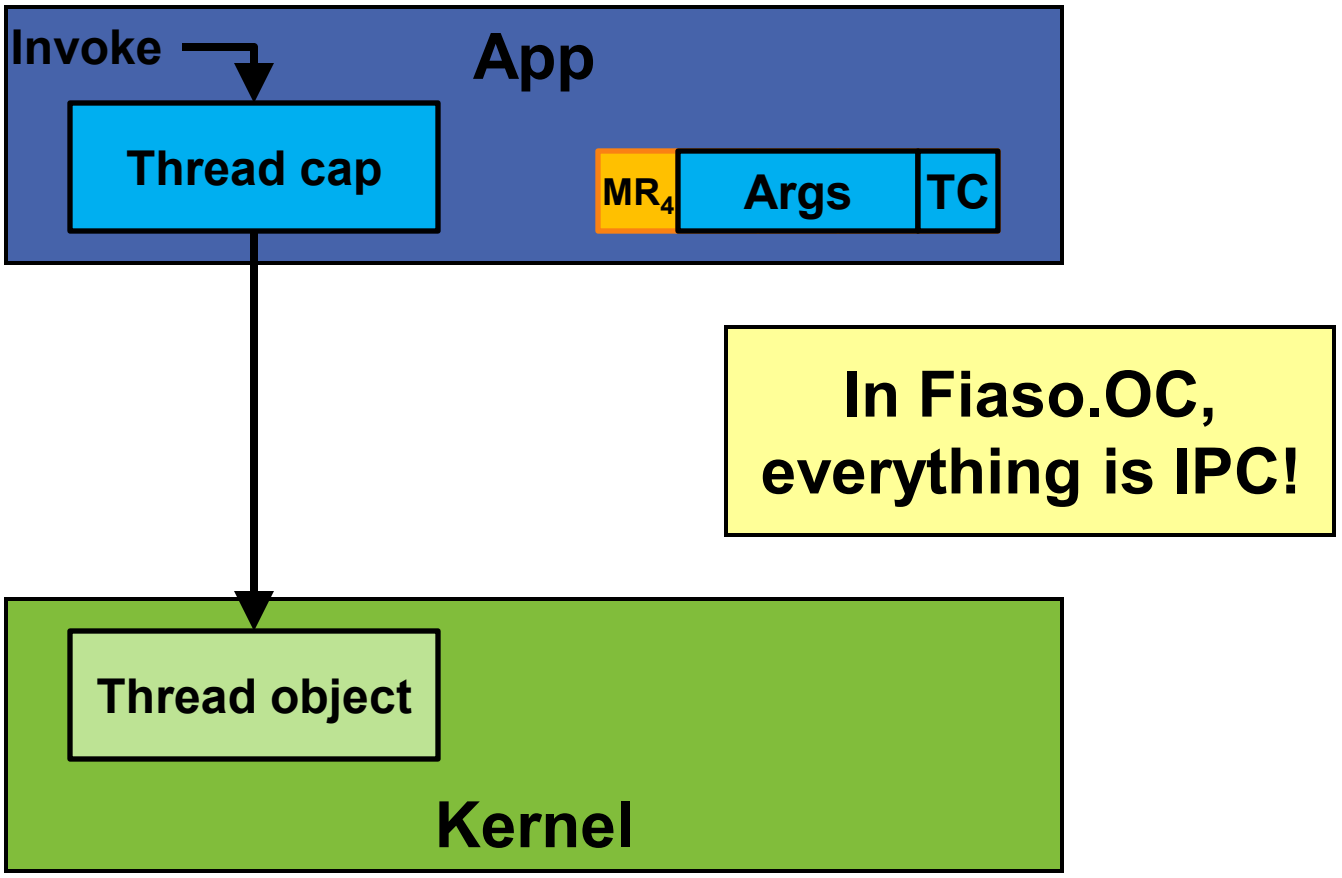
- We can stop applications from attacking each other
- What about applications attacking the kernel?
  - DoS anyone?
- What about the kernel attacking applications?
  - Can't help it! The kernel is all-powerful
- What about applications attacking each other through the kernel?

- ~~Kernel needs to restrict access to its functions~~
  - Remember: No policy in the kernel!
- Need to restrict access to kernel functions from userland

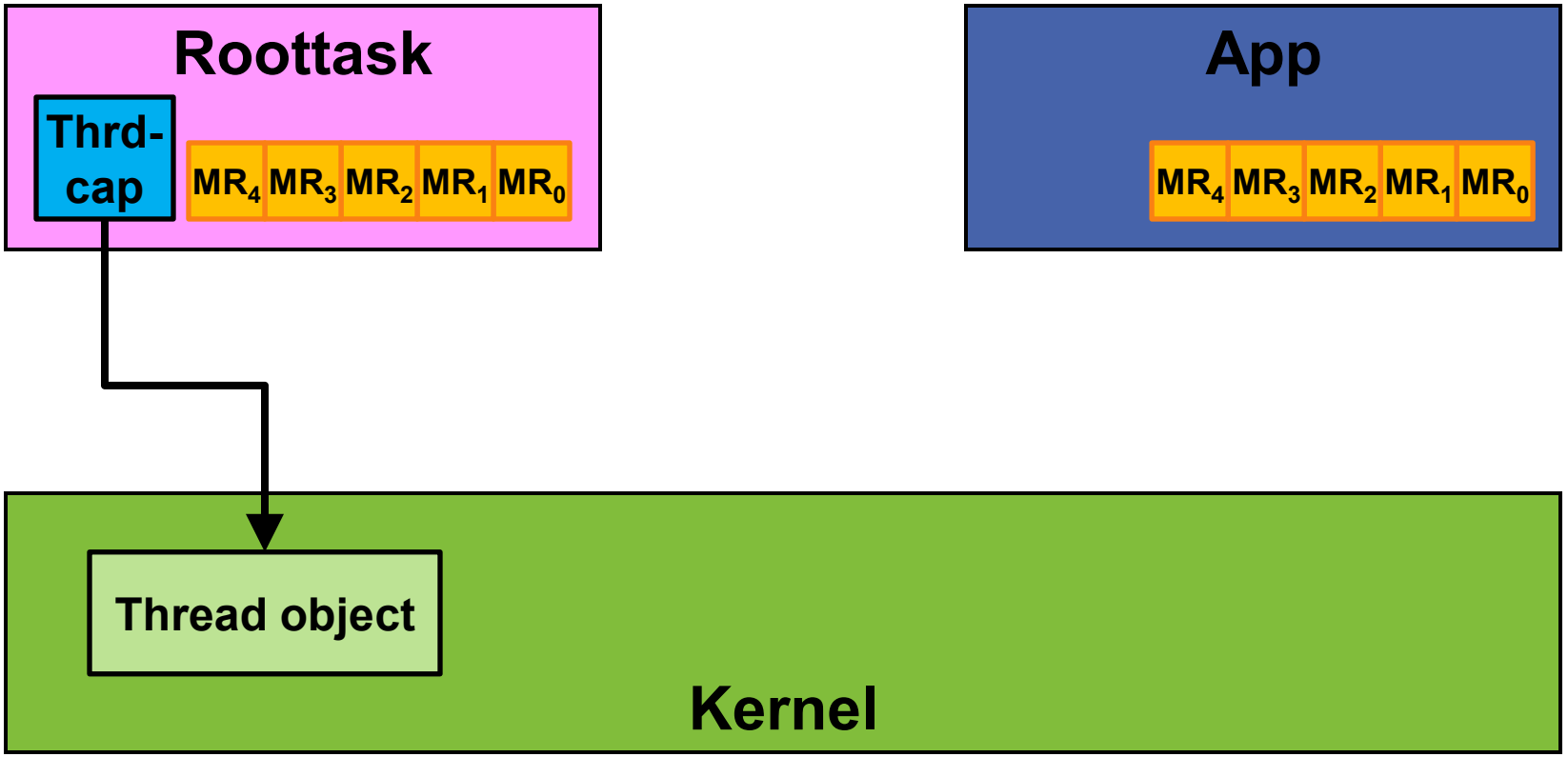
System call indirection in Pistachio



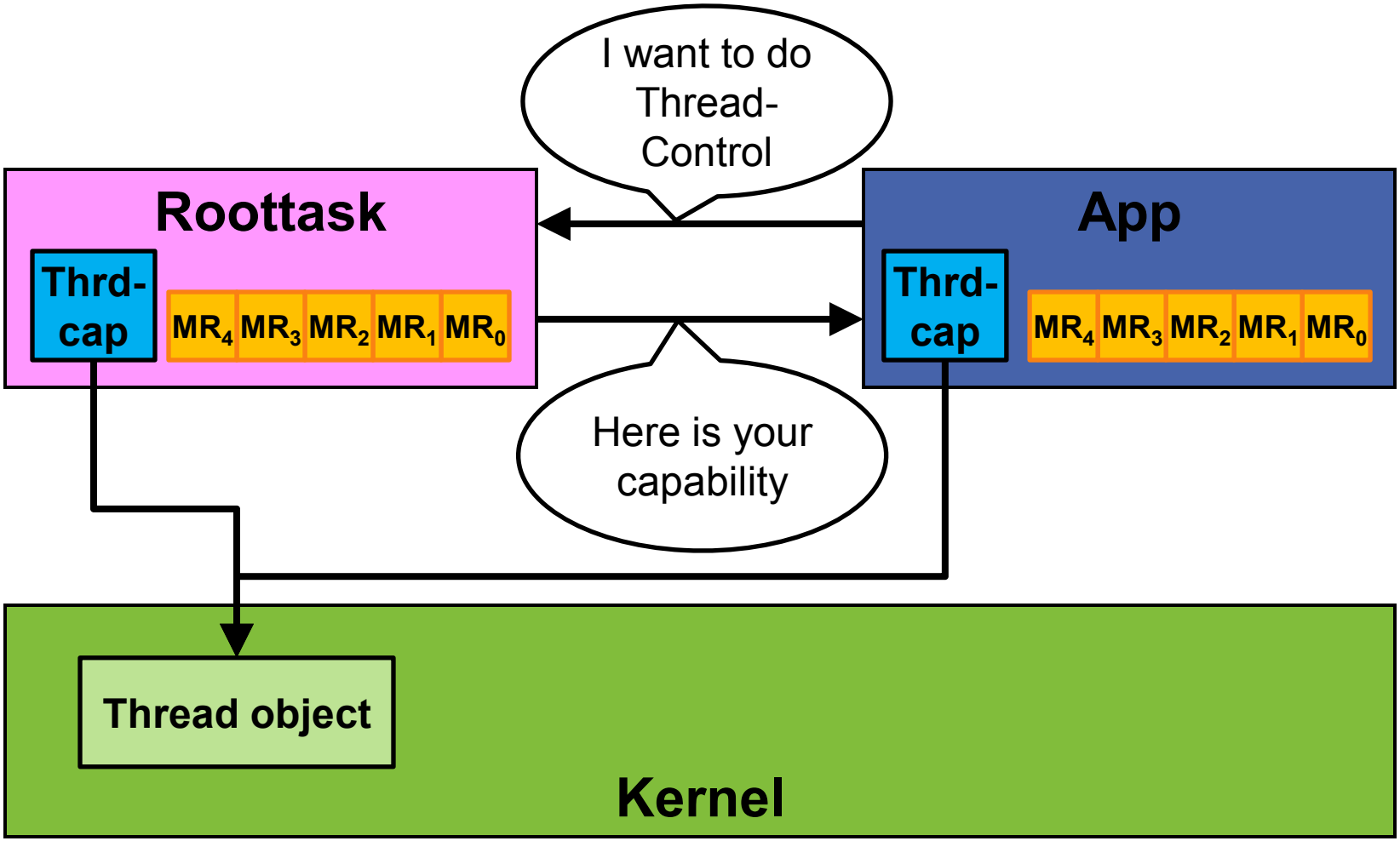
System calls in Fiasco.OC



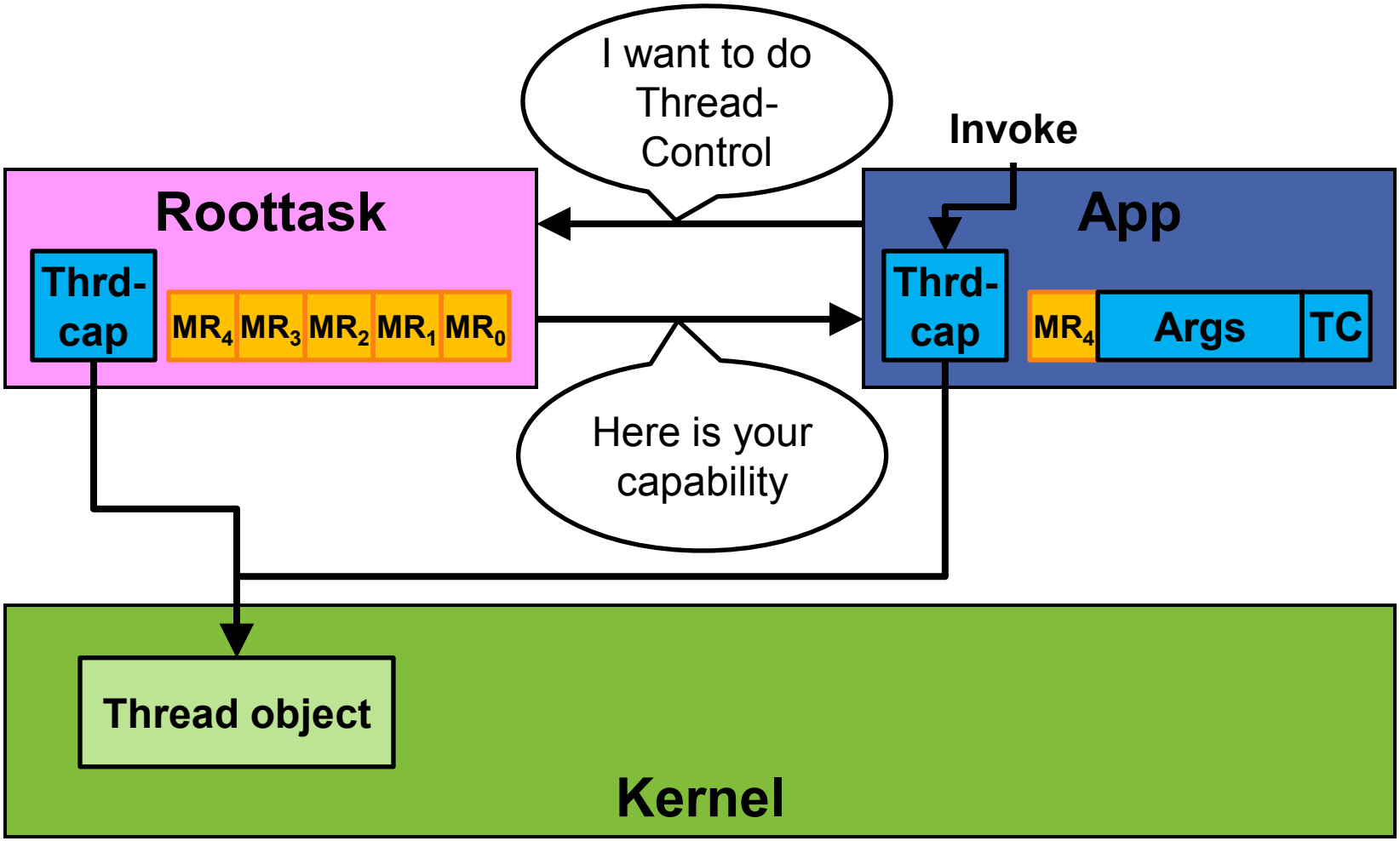
System calls with capabilities



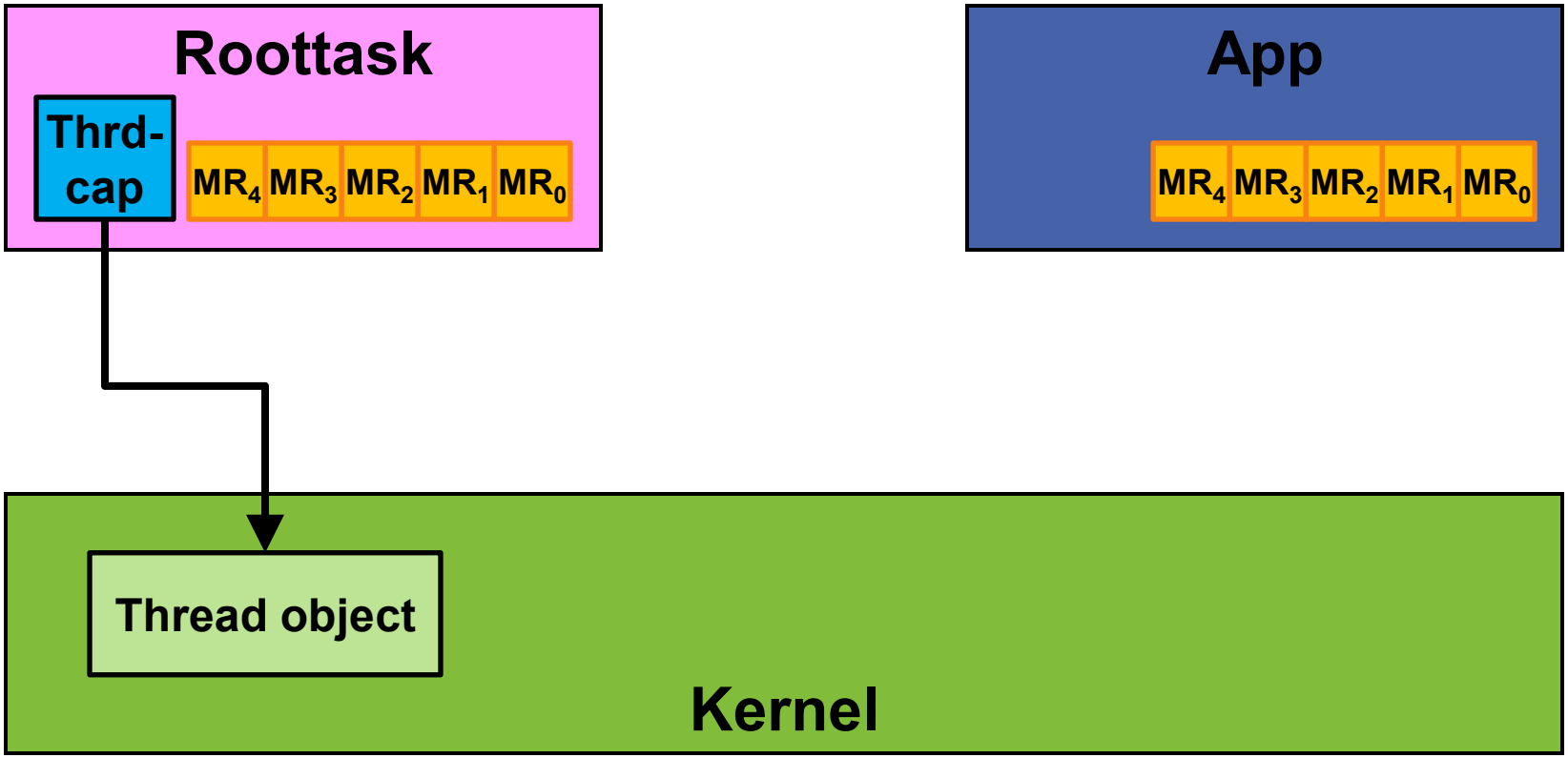
System calls with capabilities



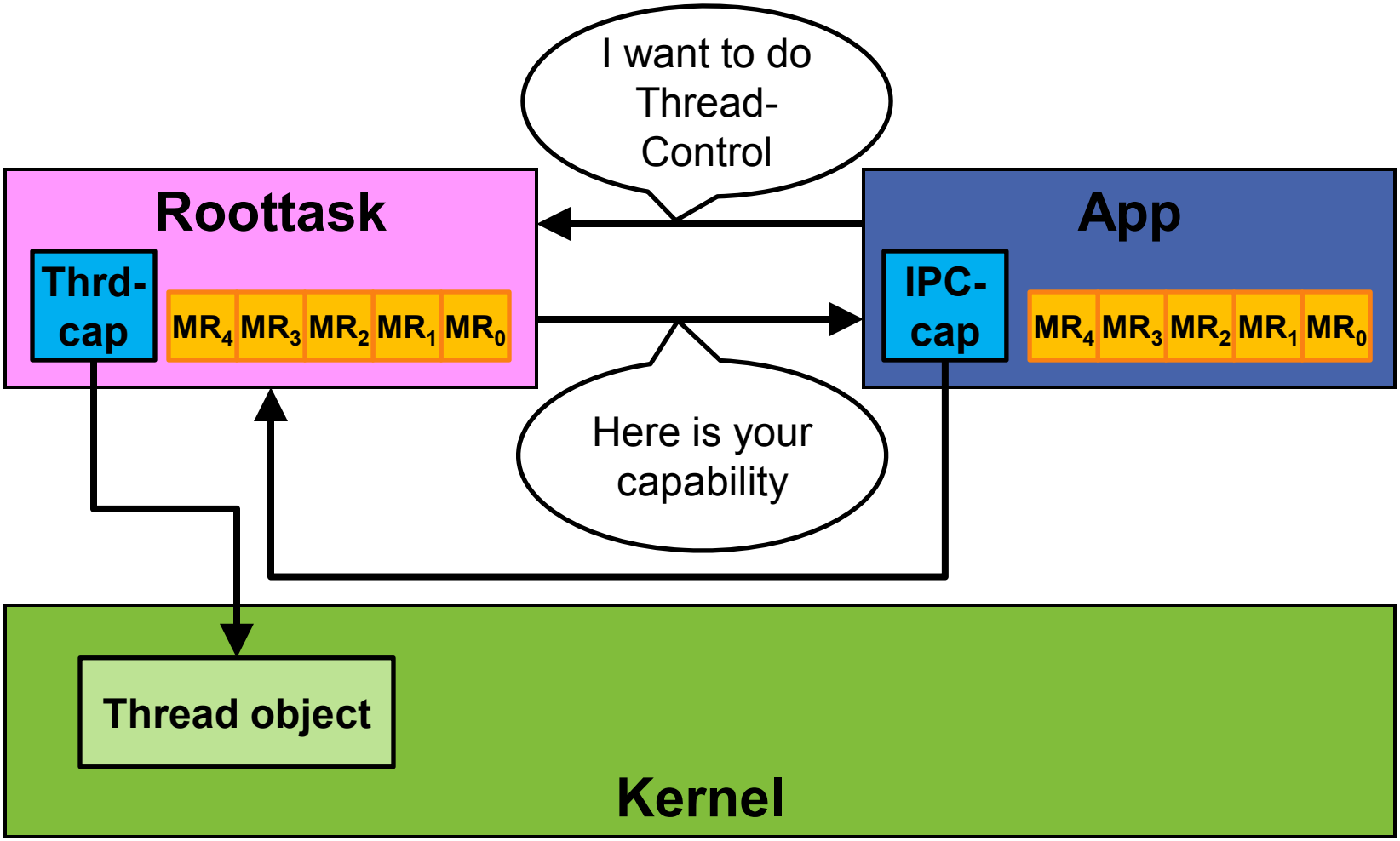
System calls with capabilities



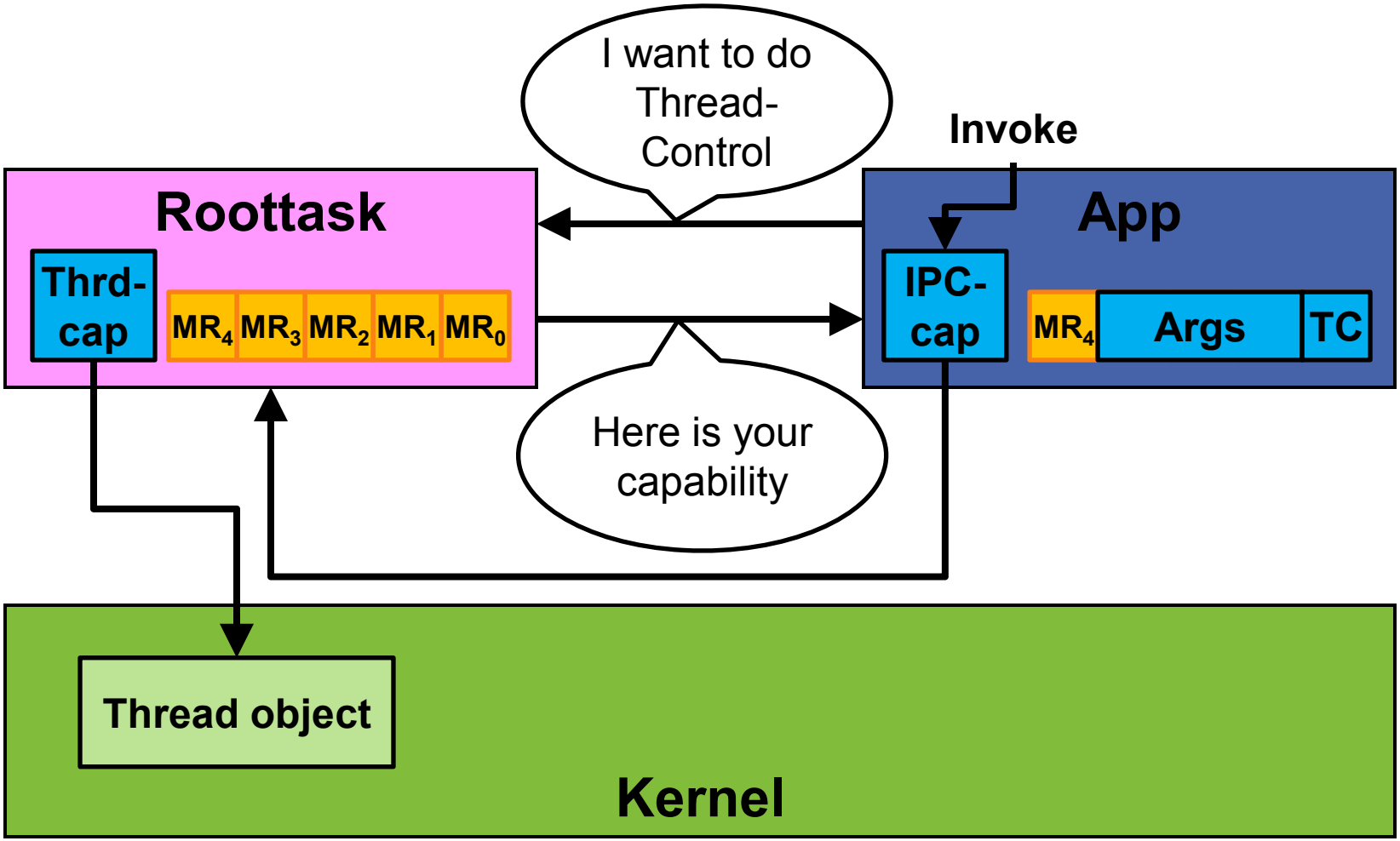
System call indirection with capabilities



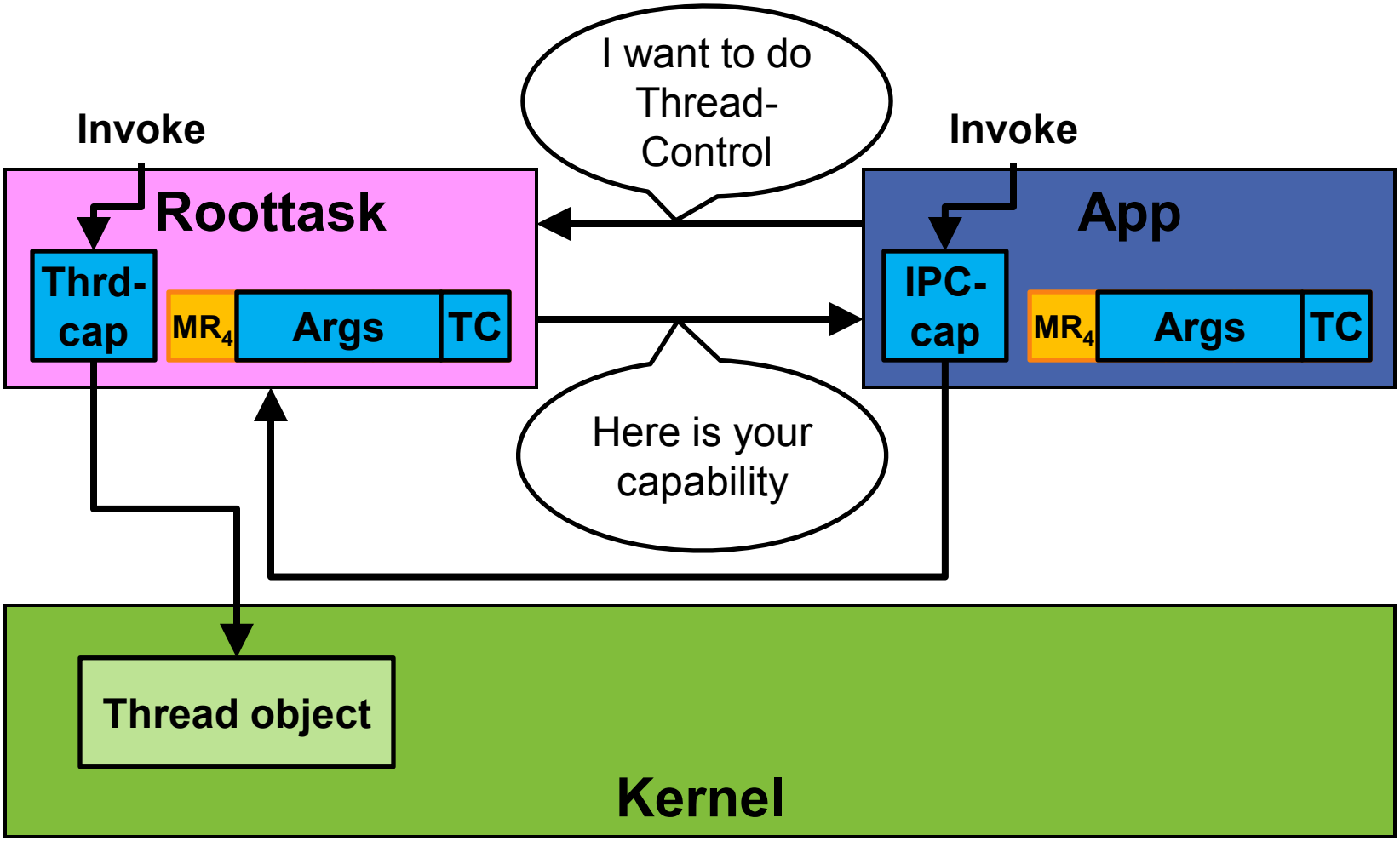
System call indirection with capabilities



System call indirection with capabilities



System call indirection with capabilities

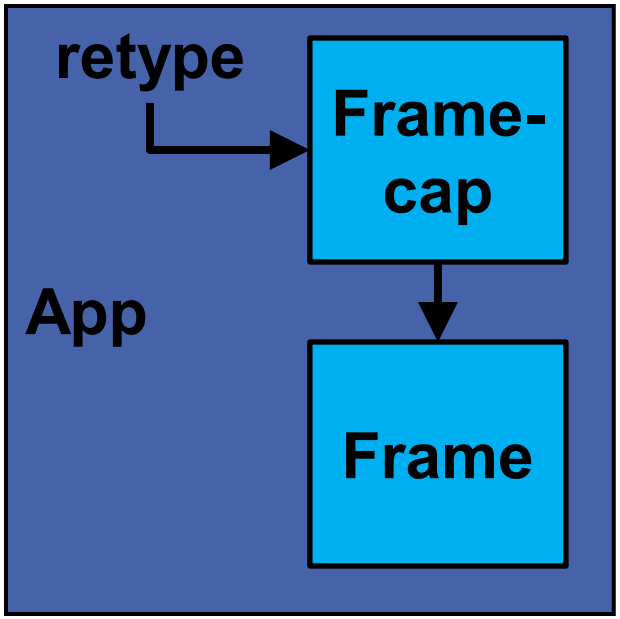


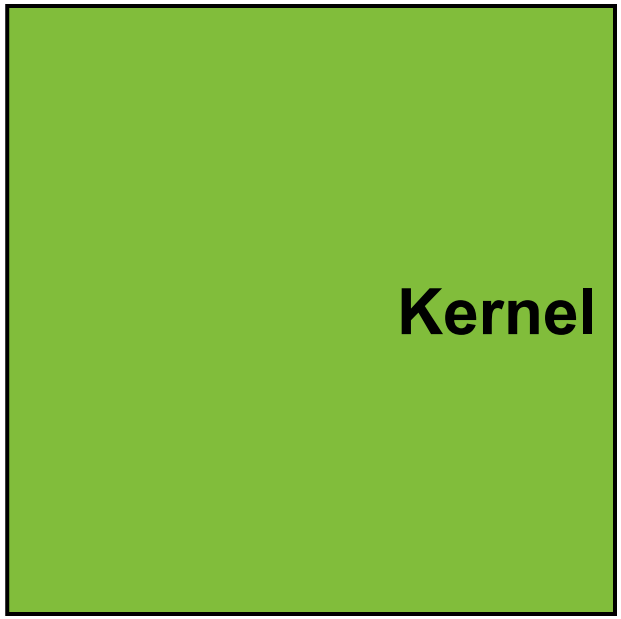
Kernel resource management

- Apps with access to some syscalls can exhaust kernel resources
- No choice but to filter every call
  - Really?
- Solution: Make apps use their own memory for syscalls!

SeL4 kernel memory

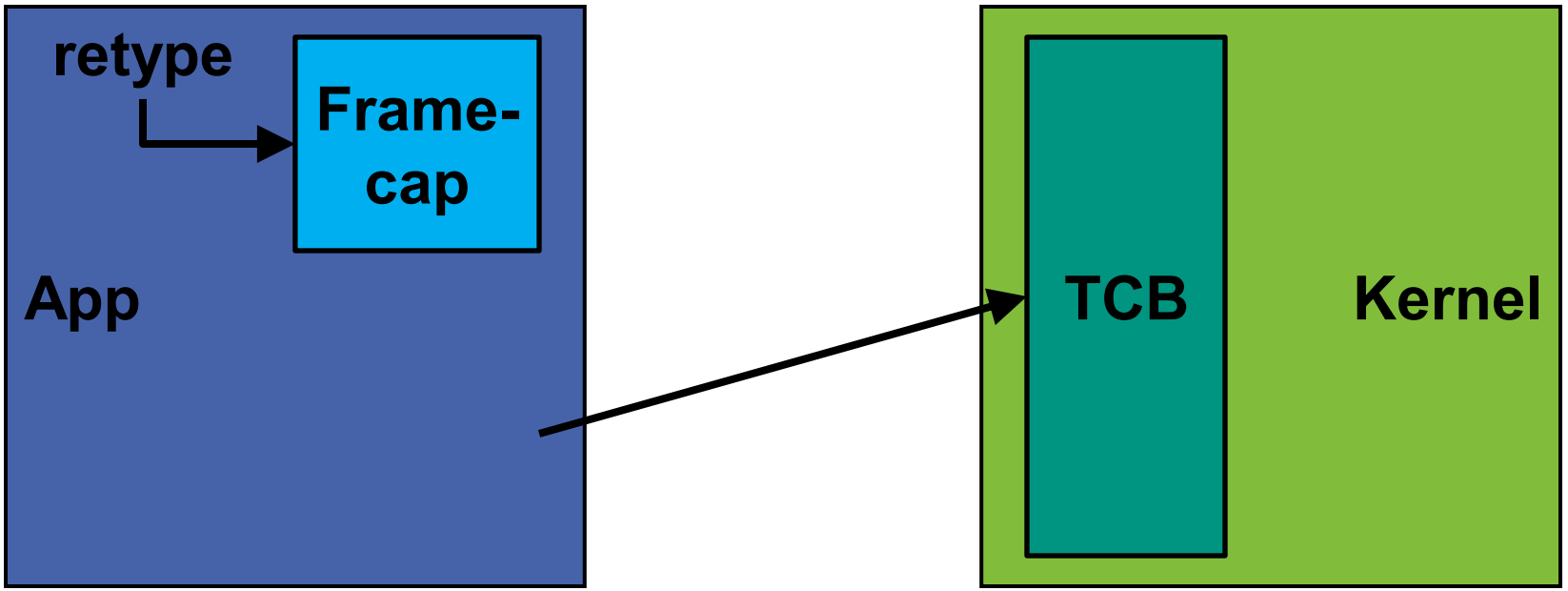
- Everything is a kernel object
- Can `retype` kernel objects





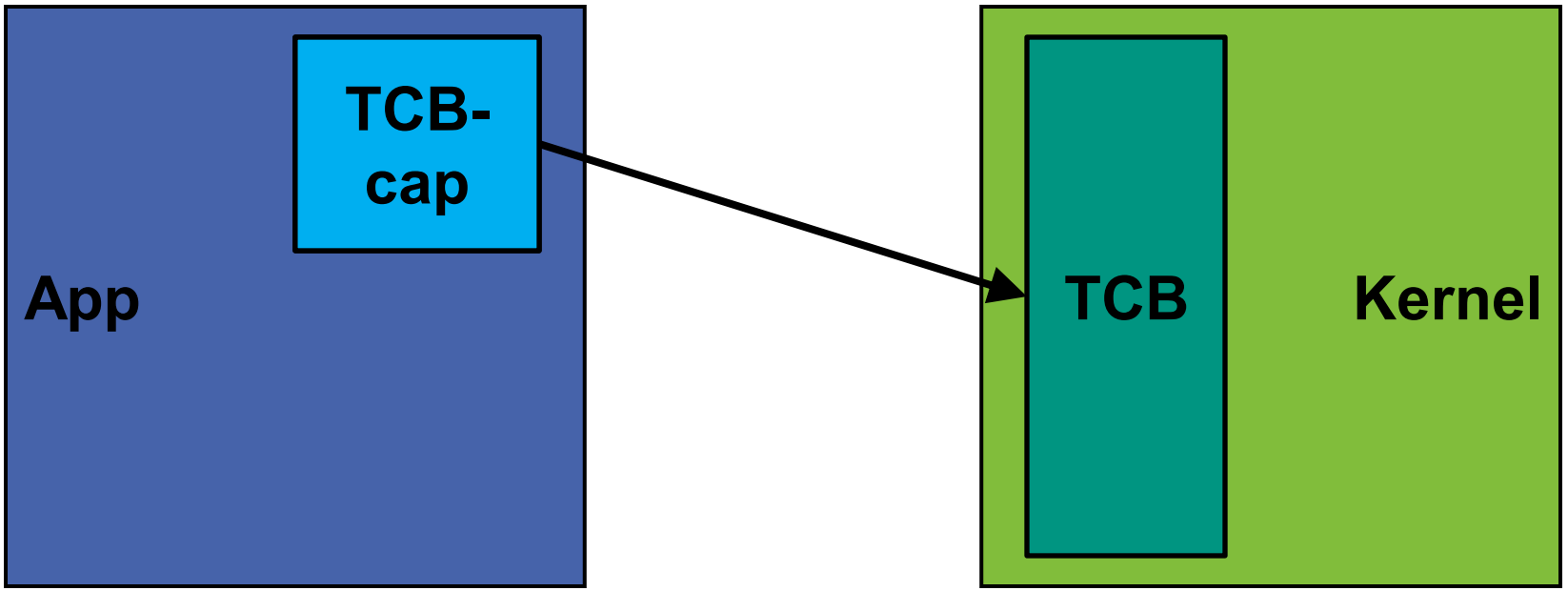
SeL4 kernel memory

- Everything is a kernel object
- Can `retype` kernel objects
- Retyping `grants` memory to kernel



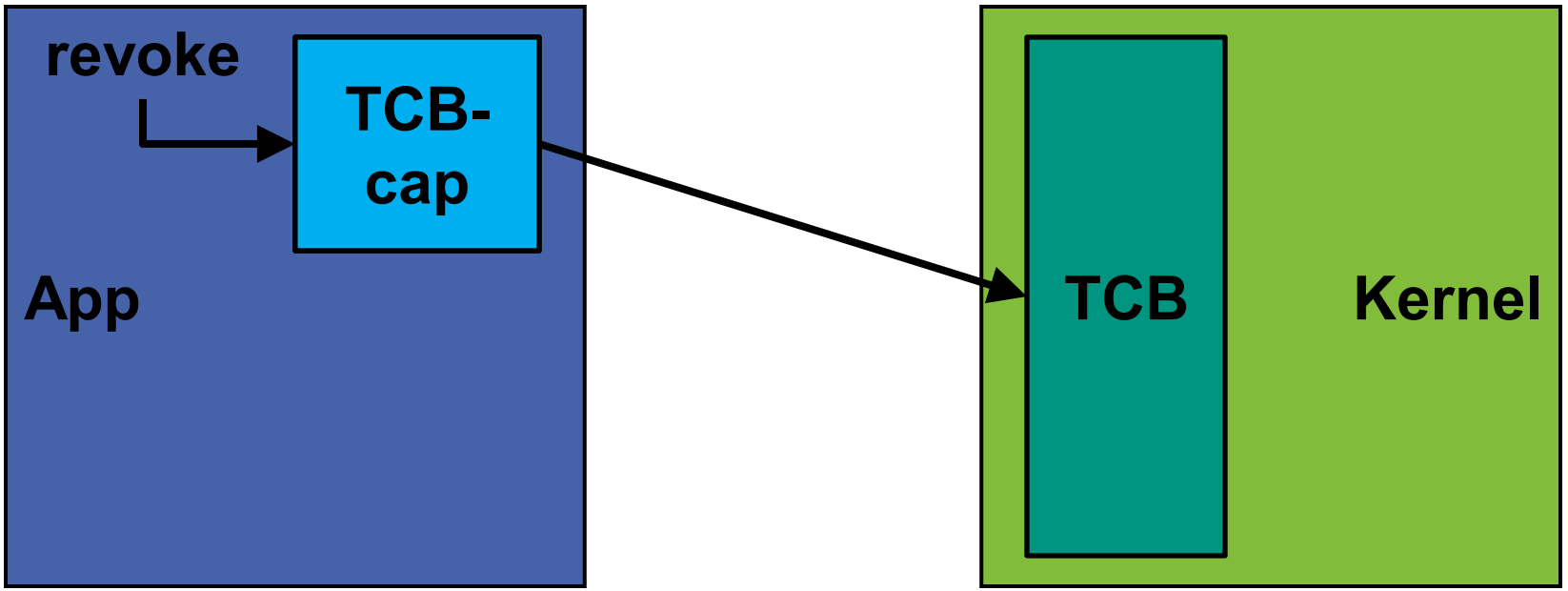
SeL4 kernel memory

- Everything is a kernel object
- Can `retype` kernel objects
- Retyping `grants` memory to kernel
  - App retains capability to retyped object



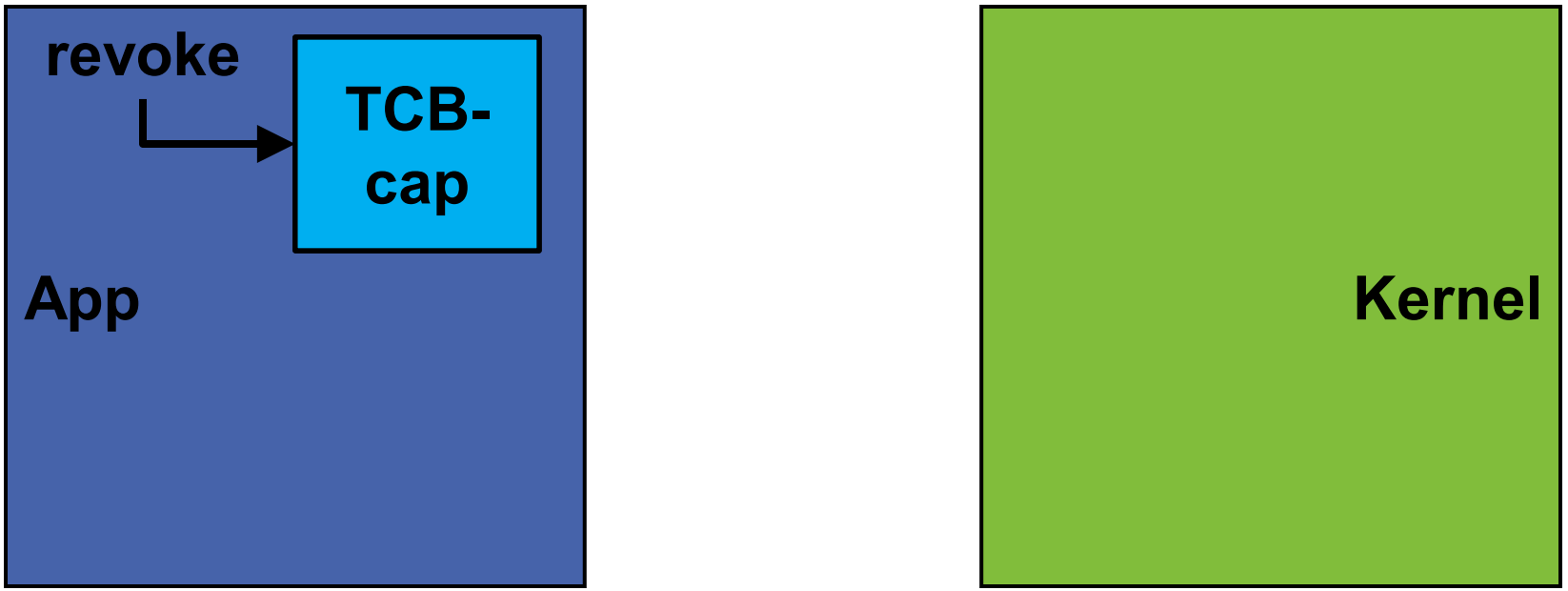
SeL4 kernel memory

- App retains capability to retyped object
- App can revoke retyping



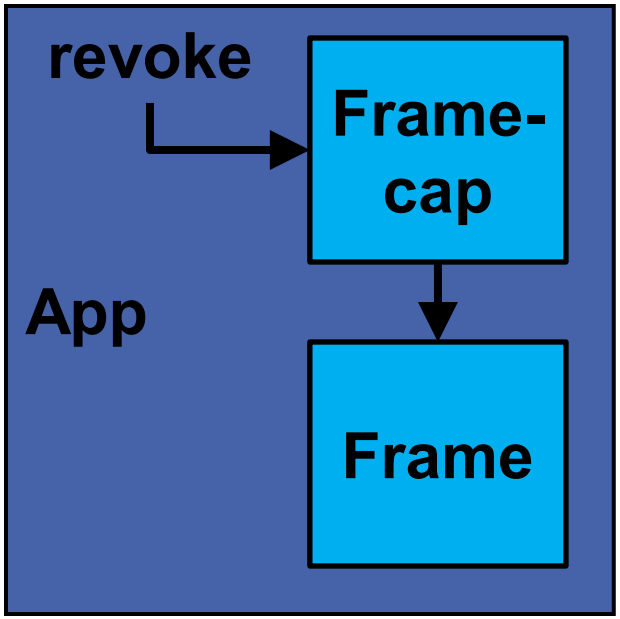
SeL4 kernel memory

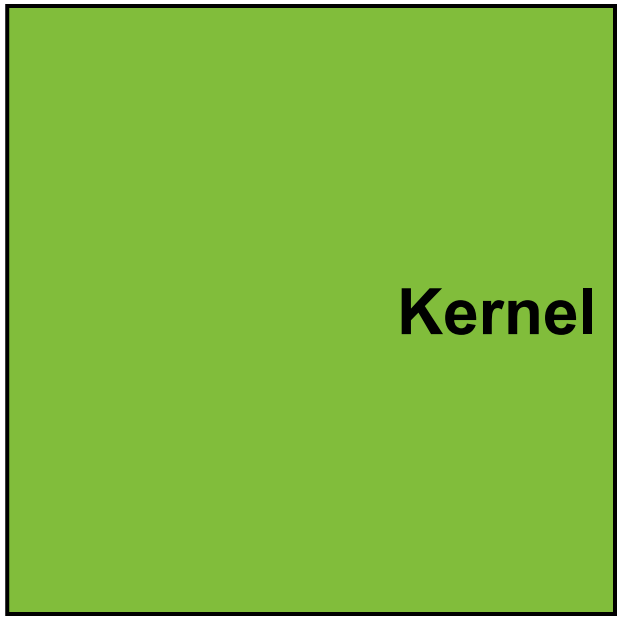
- App retains capability to retyped object
- App can `revoke` retyping
  - Destroys the referenced kernel object



SeL4 kernel memory

- App retains capability to retyped object
- App can `revoke` retyping
  - Destroys the referenced kernel object
- App can `re-use` memory





SeL4 kernel memory

- Applications must provide memory for everything they need from the kernel
  - Threads, page tables, Cap-array, IPC endpoints, ...
- Applications cannot DoS the kernel, only themselves
- No kernel memory manager needed
  - Simplifies proof!

- Kernel needs some memory before applications exist
  - E.g., code, kernel stack
  - Strictly bounded => provable
- More complex capability management
  - Must remember retype history => Capability derivation tree

Summary

- Clans & Chiefs: Static, inefficient
- Generic IPC redirection: Centralized

- Capabilities:
  - Fine-grained control
  - Decentralized management (& naming)

- Capabilities to kernel objects
  - System call indirection (everything is IPC)

- Application memory for kernel services