Microkernel

Jinyu Gu

Sept. 27, 2020

Microkernel-based Harmony OS

INTRODUCTION

The Role of Operating Systems

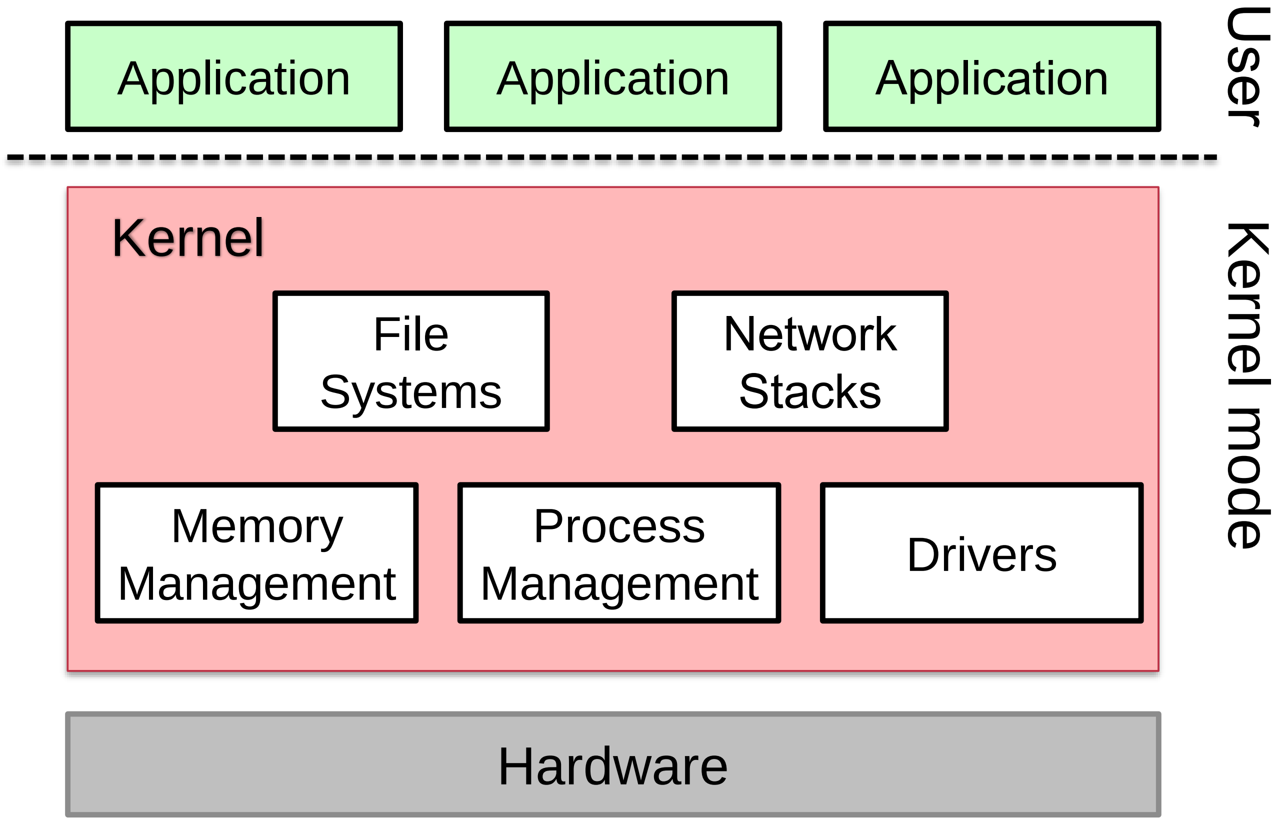
- **Manage hardware resources**
  - E.g., CPU, Memory, I/O
- **Provide ease-to-use interfaces to access resources**
  - E.g., network sockets
- **Perform privileged/HW-specific operations**
  - E.g., ring 0/3
- **Provide separation and collaboration**
  - E.g., users/processes isolation and communication

OS Architectures

Monolithic vs. Microkernel

- **Monolithic Kernel**
  - Linux
  - Windows
  - BSD

Monolithic Kernel Design



Disadvantages of Monolithic OSes

- **System components run in privileged mode**
  - No protection between system components
    - Security issues: e.g., faulty driver
  - No isolation between system components
    - Resilience issues: e.g., crash together
  - Big and inflexible
    - Difficult to replace system components
    - Difficult to understand and maintain

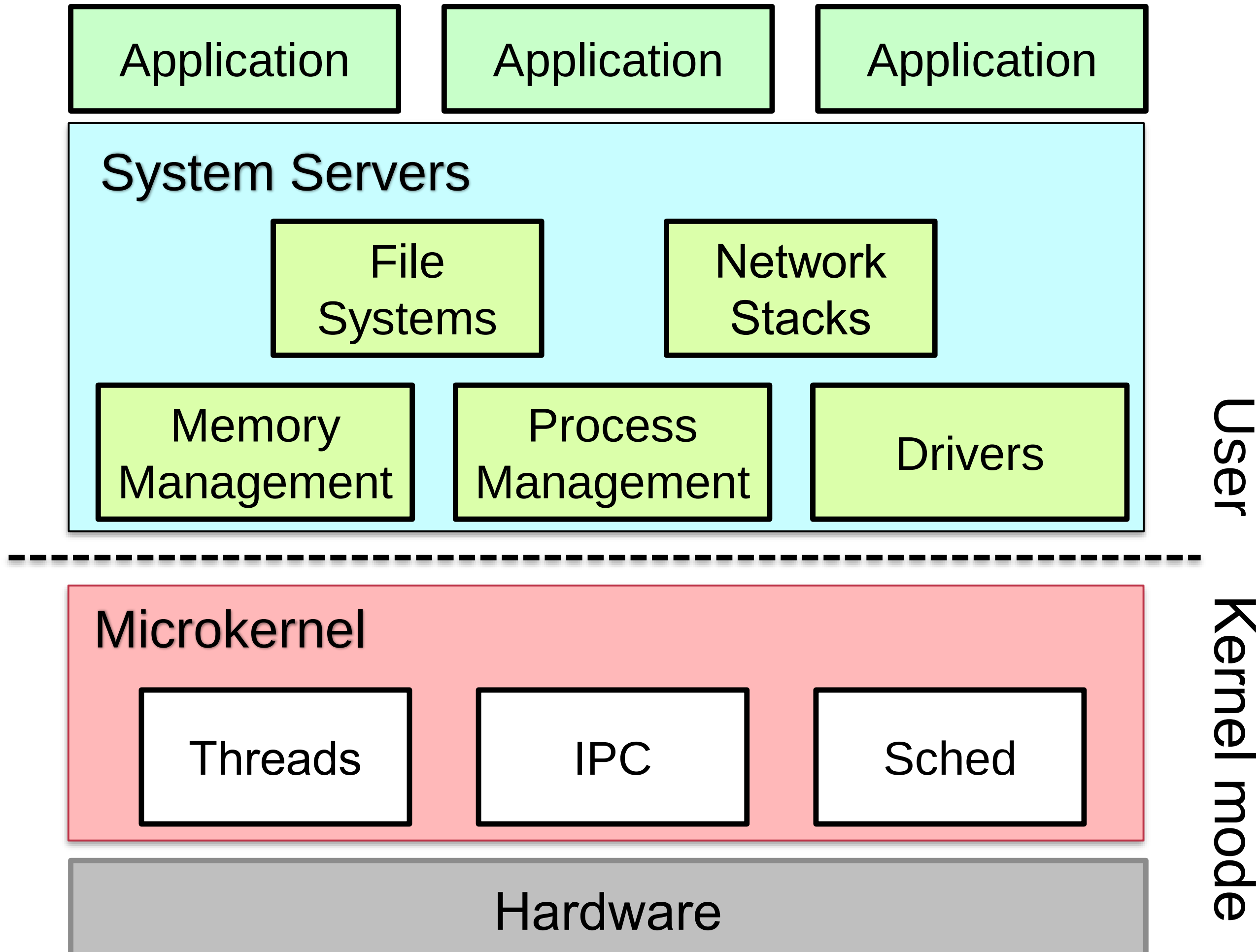
Monolithic vs. Microkernel

- **Monolithic Kernel**
  - Linux
  - Windows
  - MacOS
- **Microkernel**
  - QNX (car)
  - Fuchsia/Zircon (Google)
  - Fiasco (L4)
  - Harmony (mobile, TV, car)
  - RT-Thread Smart (IoT)

The Microkernel Vision

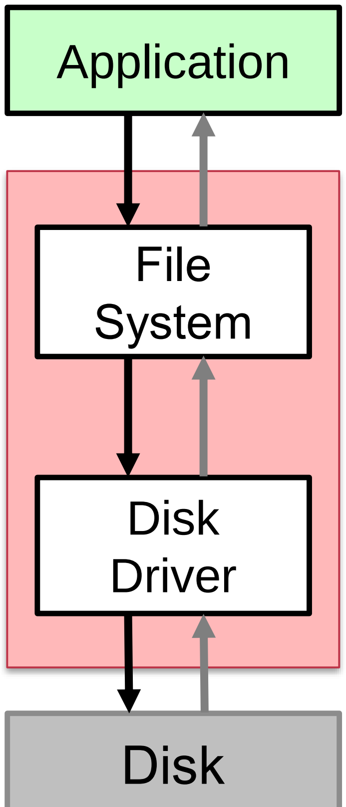
- **Minimal OS kernel**
  - Small code size
  - Suitable for verification
- **System services in user-level servers**
  - Flexible and extensible
- **Isolation between individual components**
  - More resilient (no crash whole)
  - More secure (inter-component isolation)

Microkernel Design



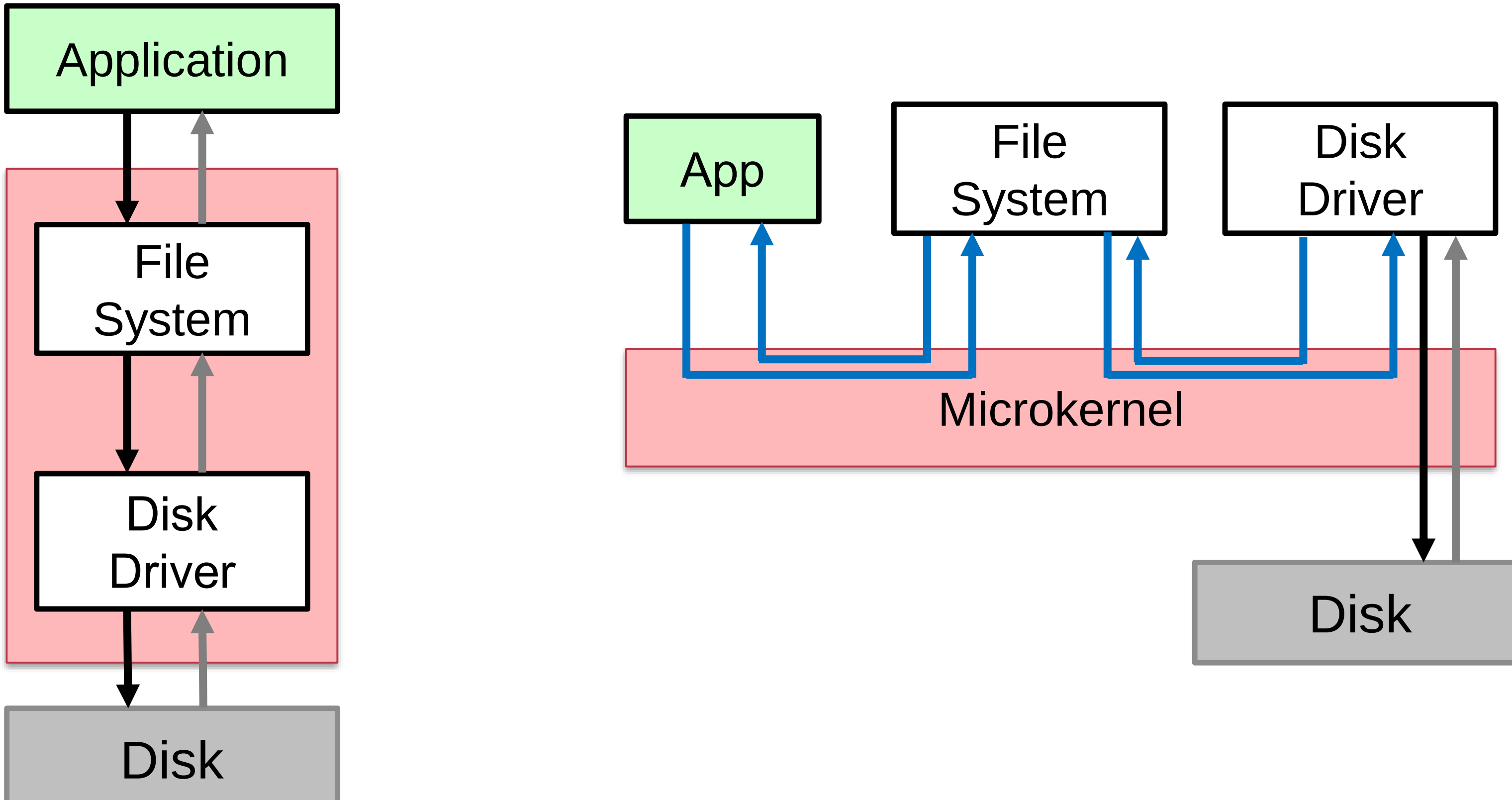
Interaction Becomes Different

Example: file operations



Interaction Becomes Different

Example: file operations



From the View of Developers

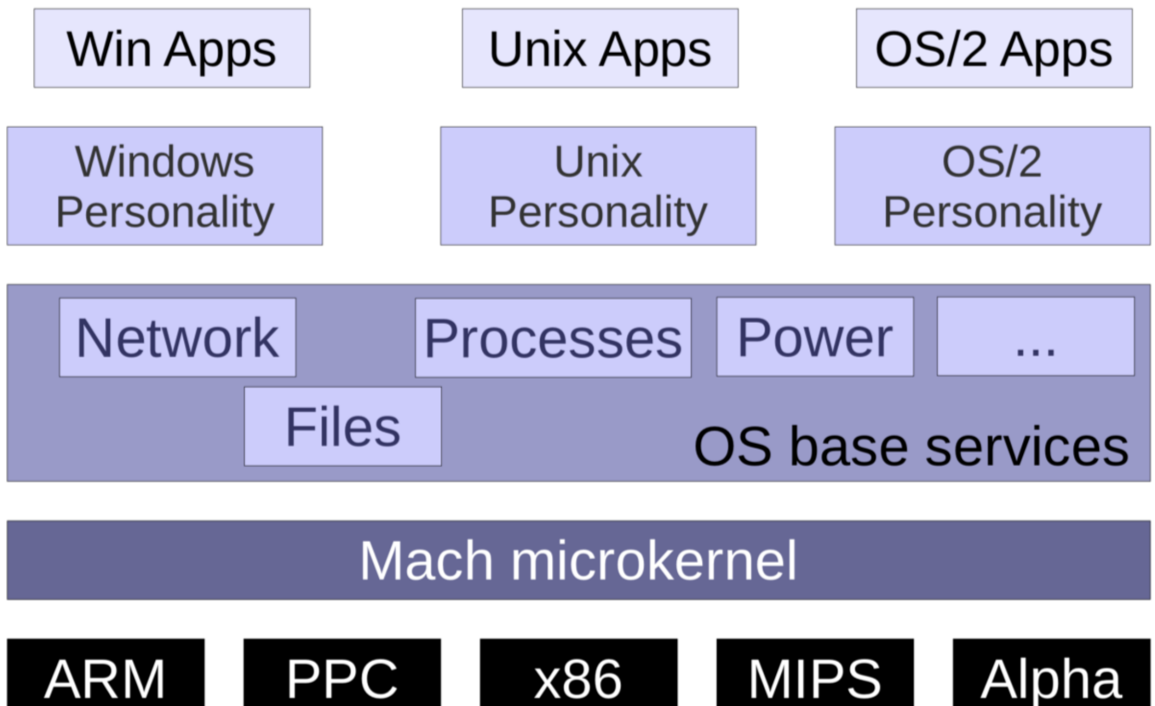
- **Customizability**
  - Configure suitable servers (embedded, desktop …)
  - Remove unneeded servers
- **Enforce reasonable system design**
  - Well-defined interfaces between components
  - No access to components besides the interfaces
  - Improved maintainability

Mach: The First Microkernel

- **Mach – developed at CMU, 1985 – 1994**
  - Rick Rashid (former head of MS Research)
  - Avie Tevanian (former Apple CTO)
  - Brian Bershad (professor @ U. of Washington)
- **Foundation for several real systems**
  - IBM Workplace OS
  - Next OS --> Mac OS X

IBM Workplace OS

- Main goals: 1. multiple OS personalities
  2. run on multiple HW architectures



Lessons Learned

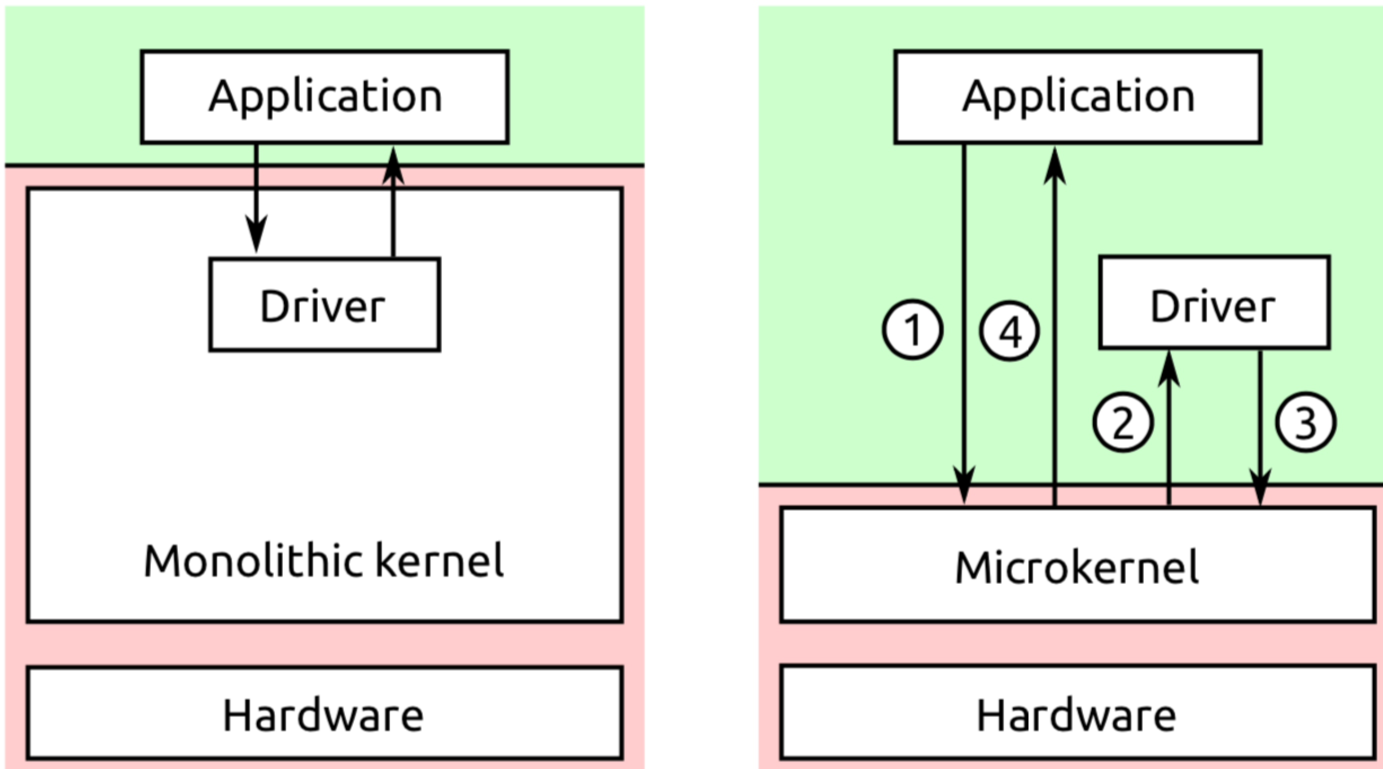
- Microkernel worked, but system atop it did not
- Underestimated difficulties in creating OS personalities
- ✔ subsystem protection/isolation
- ✔ small code size
- ✔ customizability:
  - Tailored memory management/scheduling/… algorithms
  - Adaptable to embedded/real-time/secure/… systems

Review: Monolithic vs. Microkernel

- Flexibility and Customizable
  - Monolithic kernels are typically modular
- Maintainability and complexity
  - Monolithic kernels have layered architecture
- **Robustness**
  - Isolation & Trusted code size
- **Performance**
  - Application performance degraded

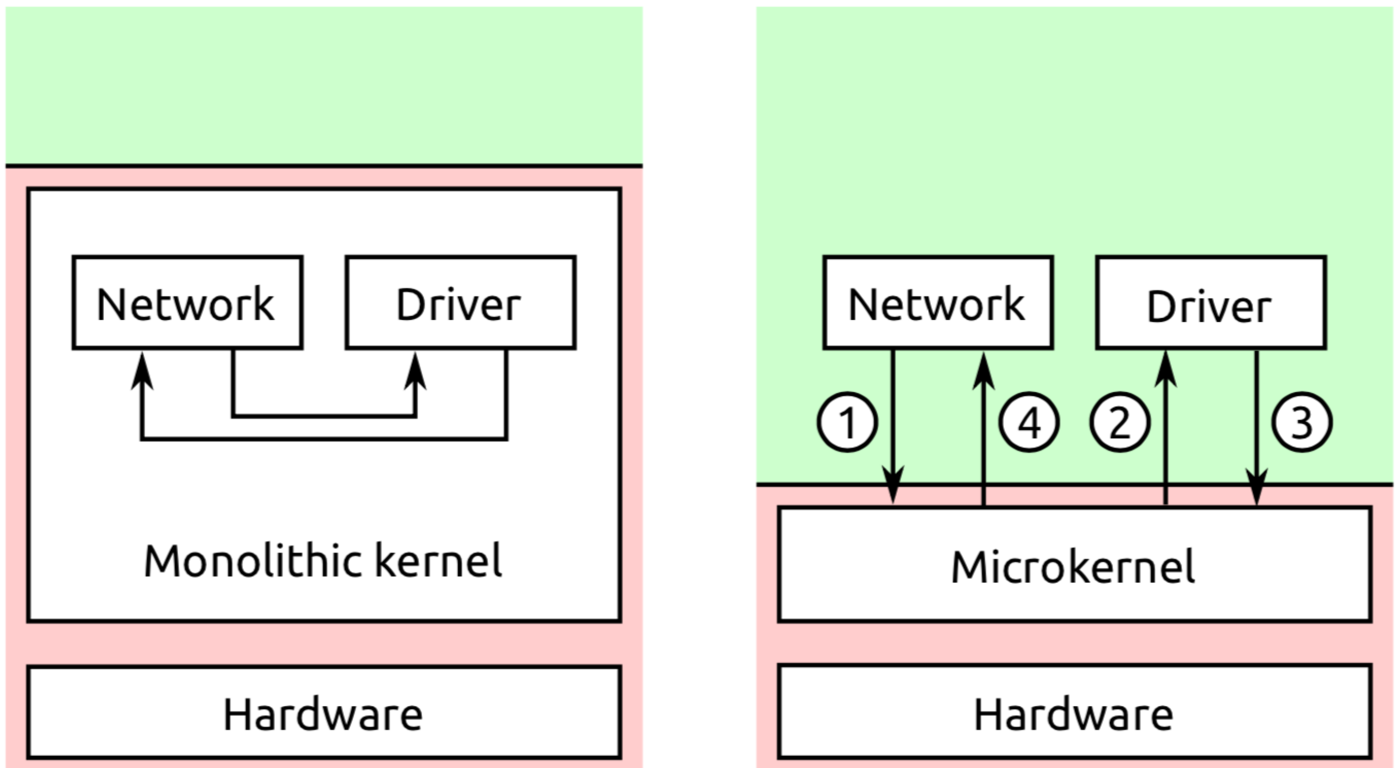
Monolithic vs. Microkernel: Syscalls

- Monolithic kernel: 2 kernel entries/exits
- Microkernel: 4 kernel entries/exits + 2 context switches



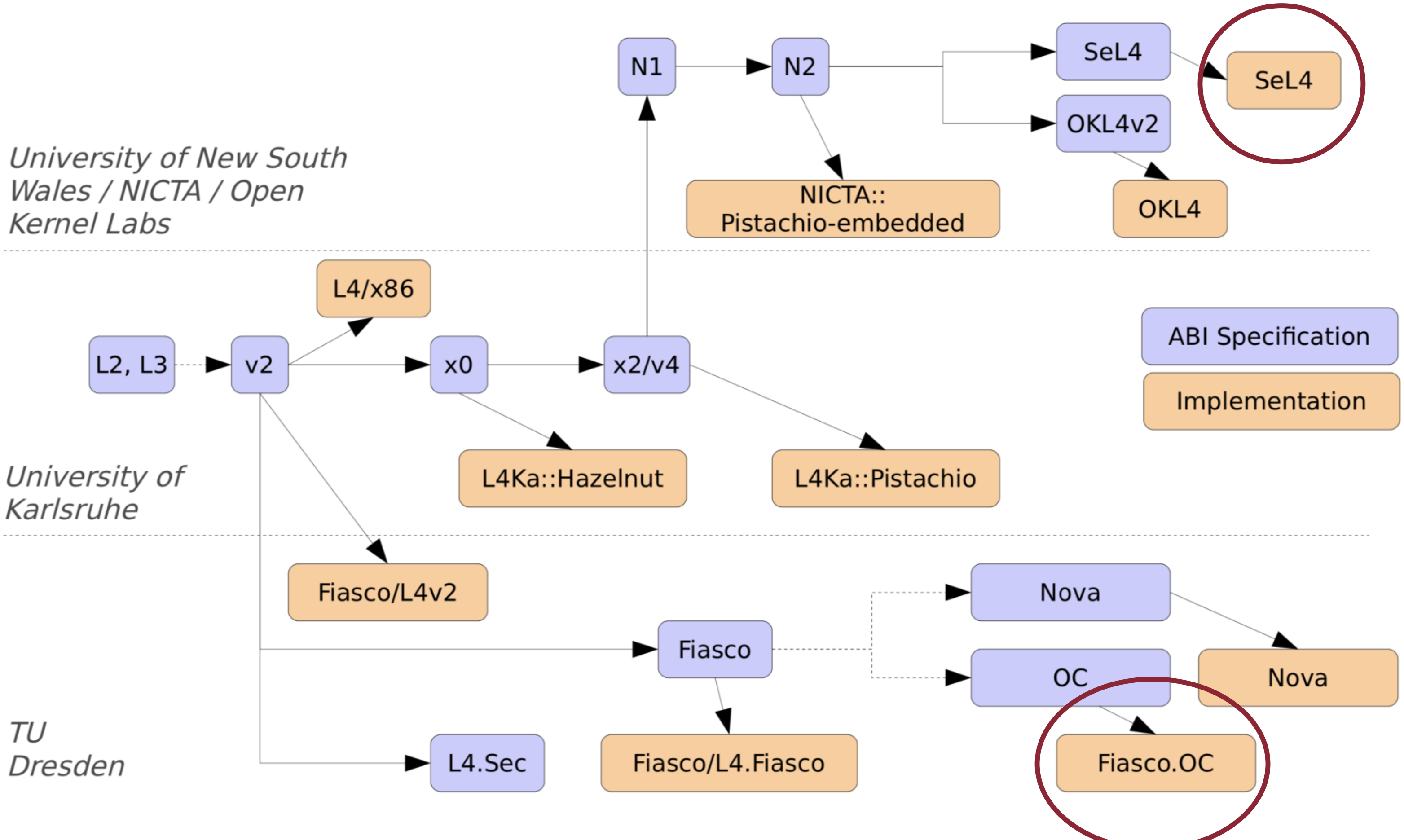
Calls Between Services

- Monolithic kernel: 2 function calls/returns
- Microkernel: 4 kernel entries/exits + 2 context switches



WHAT DO MICROKERNELS HAVE

L4 Family Microkernel



L4 Concepts

- Jochen Liedtke: "A microkernel does no real work"
  - Kernel only provides inevitable mechanisms
  - Kernel does not provide policies
  - Example: virtual memory management
    - Write page tables (mechanism)
    - How to write (policy)

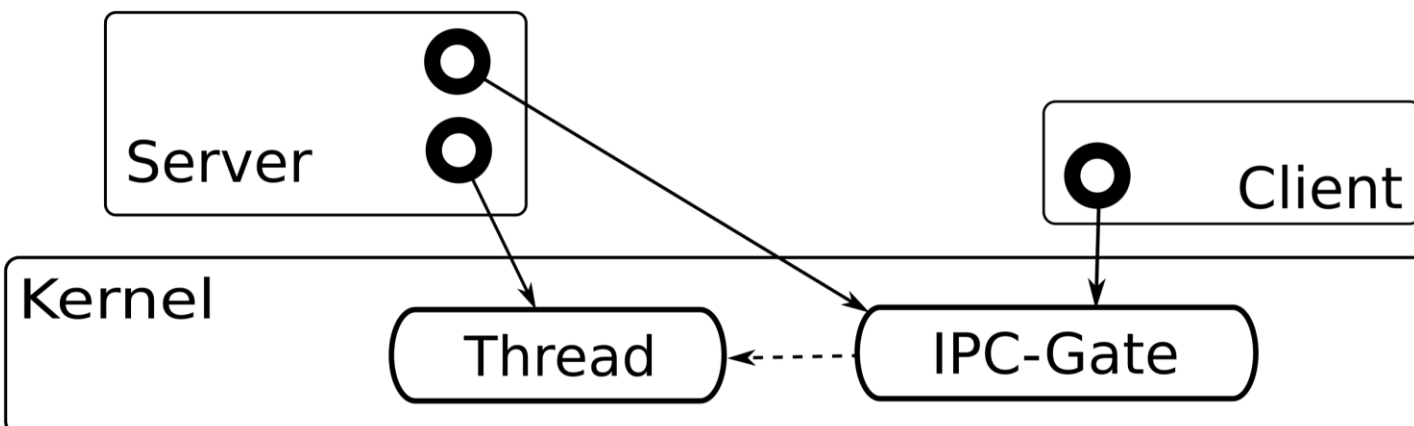
- **What things are inevitable?**

Case Study: L4/Fiasco.OC

- **"Everything is an object"**
  - Task — Address space
  - Thread — Scheduling
  - IRQ — Asynchronous notification
  - IPC Gate — Communication
- **One system call: invoke_object()**

L4/Fiasco.OC: IPC Gate

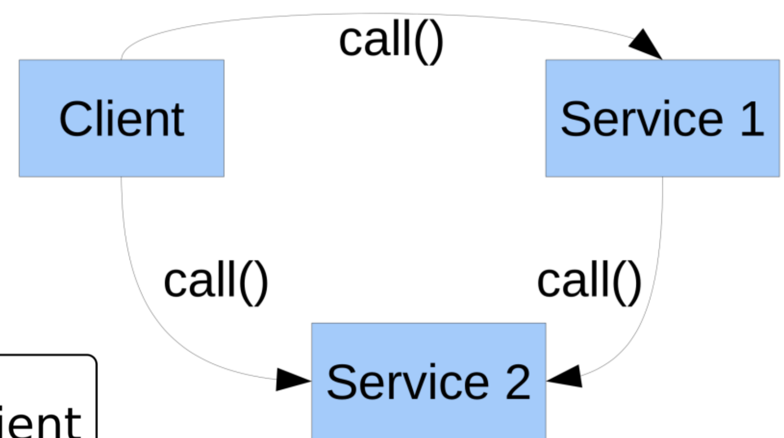
- **Generic communication object: IPC gate**
  - Send message from sender to receiver
  - Used to provide services in user-level applications
- **Example:**

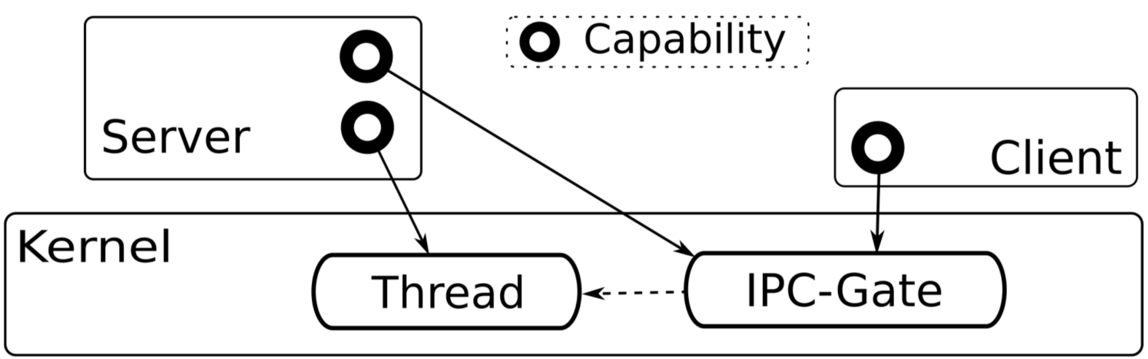


L4/Fiasco.OC: IPC Gate

- **Everything above kernel is built using IPC gates that can provide different services**
  - Networking stack
  - file system
  - User Database
  - ...

Policy: defined by user-level

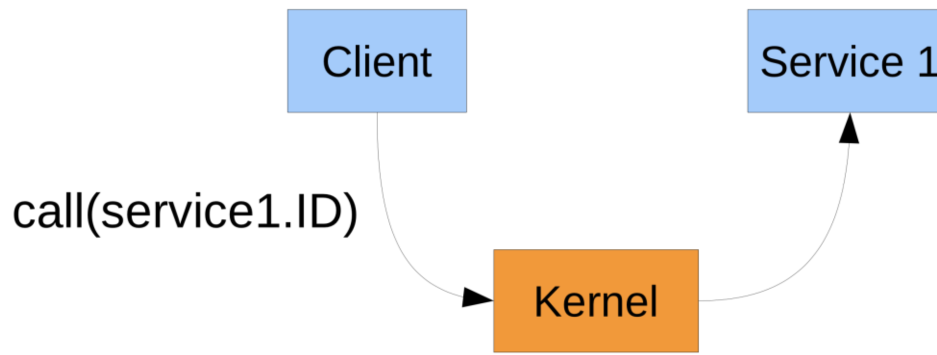




Mechanism: IPC gate

L4/Fiasco.OC: How to Call Objects?

- **Recap: invoke_object()**
- **To call an object, we need an identifier:**
  - Telephone number
  - Postal address
  - IP address
  - …



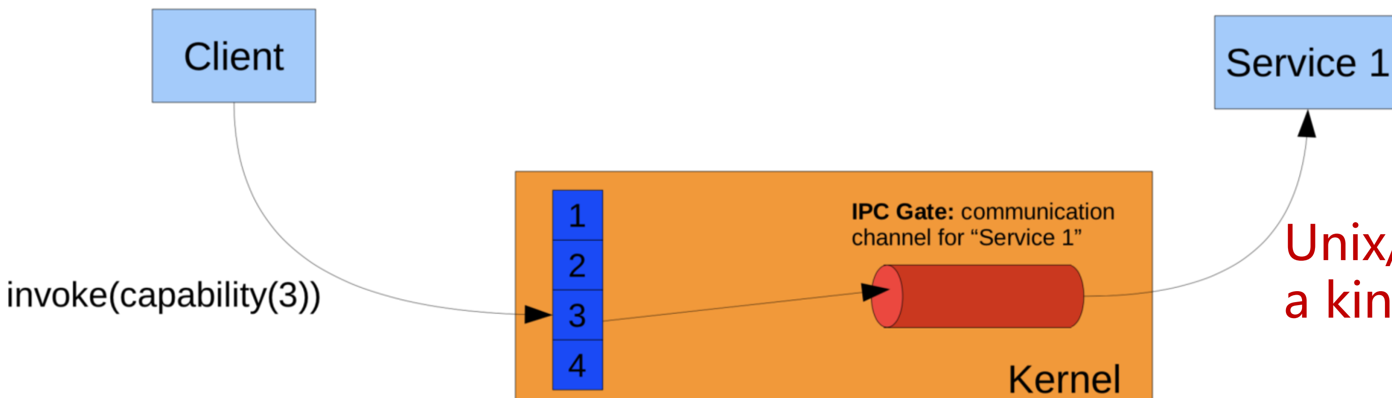
- **Simple idea, right?**
- **ID is wrong? Kernel returns ENOTEXIST**
- **This scheme is insecure!**
  - Client could simply guess IDs (brute-force)

L4/Fiasco.OC: Local Names for Objects

- **Using global object IDs**
  - Insecure (forgery)
  - Inconvenient (programmers needs to know about partitioning in advance)
- **Solution in Fiasco.OC**
  - Task-local **capability space** as an indirection
  - **Object capability** required to invoke the object
- **Per-task namespace**
  - Get object capabilities at start time (set by the farther task)
  - Get object capabilities at runtime (grant by other tasks)

L4/Fiasco.OC: Object Capabilities

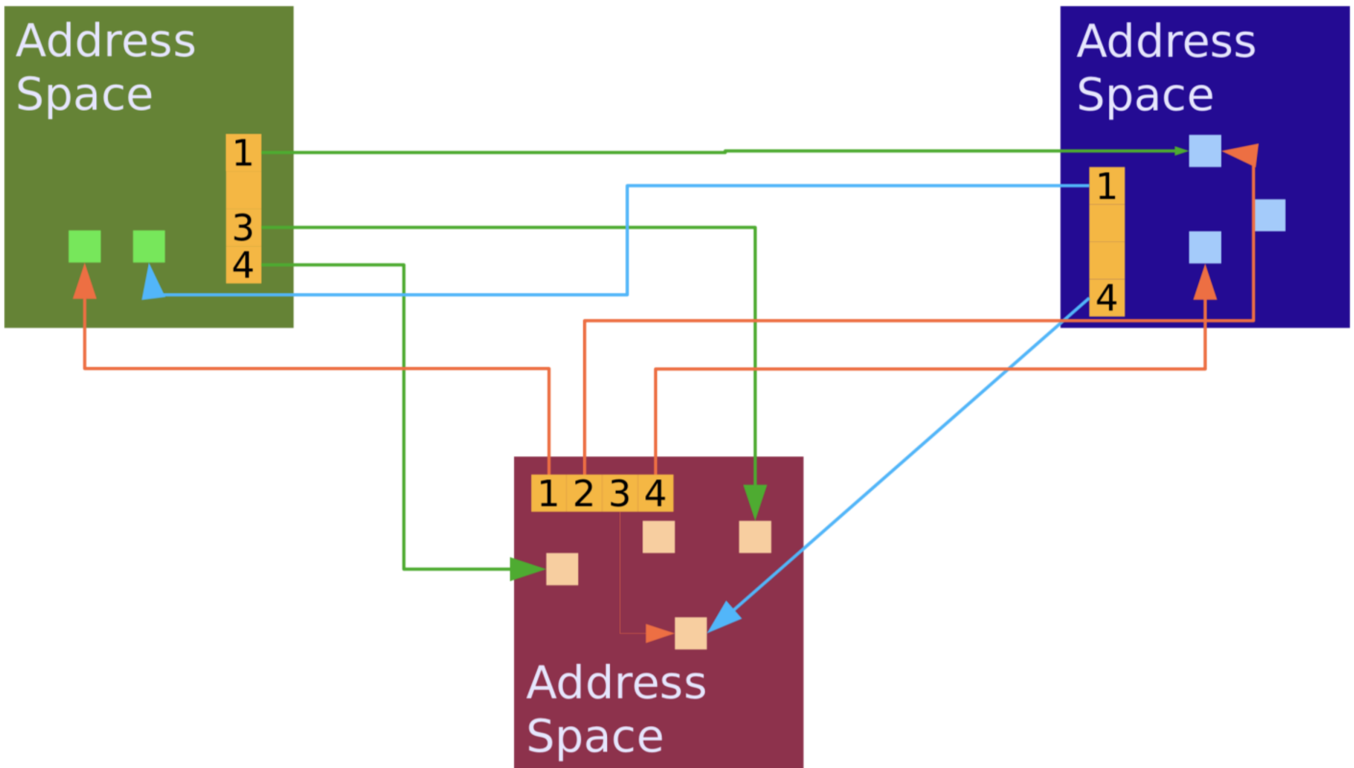
- **Capability:**
  - Reference to an object
  - Protected by the kernel
    - Kernel knows all capability-object mappings
    - Managed as a per-process capability table
    - User processes only use indexes into the table



Unix/Linux fd is just a kind of capability.

L4/Fiasco.OC: Communication

- **Indirection allows for security and flexibility**



GO DEEPER INTO DETAILS

Abstraction: Thread

- An independent flow of control inside an address space
- Communicates with other threads using IPC
- Characterized by a set of registers and the thread state
- Dispatched by the kernel according to a defined schedule

Implementation in L4/Fiasco.OC

- **Execution Context**
  - Register state
  - Address Space
  - FPU state
  - Continuation
  - Msg buffer
- **Scheduling Context**
  - Priority
  - Budget
  - Remaining Budget
  - Prev/next pointer

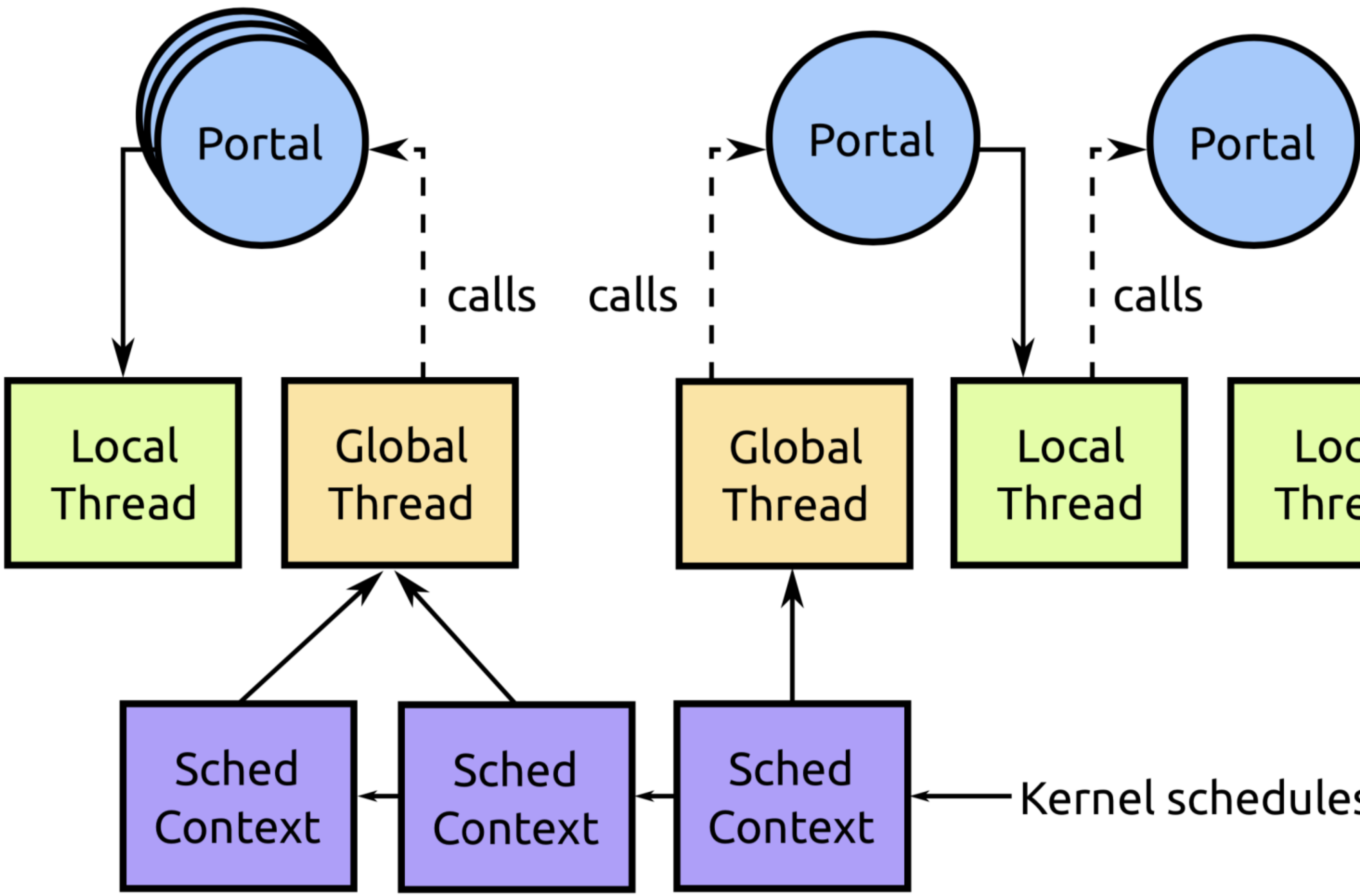
Thread Variants

- **Global Thread**
  - Needs a scheduling context， i.e., CPU time, to execute
  - Causes exception on startup to let creator set register state
- **Local Thread (Passive Thread)**
  - Has no scheduling context
  - Only used to handle portal calls
  - Waits in the kernel until someone called an associated portal

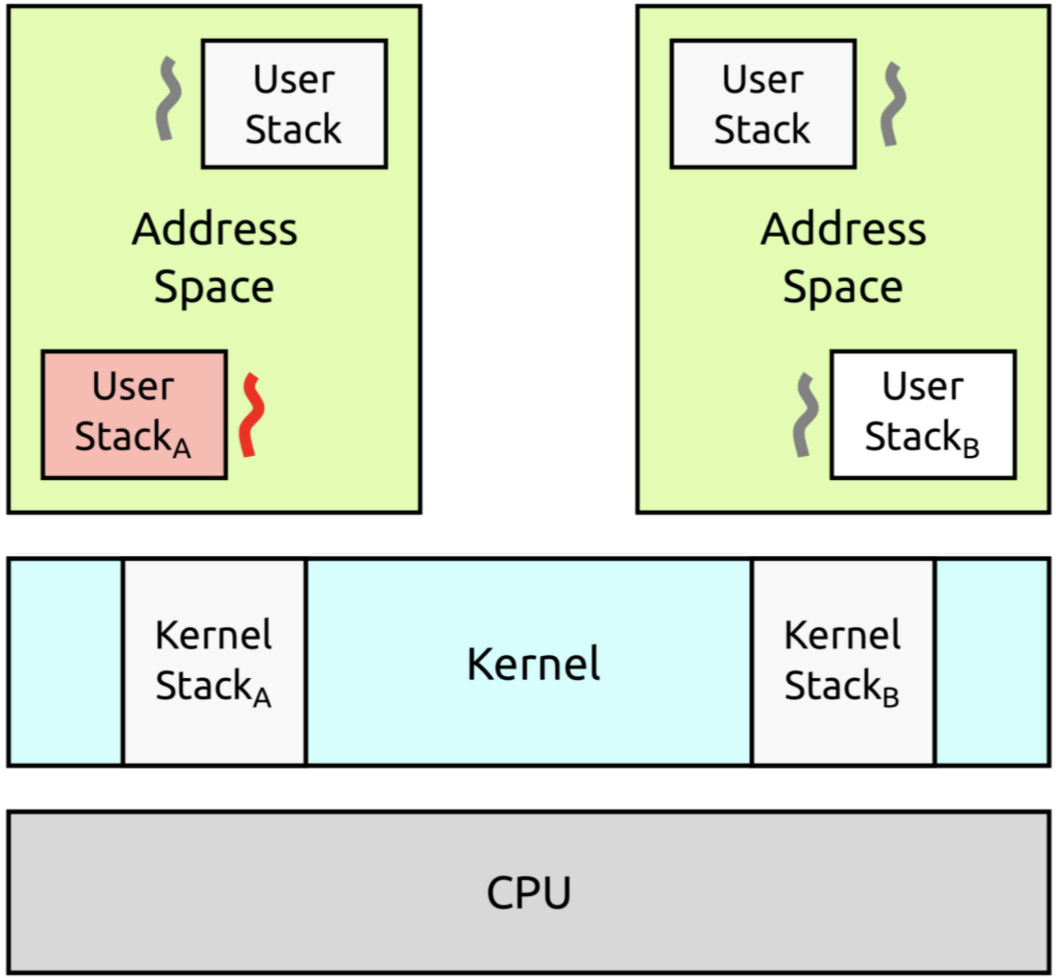
Portals

- **A portal is an IPC gate mentioned before**
  - Called via system call
- **Associated with local threads**
  - CPU time is donated from caller
  - Message is transferred from sender msg buffer to receiver msg buffer

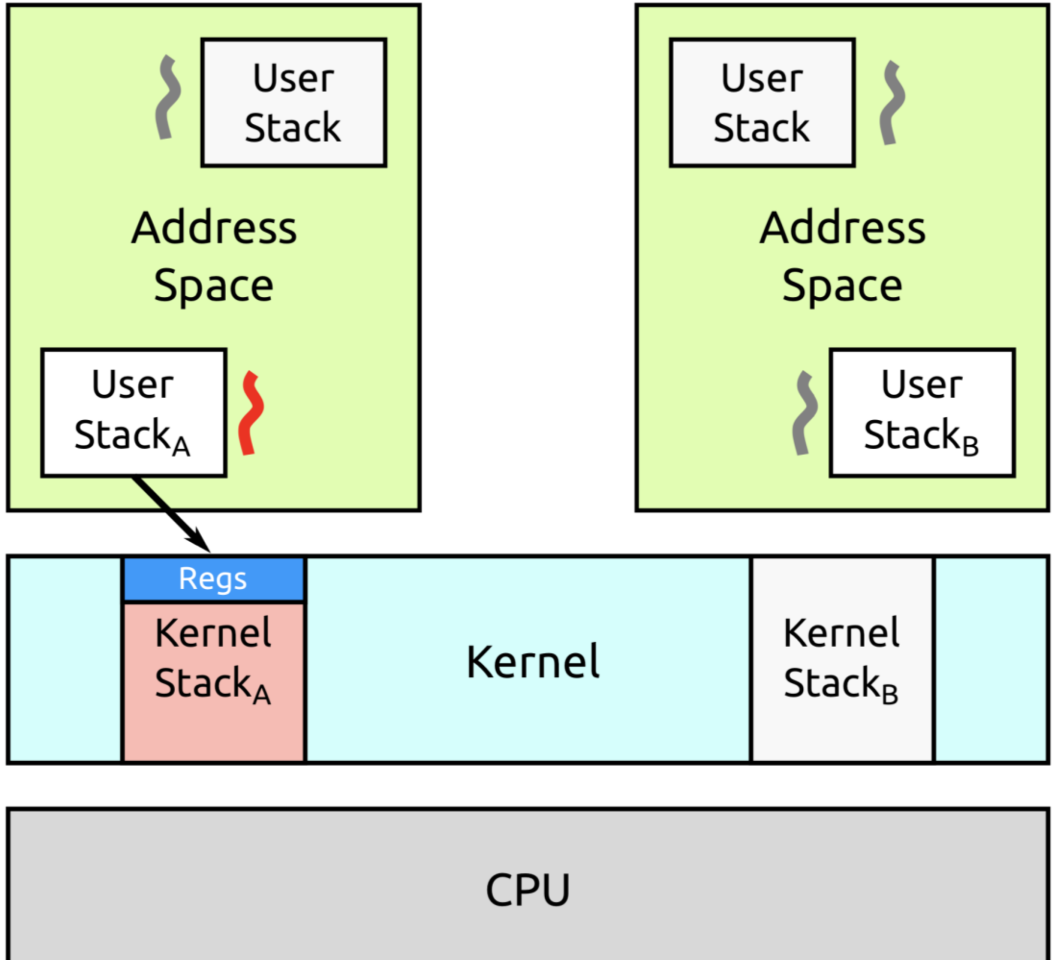
Thread Scheduling Overview



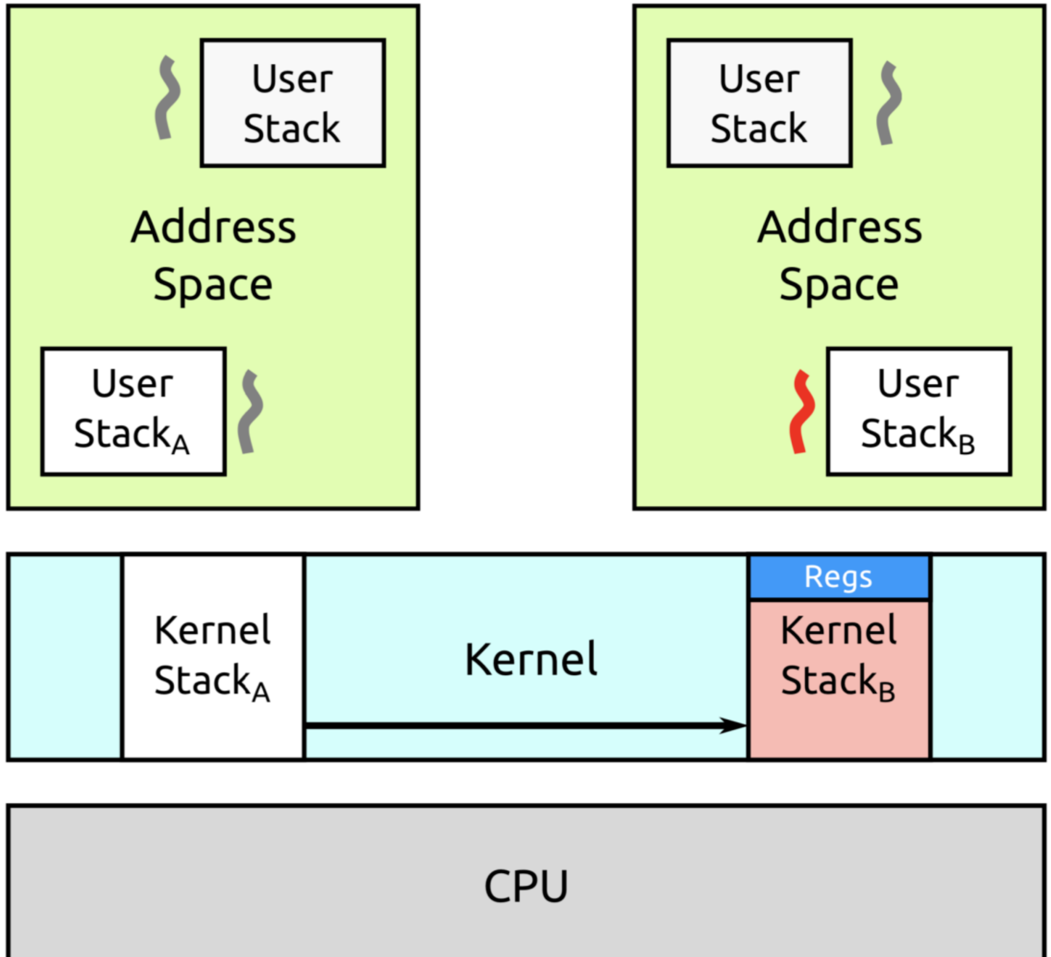
Thread Switch: Conventional



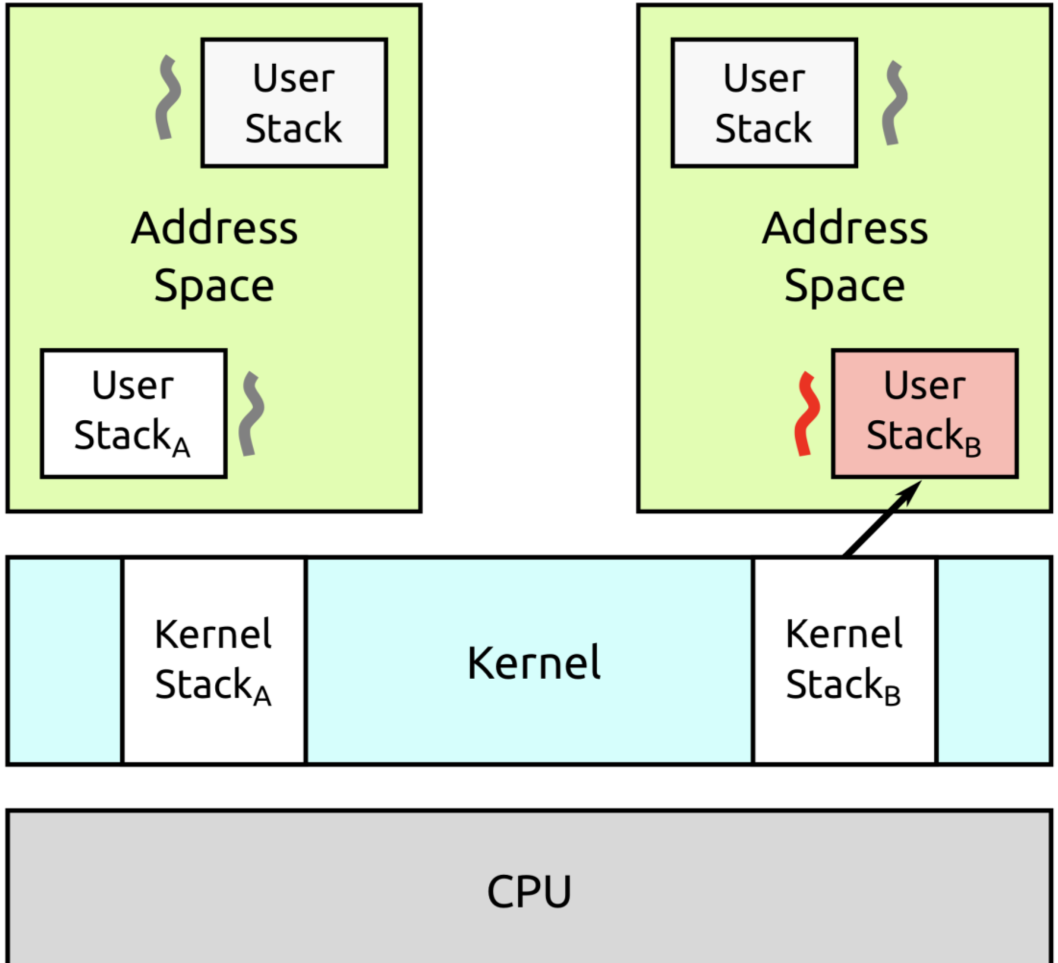
Thread Switch: Conventional



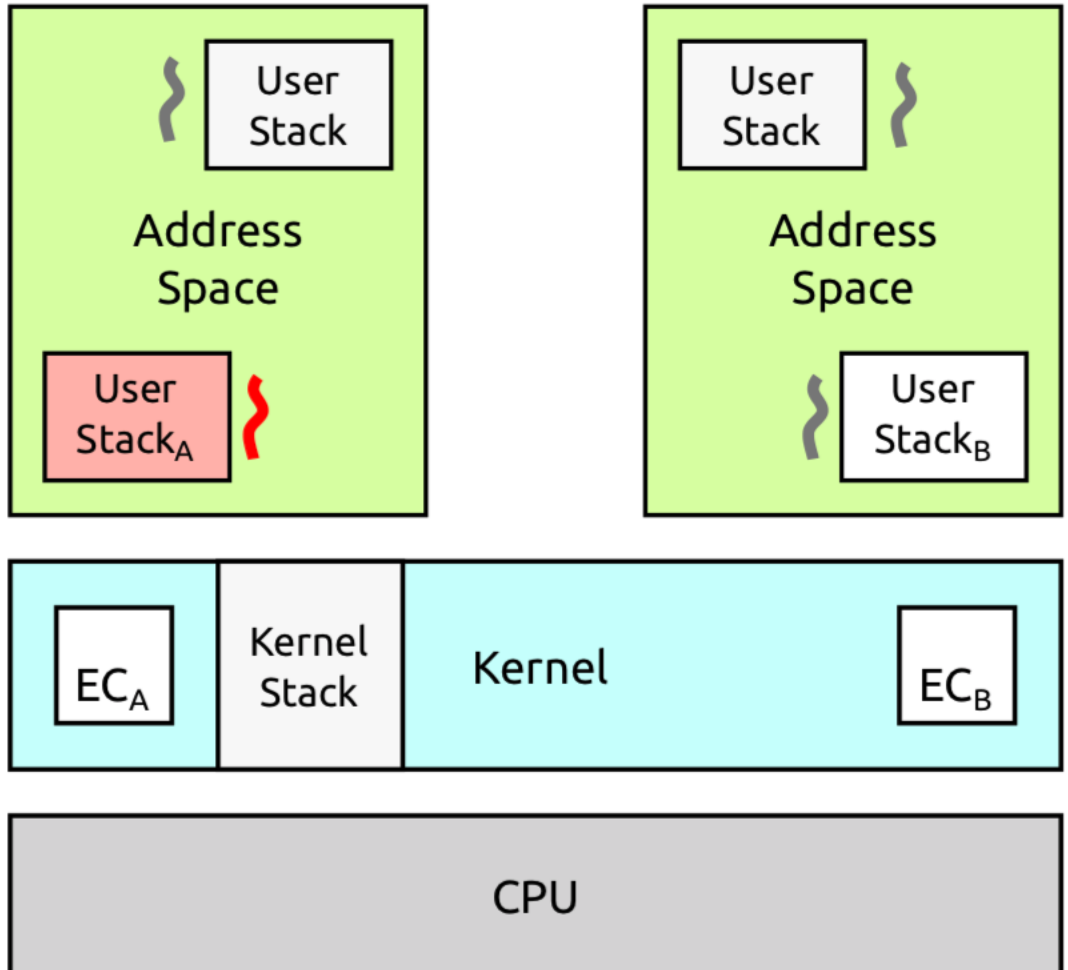
Thread Switch: Conventional



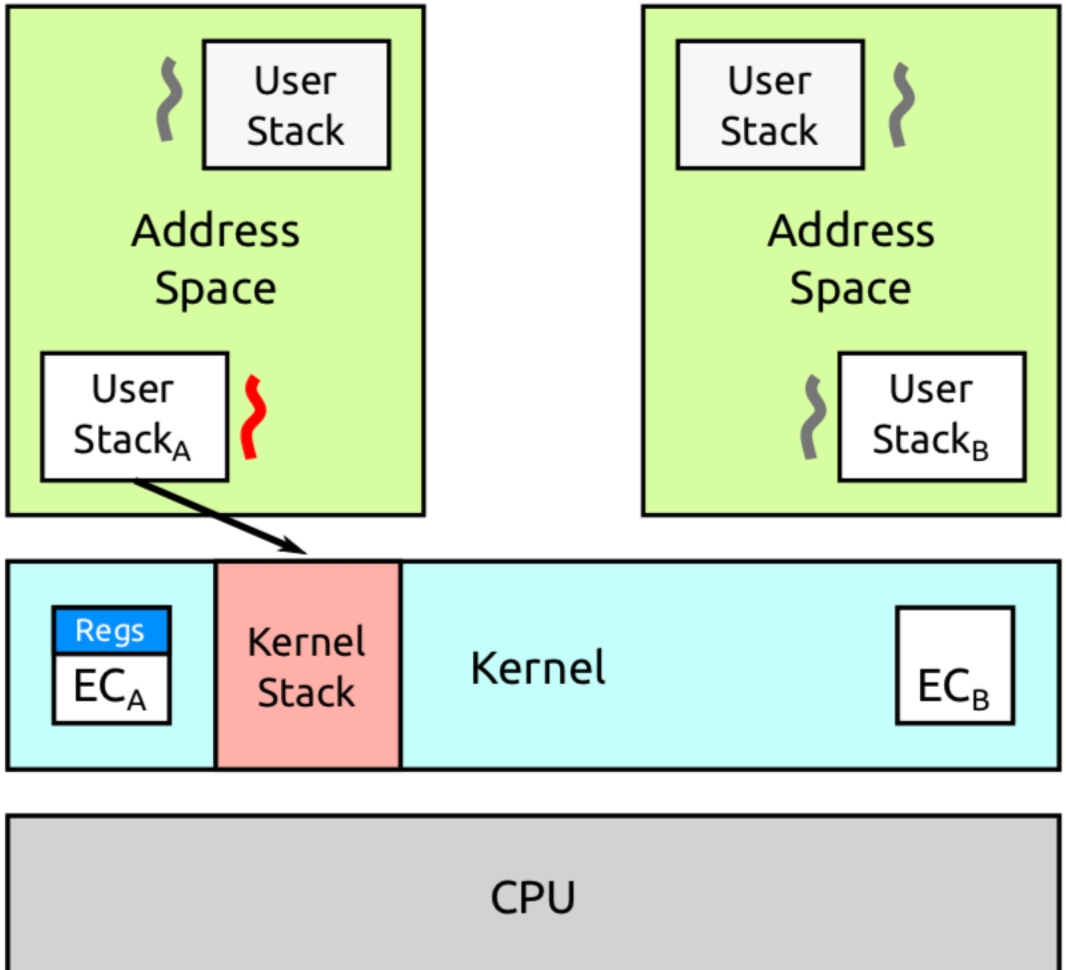
Thread Switch: Conventional



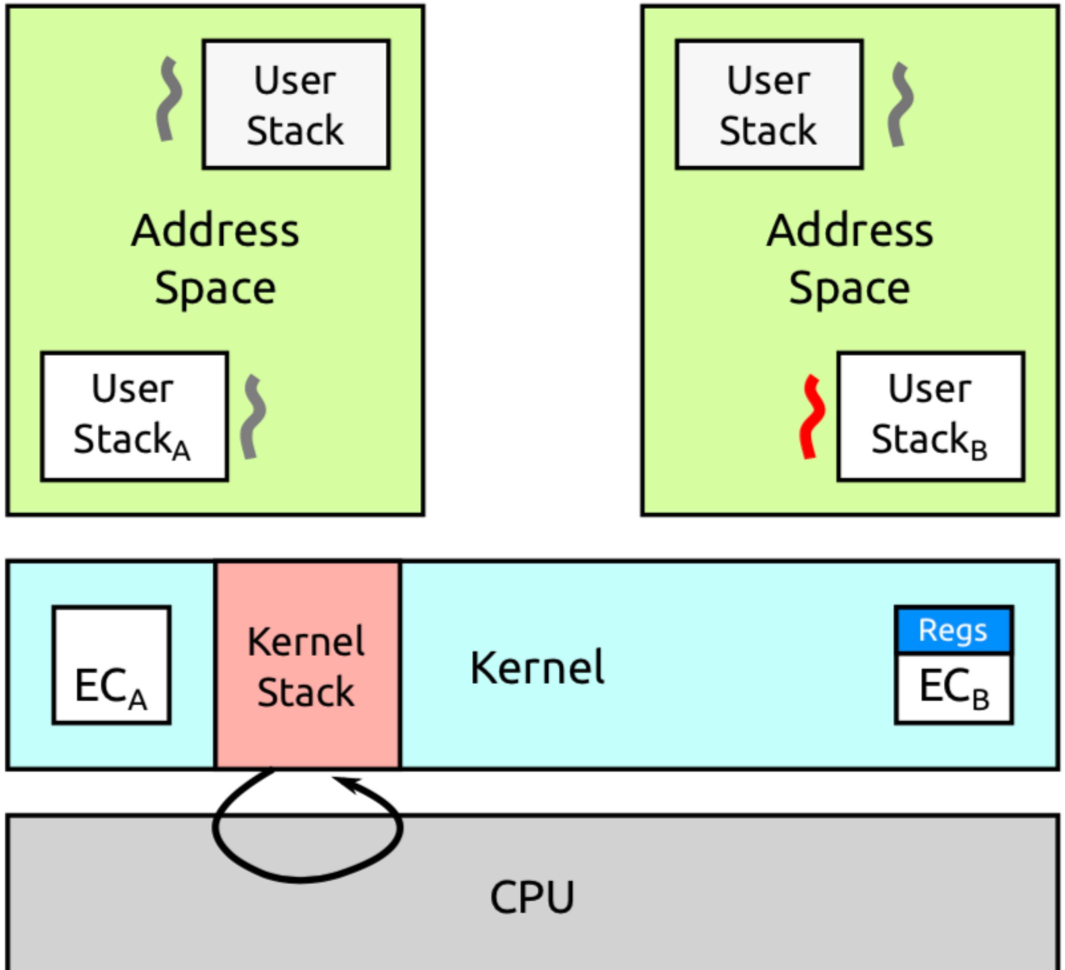
Thread Switch: Continuation Style



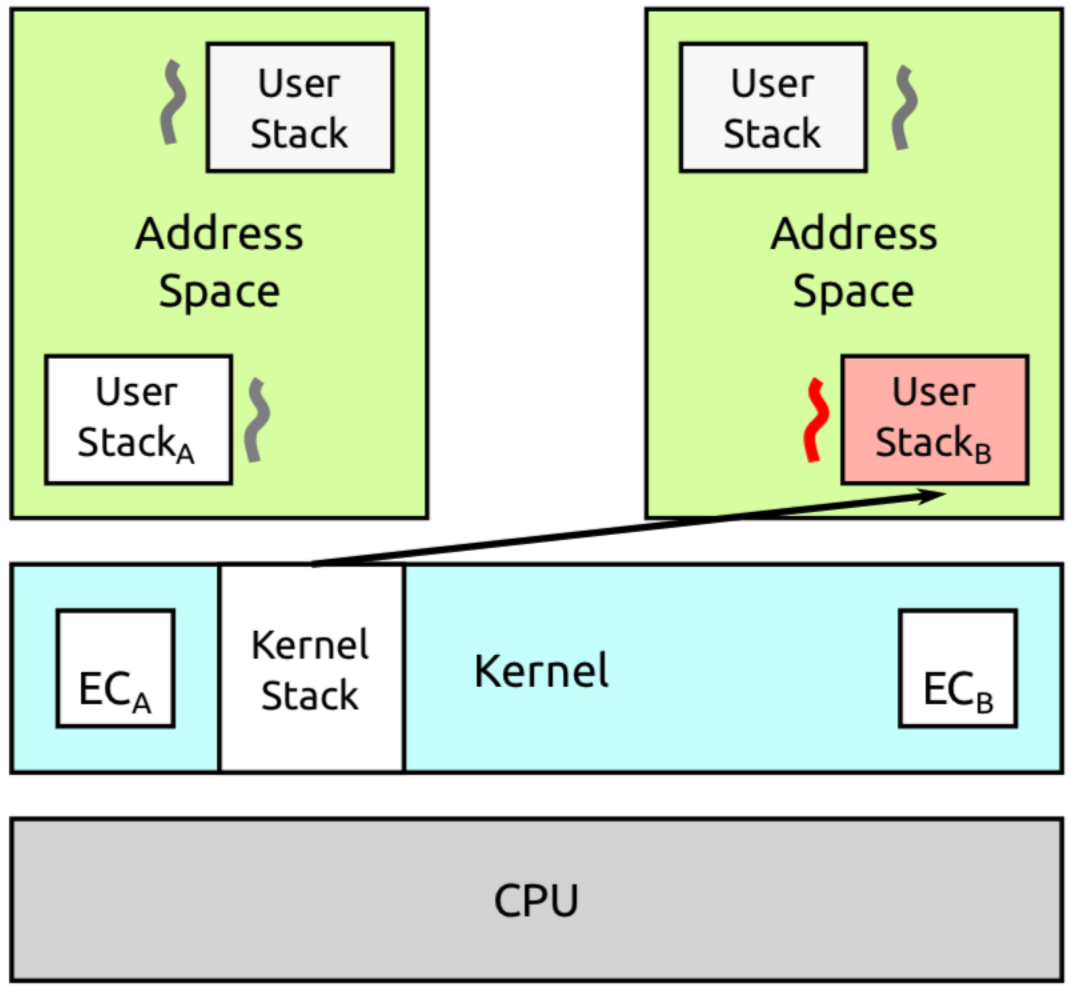
Thread Switch: Continuation Style



Thread Switch: Continuation Style



Thread Switch: Continuation Style



Improving IPC Performance

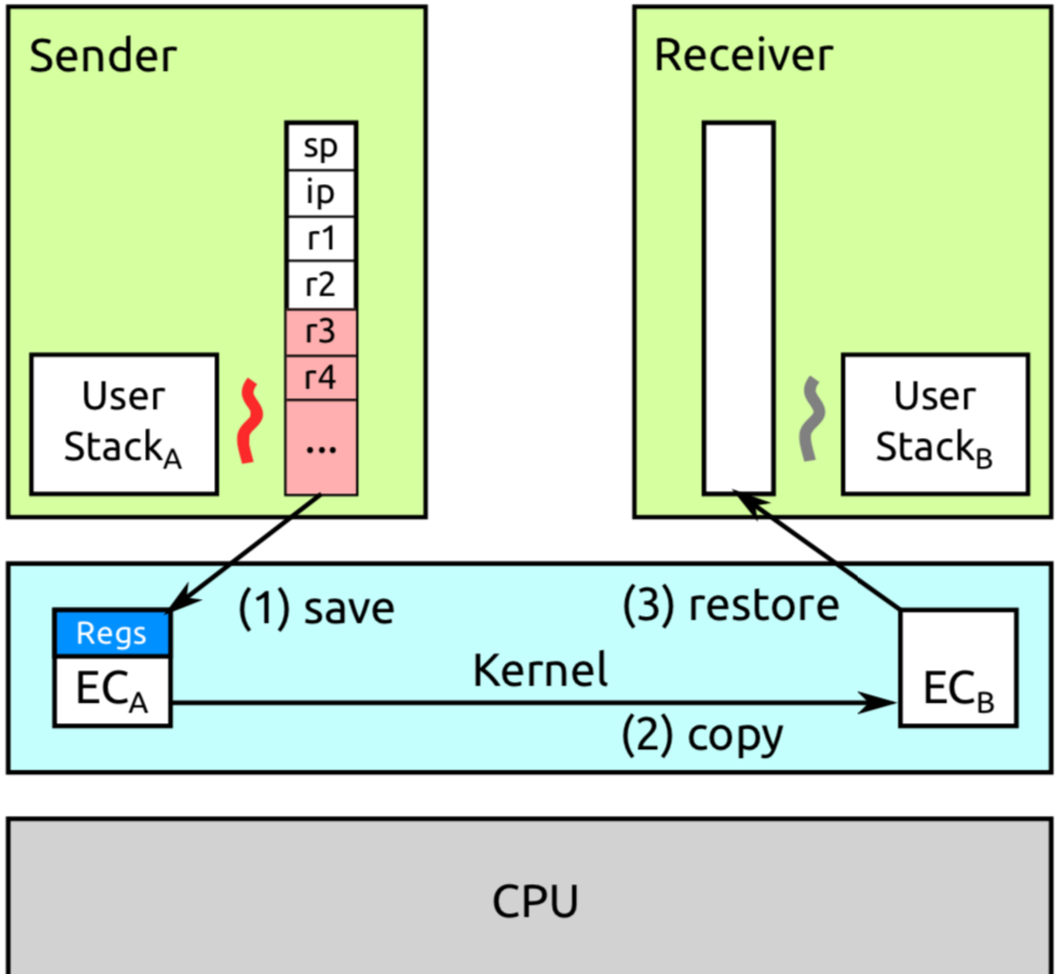
- **IPC is critical to the performance of the whole microkernel OS**

- **First generation microkernel systems**
  - Exhibited poor performance when compared to monolithic UNIX implementations
  - Particularly Mach, the best-known example
- **Typical results:**
  - Move OS services back into the kernel for performance
    - Mac OS X (Darwin): complete BSD kernel linked to Mach

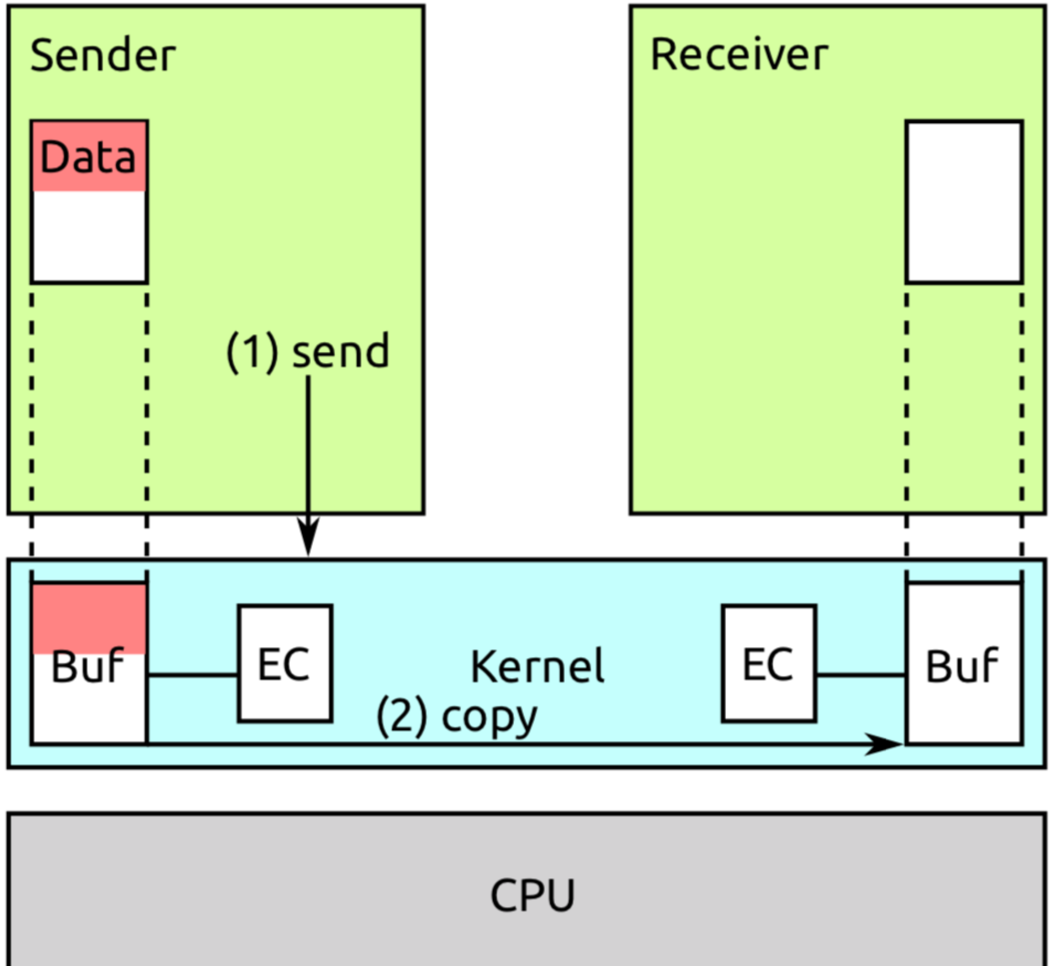
IPC Optimization Techniques

- **Minimize number of system calls**
  - Combined operations: Call, ReplyWait
  - Complex messages
    - Combines multiple messages into one operation
    - As many arguments as possible in registers
- **Copy messages only once**
- **Reducing cache and TLB misses**
- **Avoid jumps and checks on fast path**
- **Direct process switch to receiver**
- **…**

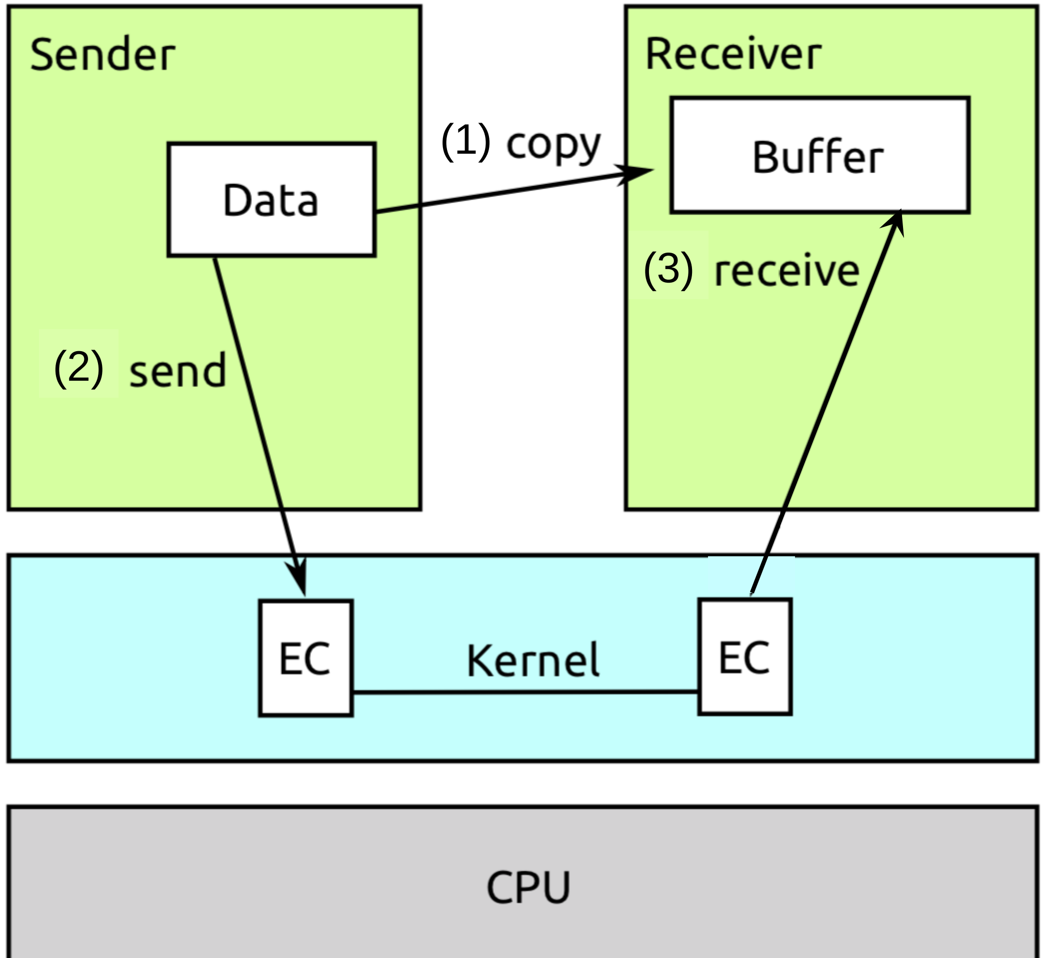
Register IPC



Kernel Memory IPC



Shared Memory IPC



Comparison

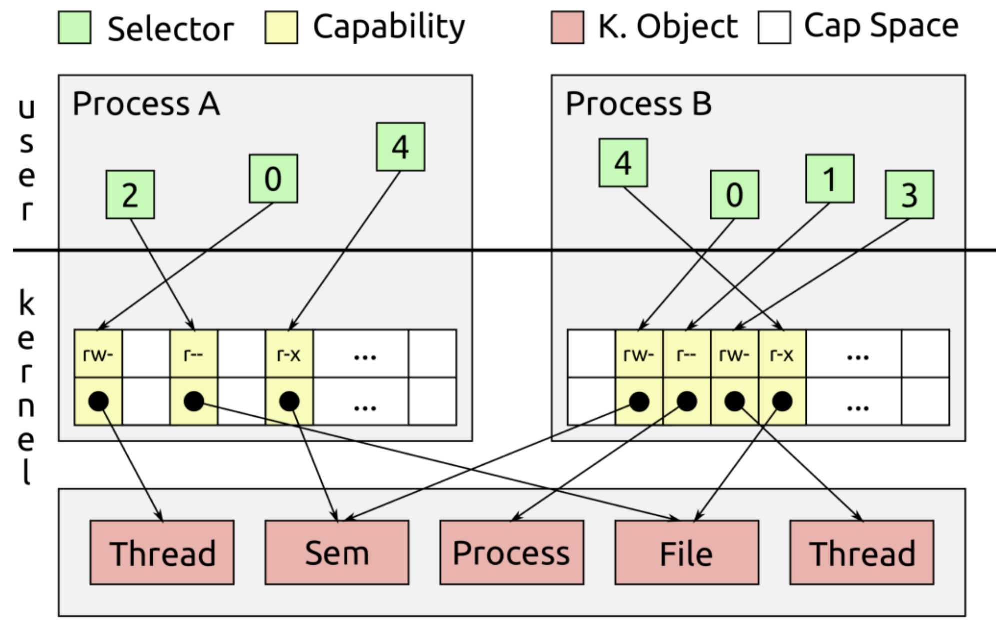
- **Register IPC**
  + Very fast
  - Amount of data limited to CPU registers
- **Kernel Memory IPC**
  + Larger amount of data (still limited)
  - More memory copies
- **Shared Memory IPC**
  + Amount of data not limited
  + Less memory copies
  - Security issues (e.g., TOUTOC, manipulation on the fly)

Sync and Async IPC

- **Synchronous**
  - Sender is blocked until receiver is ready
  - Data and control transfer directly from sender to receiver
- **Asynchronous**
  - Data is transferred to temporary location
  - Sender continues execution
  - If receiver arrives, the data is transferred to him
- **Comparison**
  - Sync is typically simpler and faster (no buffering)
  - Sync is less prone to DoS attacks (buffer memory)
  - Async is typically more functional
  - Async allows to do other work while waiting

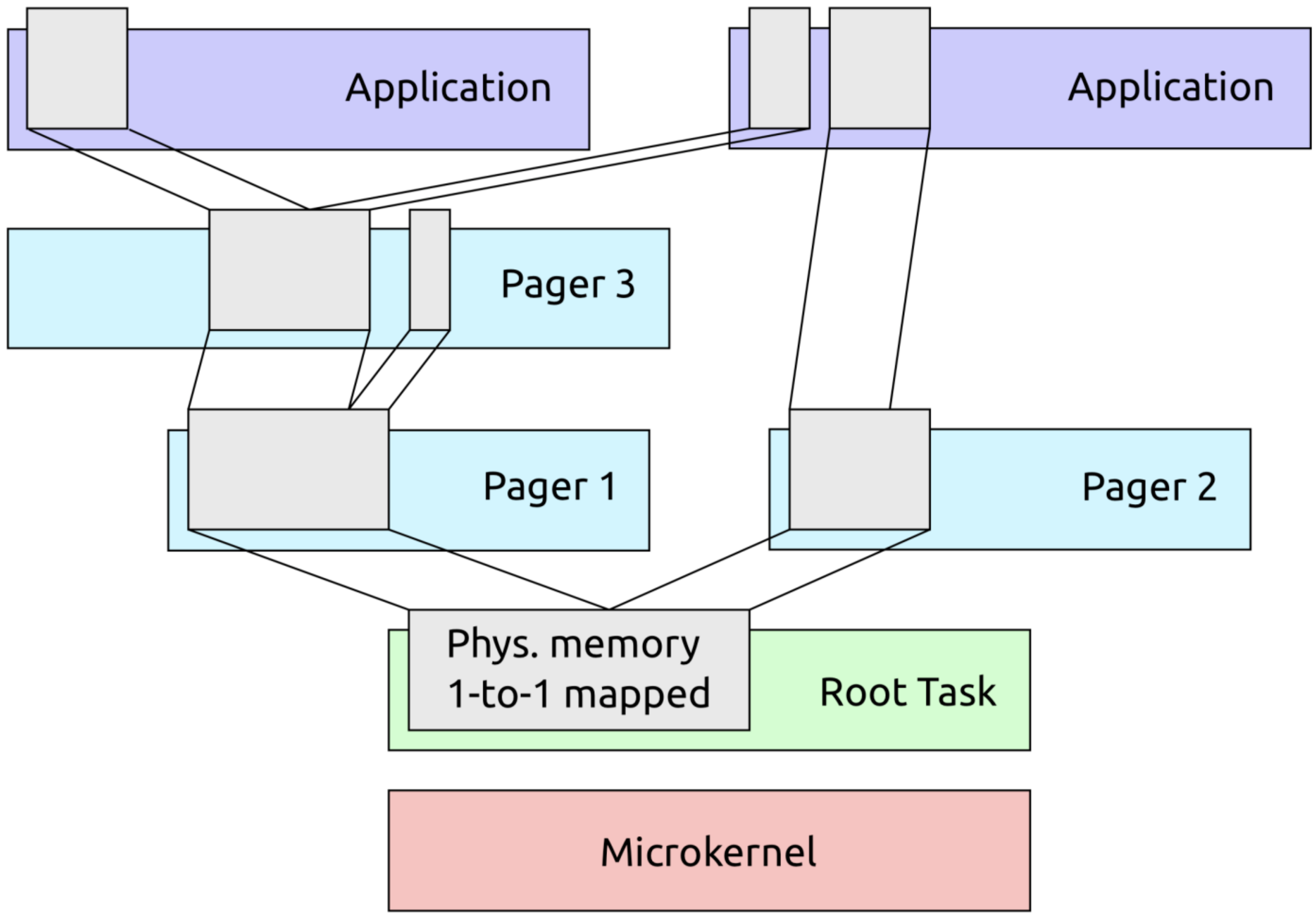
Usages of Sync and Async IPC

- **Synchronous IPC**
  - Exchange data
  - Exchange capabilities
    - **Copy/Move/Revoke**
- **Asynchronous IPC**
  - Signaling
  - Deliver interrupts to user space

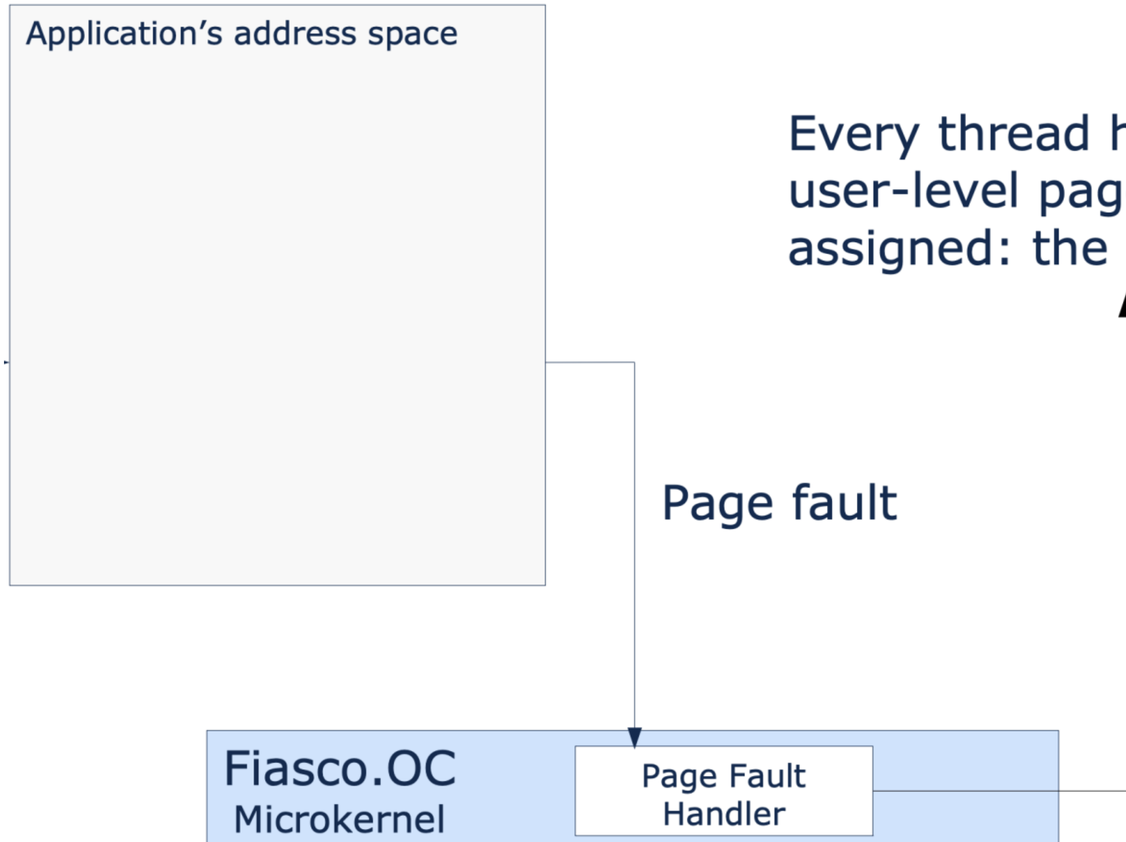


MEMORY MANAGEMENT

Hierarchical MM in L4

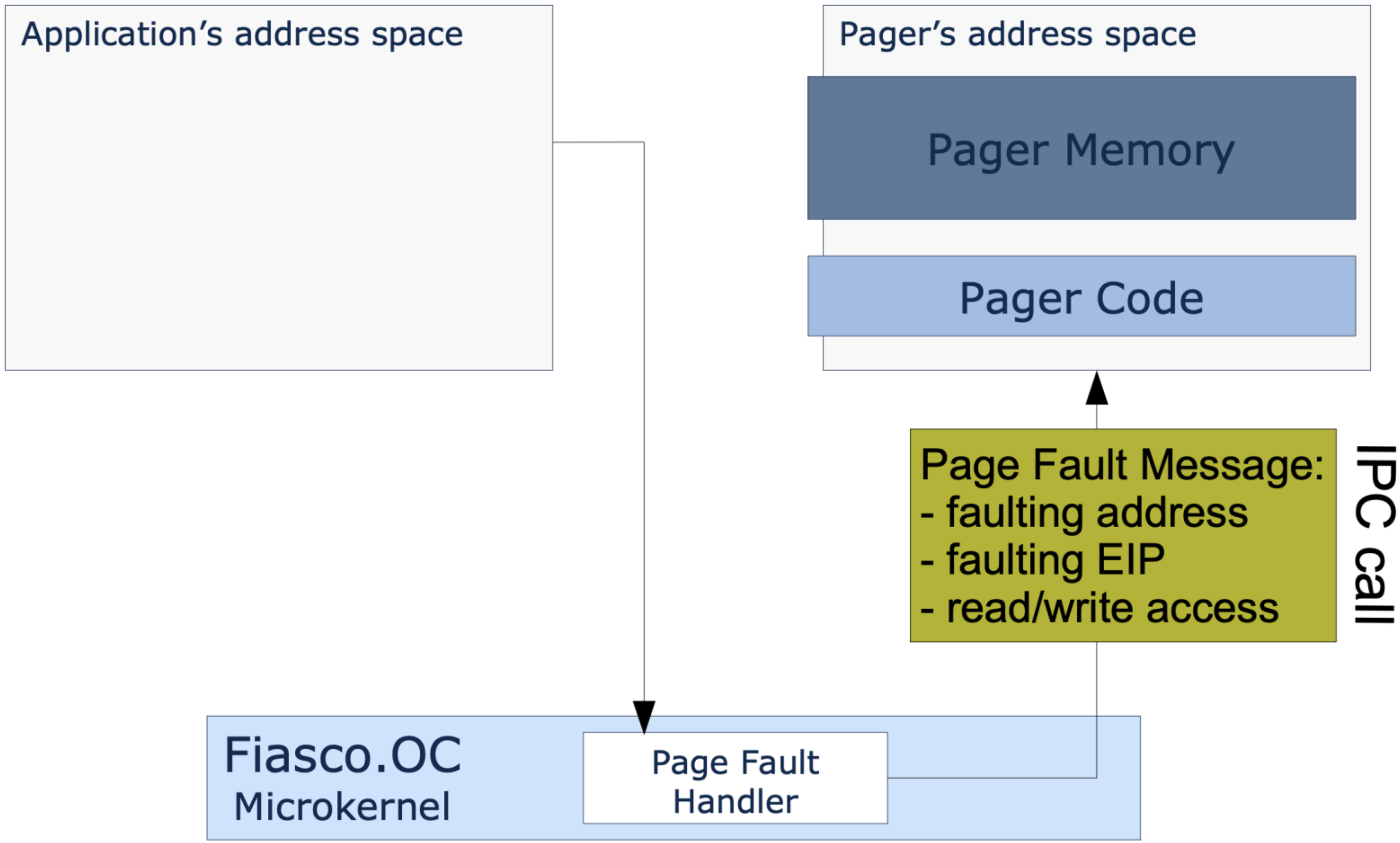


Page Fault Handling

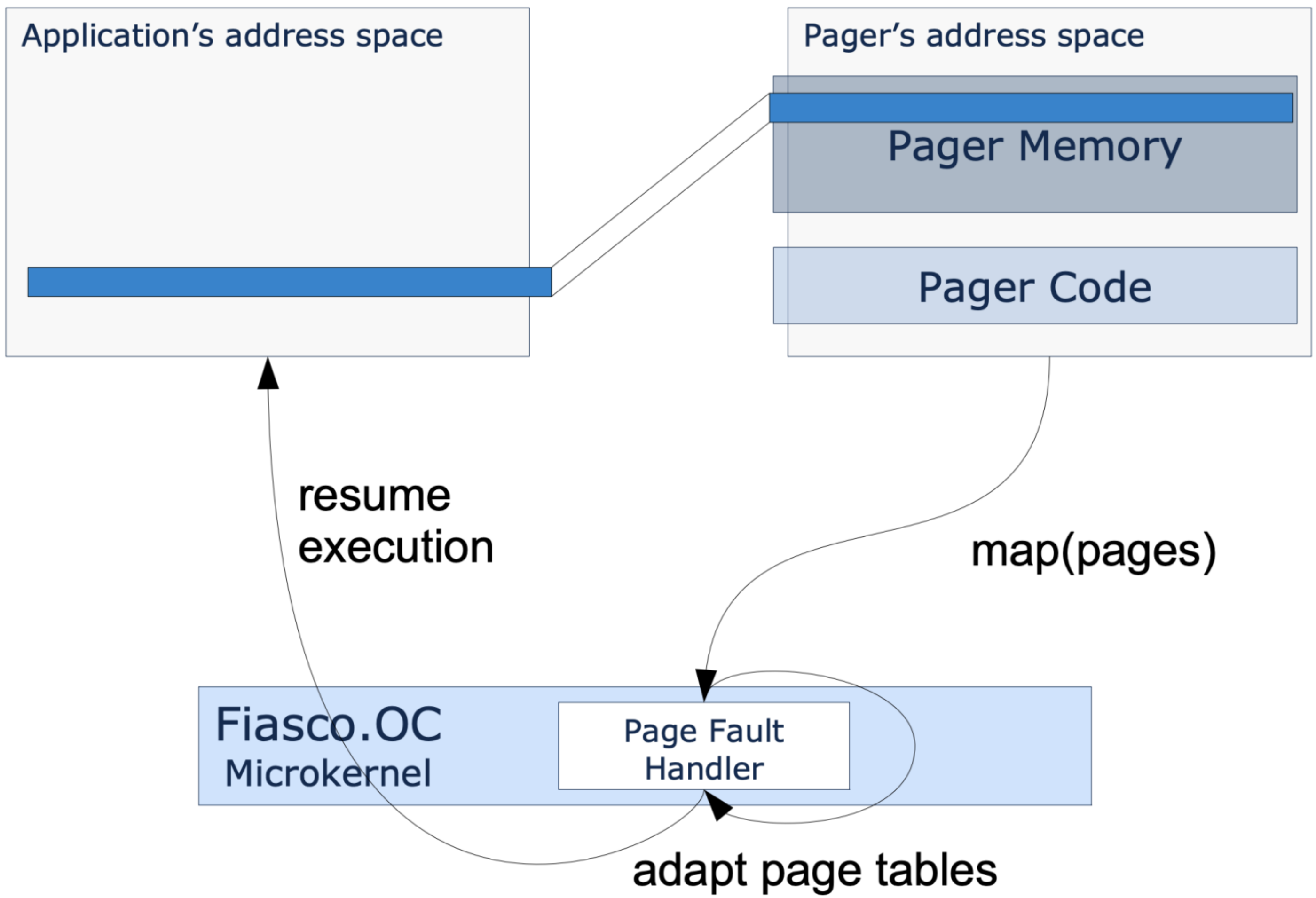


Every thread has a unique user-level page fault handler assigned: the **pager**.

Pager


Application's address space
Pager's address space
Pager Memory
Pager Code
Page Fault Message:
- faulting address
- faulting EIP
- read/write access
IPC call
Fiasco.OC
Microkernel
Page Fault
Handler

Memory Mapping

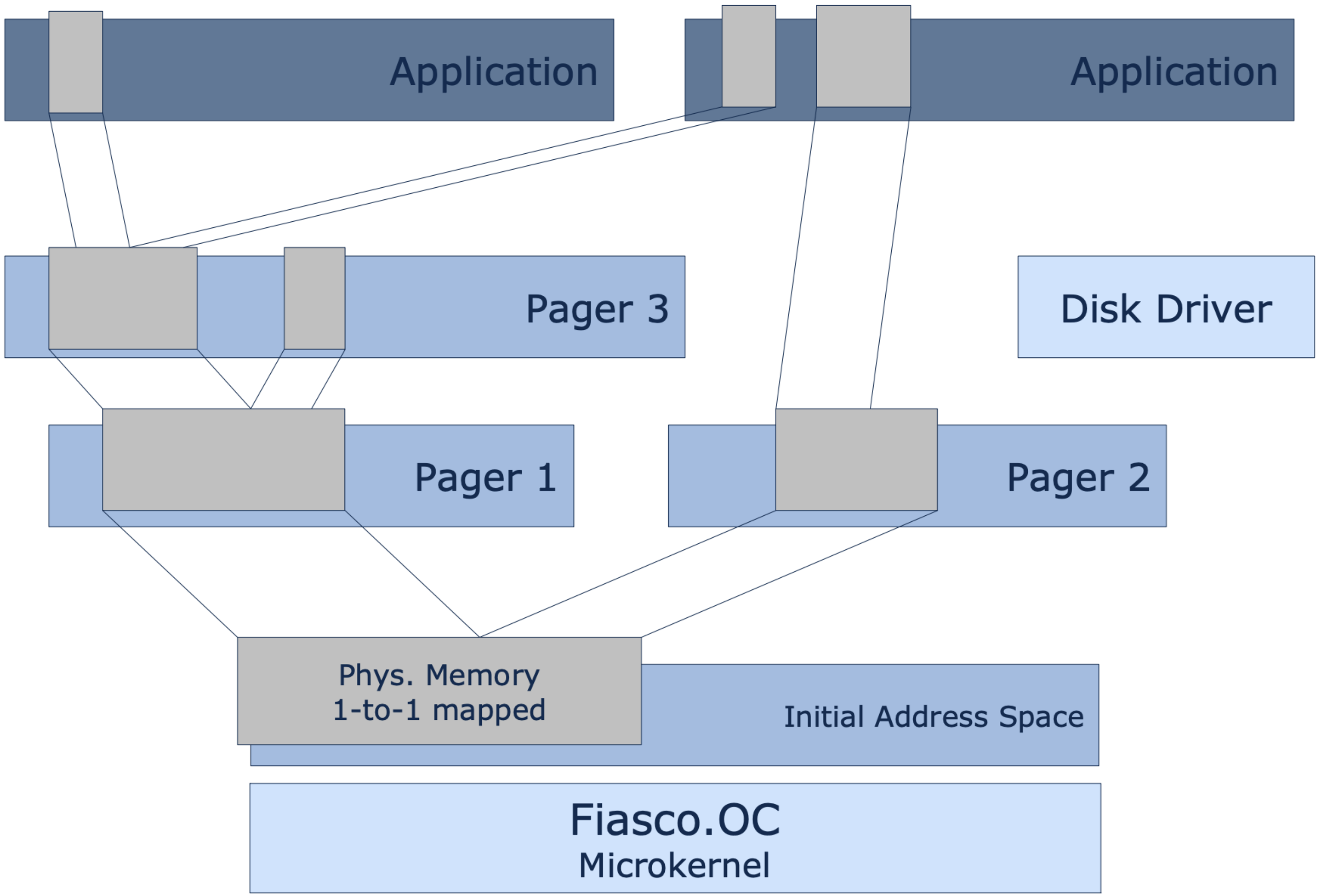


User-Level Pagers

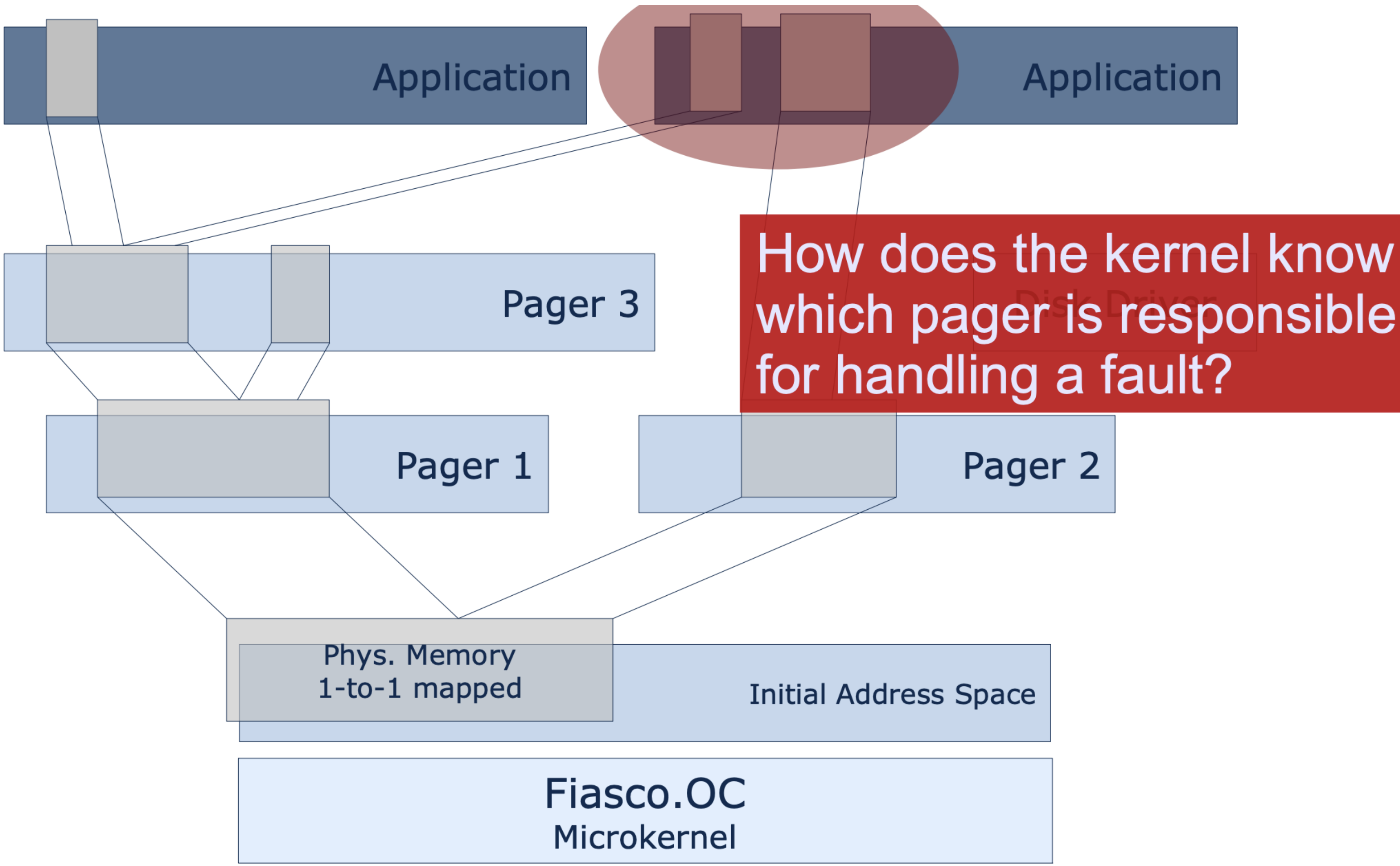
- **Chicken vs. egg: who resolves page faults for the Root Task?**
  - Fiasco.OC initially maps all memory to it, so that it never raises page faults

- **Complex vs. simple management policies**
  - Solution: Pagers can be stacked into pager hierarchies

Pager Hierarchies

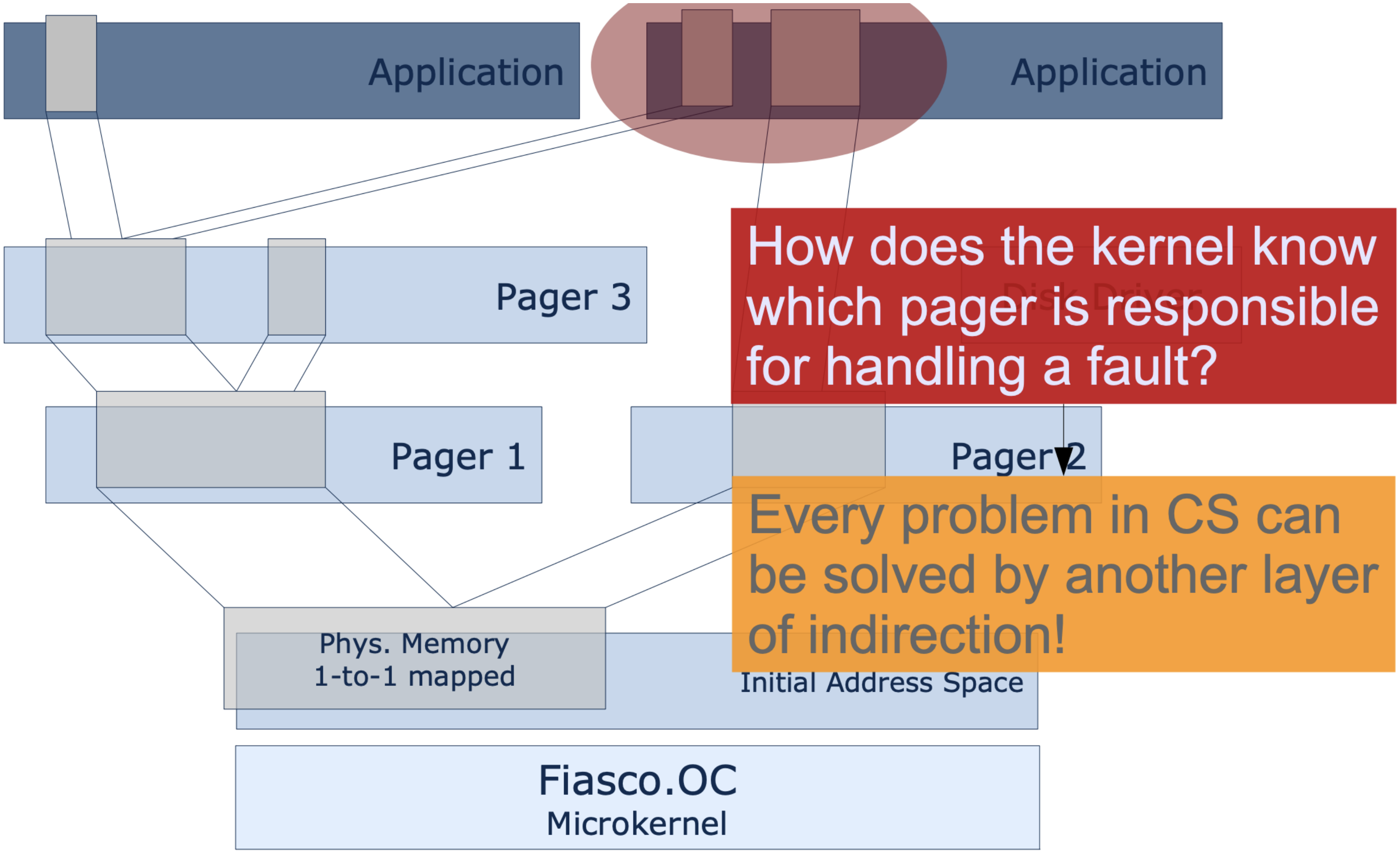


Pager Hierarchies



How does the kernel know which pager is responsible for handling a fault?

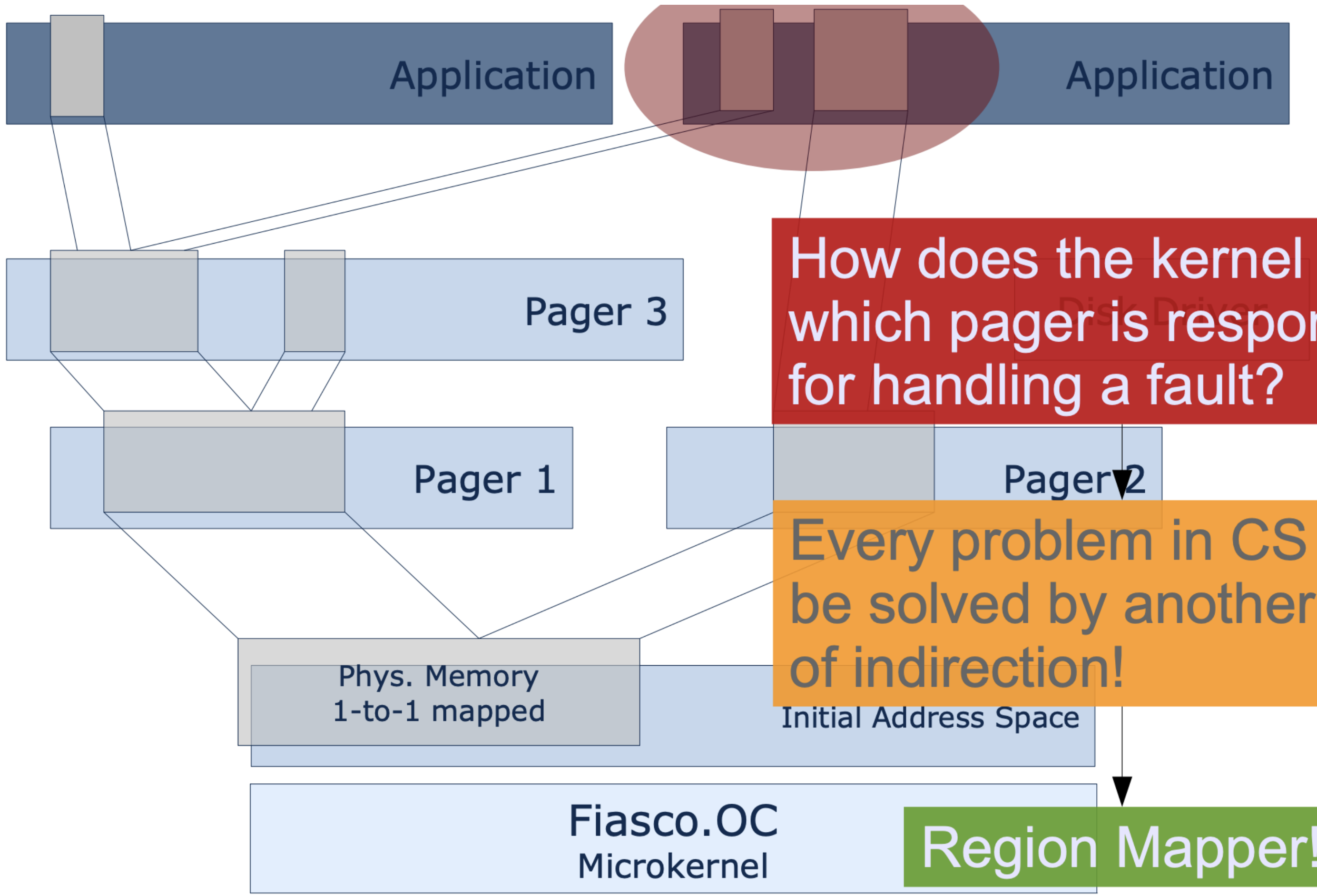
Pager Hierarchies



How does the kernel know which pager is responsible for handling a fault?

Every problem in CS can be solved by another layer of indirection!

Pager Hierarchies



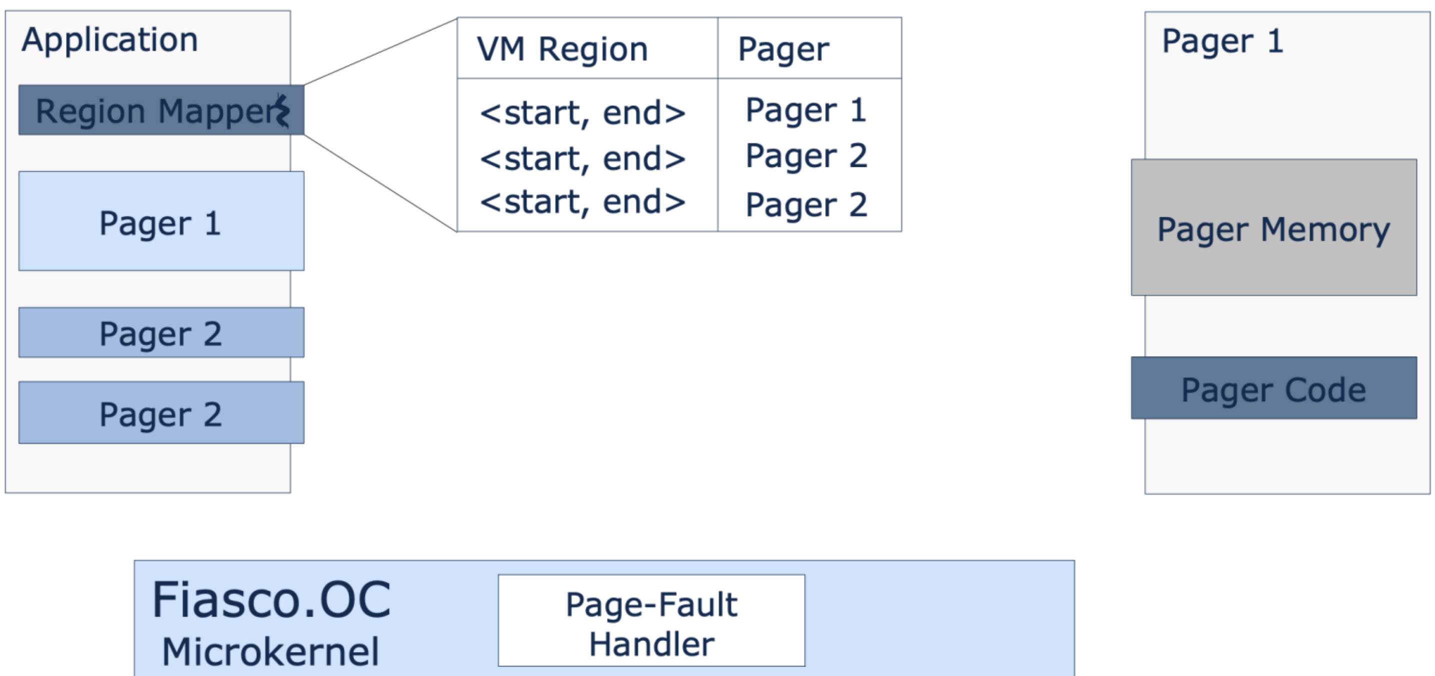
How does the kernel know which pager is responsible for handling a fault?

Every problem in CS can be solved by another layer of indirection!

Region Mapper!

Region Mapper

- Every task has a single pager thread, the **region mapper (RM)**
  - RM is the first thread within a task in Fiasco.OC
  - RM maintains a region map



Region Mapper and Page Faults

<u>Step-1:</u> All threads of a task have the RM assigned as their pager

<u>Step-2:</u> Fiasco.OC redirects all page faults to the RM

<u>Step-3:</u> RM then uses synchronous IPC calls to obtain real memory mappings from the external pager responsible for a region

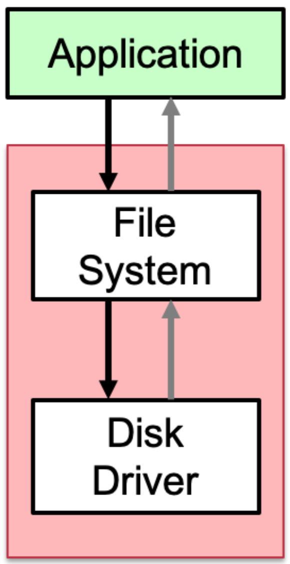
Disadvantages of Pager

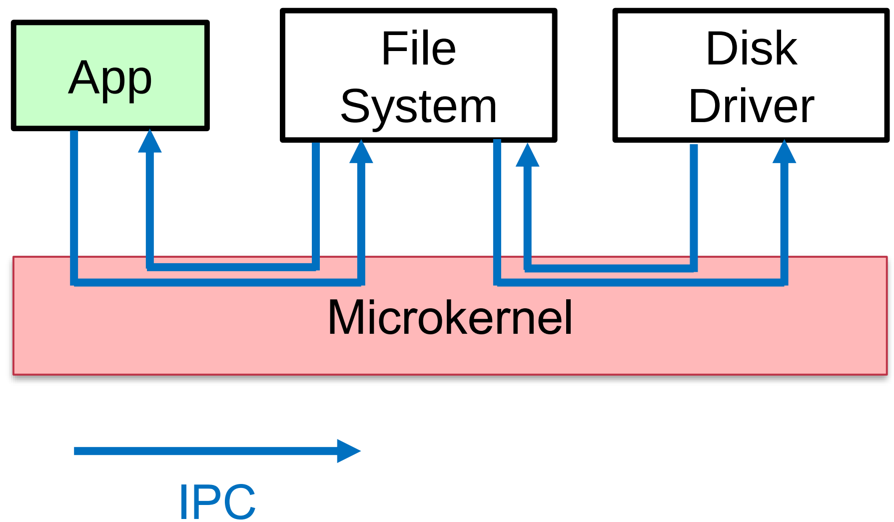
- **Poor performance**
- **Complex developing efforts**
  - Is that really necessary?

- **Hybrid solution**
  - E.g., Google Zircon

SOME RESEARCHES

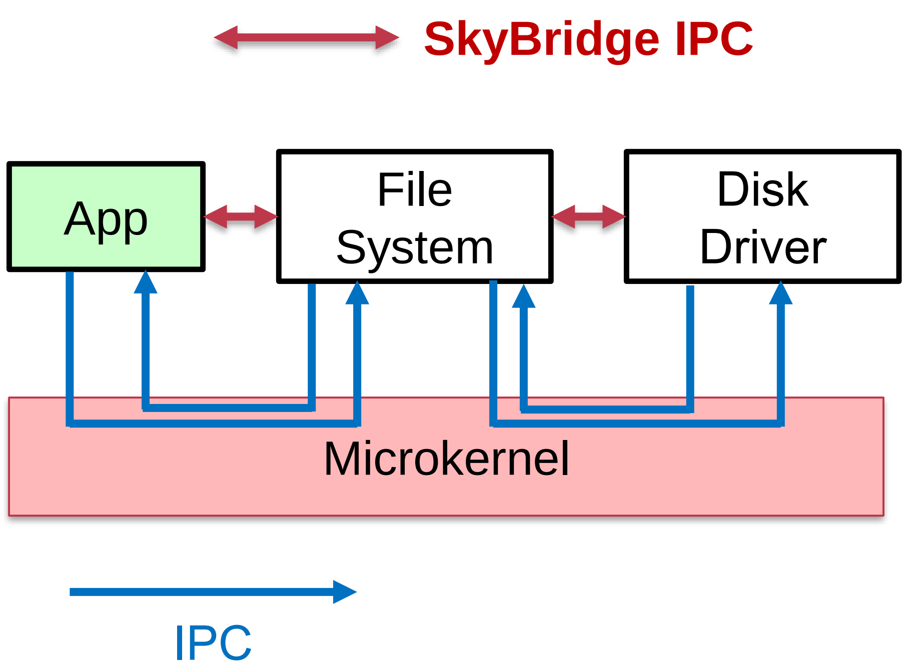
Tradeoff: Performance and Isolation





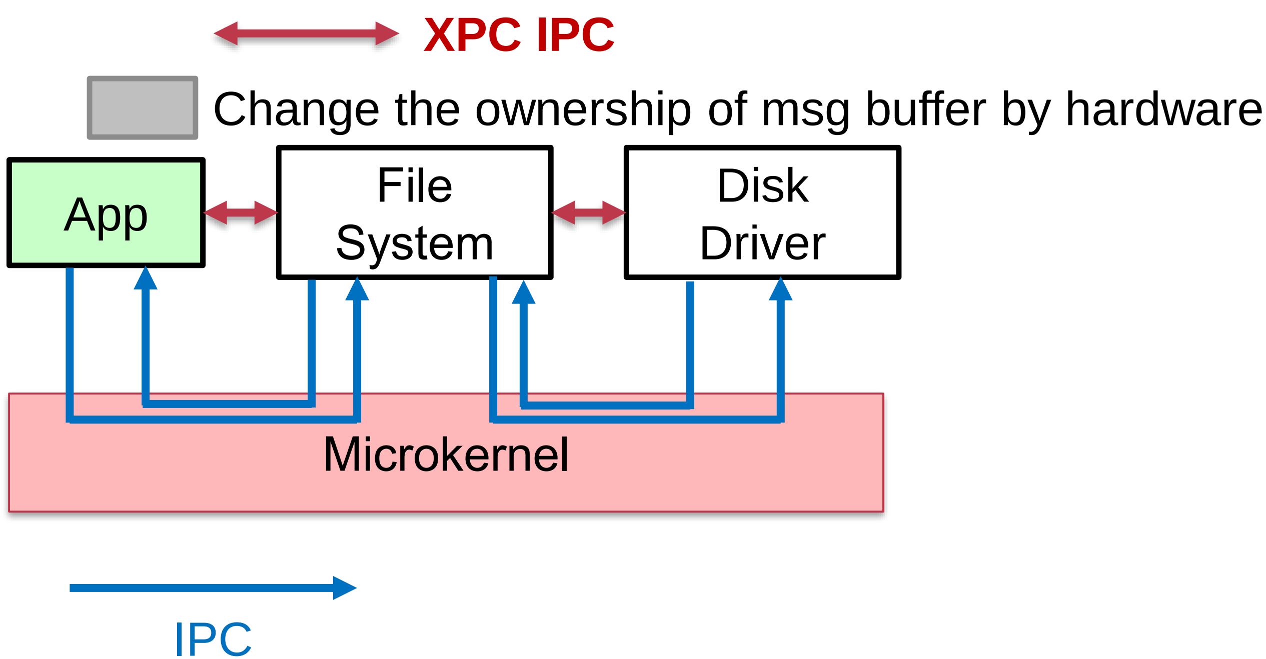
SkyBridge: Make IPC Faster

- **EuroSys19: No kernel involvements during IPCs**



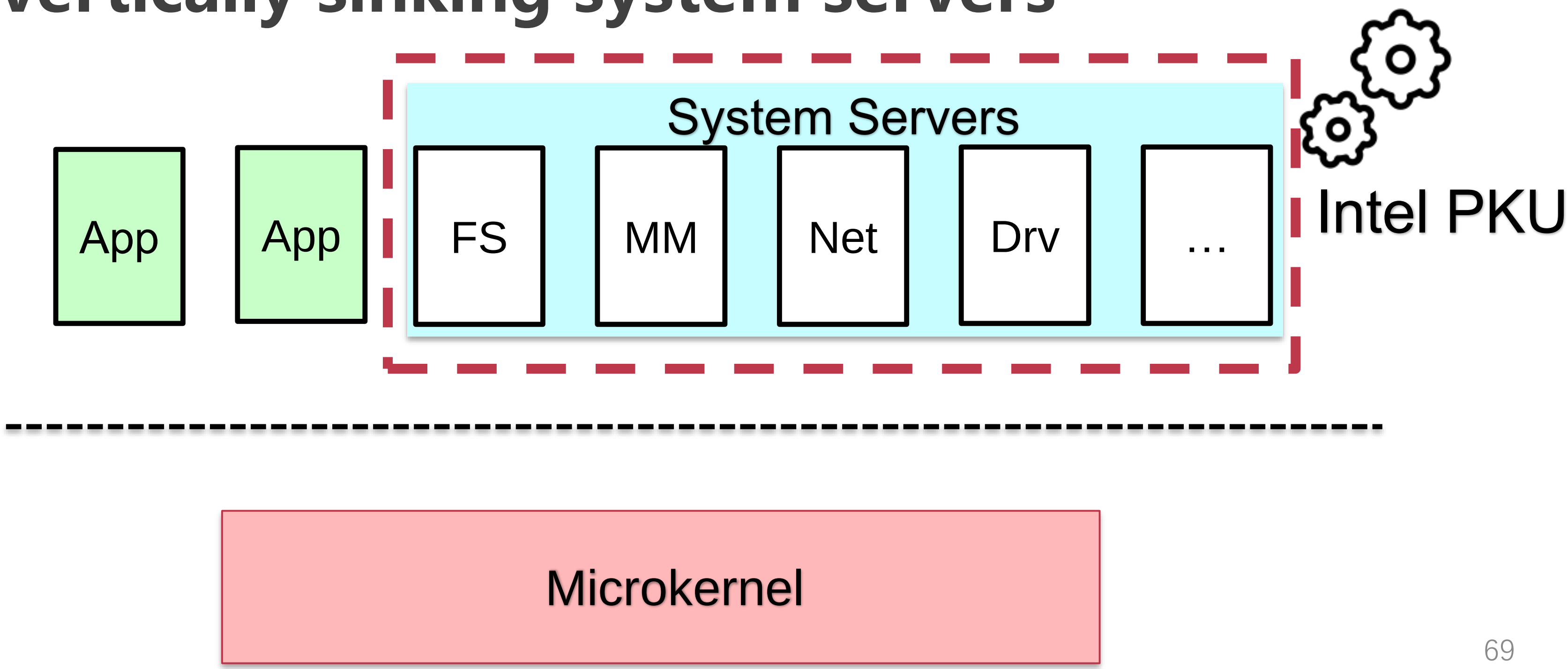
XPC: Make IPC Faster

- **ISCA19: No kernel involvements during IPCs; Zero-copies during IPCs**



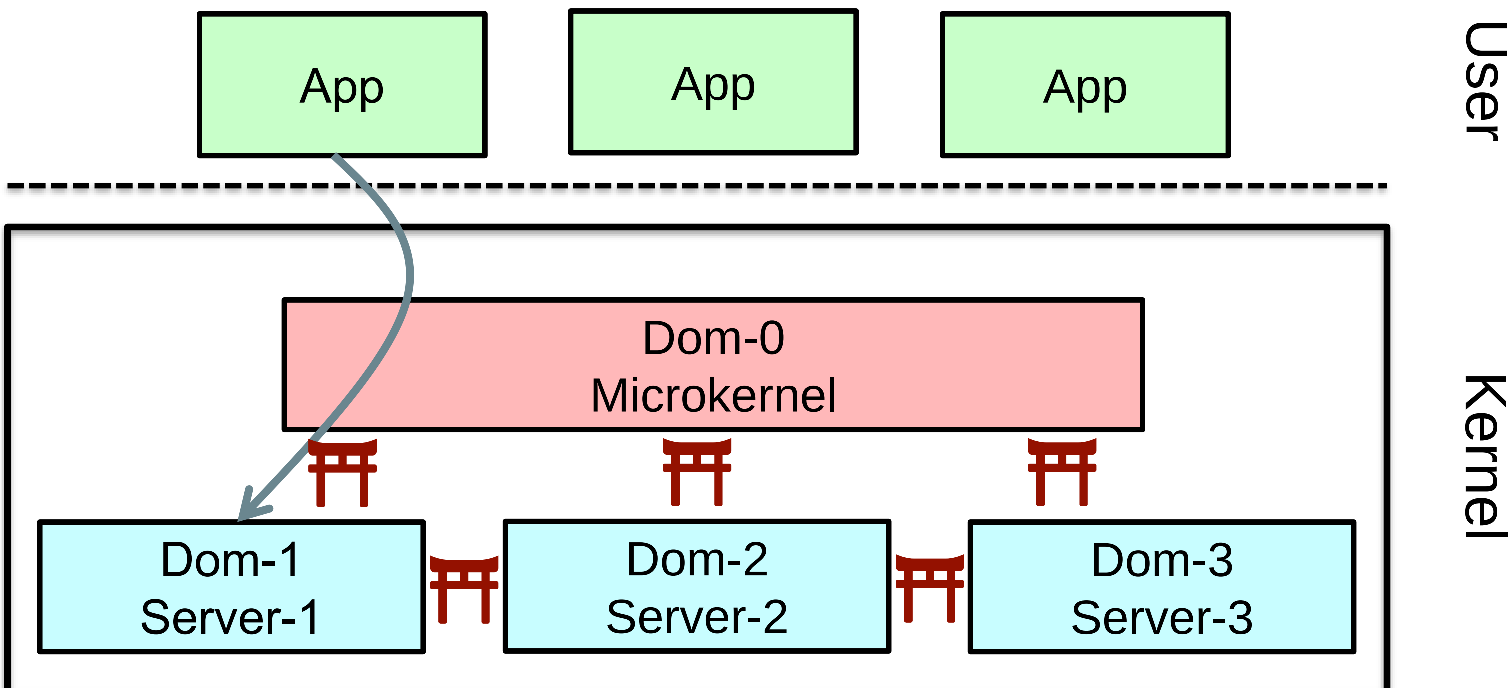
UnderBridge: Make IPC Faster

- **ATC20: No kernel involvements during IPCs by vertically sinking system servers**



UnderBridge: Make IPC Faster

- Build **execution domains** in the **kernel space**



CuriOS: Improving Reliability through Operating System Structure

Francis M. David & Ellick M. Chan &

Jeffrey C. Carlyle & Roy H. Campbell

UIUC

OSDI 08

Is Microkernel Reliable enough?

- **Benefits: Better Isolation? Fault Recovery?**
- **Problem**
  - Blindly restarting a service results in the state loss and affects all clients that were using the service
- **Goal**
  - Use lightweight distribution, isolation and persistence of OS service state to mitigate the problem of state loss during a restart

Fault Tolerance in Microkernel

- **Microkernel designs componentize the OS into servers managed by a minimal kernel**
  - Inter-component error propagation is significantly reduced
  - Better fault isolation
- **Recovery from a microkernel server failure is typically attempted by restarting it**
  - Minix3: a printer driver √
  - Minix3: a file system ×

Simply Restarting is not Enough

- **Obstacle: many OS services maintain states related to clients**

- **Take FS as an example**
  - Reads and writes to existing open files cannot be completed because the restarted server cannot recognize the file handles (i.e., client state lost)

How to Solve the Problem

- **Design-Choice-1:**
  - Clients take care of service restarts and state loss
  - Increased code complexity ×

- **Design-Choice-2:**
  - Provide some form of persistence to the client-related state information √

Simply Persisting is not Enough

- **Intra-component error propagation**
  - An error that occurs in an OS server can potentially corrupt any part of its state before being detected
  - A significant limitation of traditional microkernel systems

Server State Management

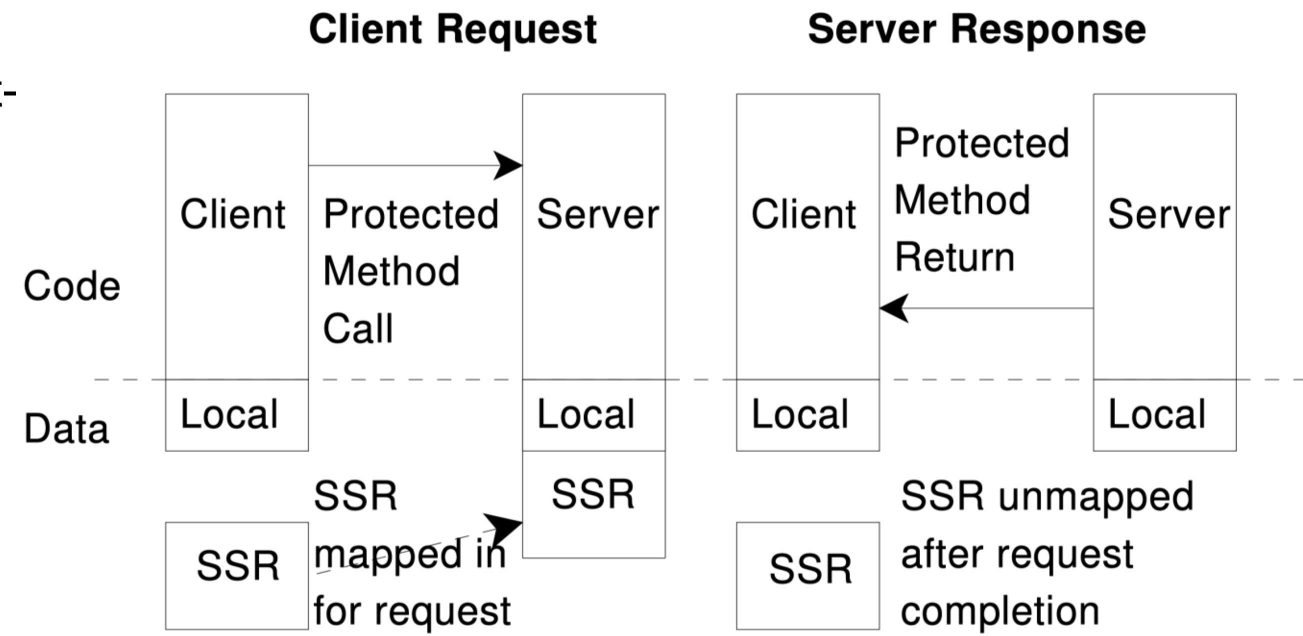
SSR: Server State Region

Stores an OS server's client-related information

Created during connection

Protected from clients and server

Passed as an argument



Server State Management

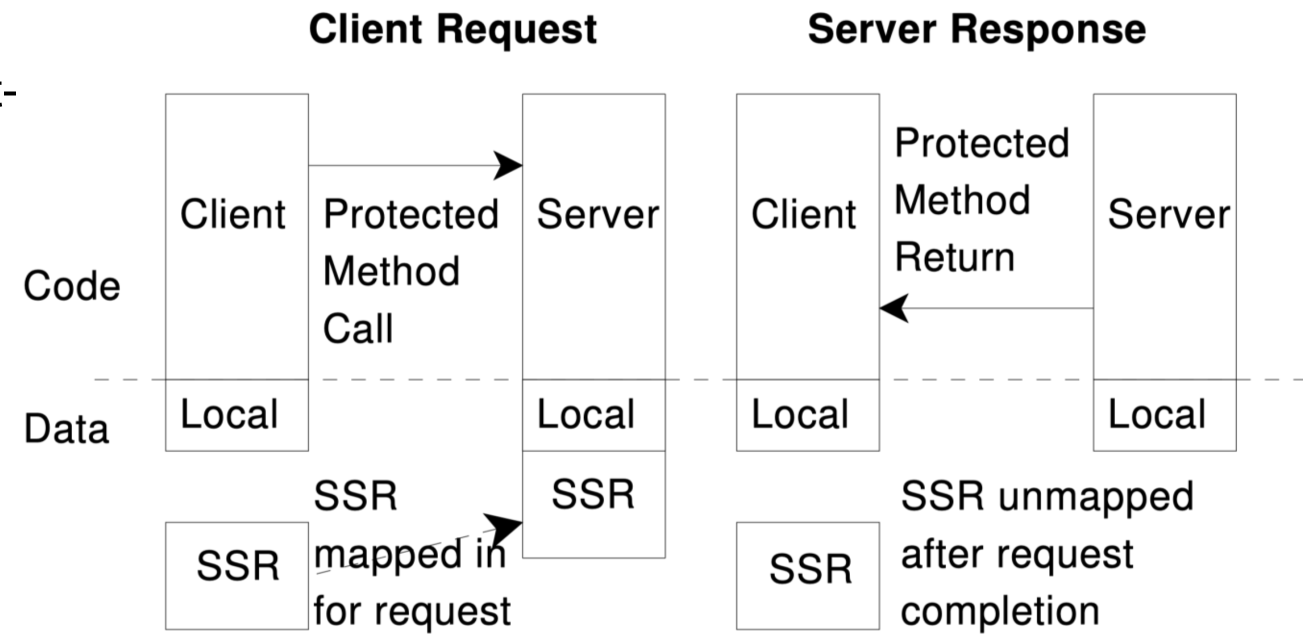
SSR: Server State Region

Stores an OS server's client-related information

Created during connection

Protected from clients and server

Passed as an argument



Persistence and isolation of client-related state

SSR Manager

- **Register a new server**
- **Bind a client to a server**
  - SSR creation
- **Undo a client-server binding**
  - SSR deletion
- **Enumerate all the SSRs associated with a server**

OS Service Construction

- **Three types of servers:**
  - Stateless (e.g., some drivers)
  - No require collective information about all of its clients
    - E.g., a server that provides pseudo random numbers based on a per-client seed
  - Require knowledge about all of its clients in order to service a request
    - E.g., a scheduler
    - redundantly cache this information locally to process

Server Recovery

- **Restarting only is enough for the first two types**

- **Type-3 servers need to register a recovery routine**
  - Query the SSR Manager to obtain all associated SSRs
  - Re-create the internal state of the restarted server
  - Heuristics consistency check is required since some SSR may be corrupted (and most clients can survive)

CuriOS

- **Make efforts on recovering stateful OS servers**
- **Fault tolerance**
  - Isolation is achieved by the microkernel design and is better since the isolation among SSRs
  - Recovery is achieved by persistence and isolation of client-related state
- **<u>There are still many open questions</u>**

Conclusion

- **Small code size is necessary but not enough**
- **Capability and IPC are two core abstractions**
- **Open questions**
  - Performance
  - Security
  - Compatibility
  - Reliablity

Thanks!