Advanced Operating Systems
Summer Semester 2024/2025

Martin Děcký

4

Communication

Kernel Interfaces

- **Most kernels have internal structure**
  - **Monolithic kernels:** *So large that a structure is required*
  - **Microkernels:** *Not so small that a structure is not helpful*
  - Subsystems, modules, classes, (hardware) abstraction layers, etc.
  - Usual software engineering best practices
    - Code is written once, but read many times
    - Similar things should be done in similar ways
    - Keep it simple / You aren’t gonna need it
    - Don’t repeat yourself
    - Clear definition of purpose, difficult to misuse, kind to errors

Kernel Interfaces

- **Hardware abstraction layer**
  - Interface between platform-specific and platform-independent code
    - Primitive data types (machine word), atomics, function pointers
    - Thread context (non-volatile / preserved / callee-saved registers), interrupt context (complete machine state)
    - Address space layouts, ASIDs
    - Memory mapping structures
    - Interrupt vectoring, exception levels, inter-processor interrupts
    - Stack layout (sizes, frame pointer, bias, red zone, tracing)
    - Actual platform-specific code (initial bootstrap, kernel entries and exits, atomic operations, memory barriers, cache management, assembly code, platform drivers)
    - Platform-unification code (e.g. segmentation setup on x86, register stack engine on IA-64 & SPARC)

Kernel Interfaces

- **Typical subsystems**
  - Execution management
    - CPUs, execution contexts (threads), scheduling contexts, exceptions, interrupts
  - Memory management
    - Address spaces (tasks), address space areas (paging, TLB, ASIDs)
  - Time management
    - Alarms, timeouts, delays
  - Synchronization
    - Preemption control, mechanisms, primitives
  - Syscalls
    - Safety / security boundary between kernel space / user space
  - Device drivers
  - Utilities
    - Run-time configuration, loaders, observability, debugging, logging

Kernel Interfaces

- **Additional microkernel subsystems**
  - Capabilities
    - Factories
  - User space delegation
    - Platform control, exceptions, user space device drivers
- **Additional monolithic kernel subsystems**
  - File systems
  - Network stacks
  - Power management
  - Cryptography

System Calls

- **Kernel entry point from user space**
  - Usually via a dedicated “SYSCALL” instruction
    - But other tricks exist (synchronous interrupt, exception, etc.)
    - Might encode the syscall number in the instruction
  - Similar to a method call of a virtual method table
    - The “object” is logically either the entire kernel or a capability
    - The “method table” is either a syscall table, a switch or a cascade (of either or both)
  - Basic arguments universally passed in GPRs
    - Least trouble with validation
    - Might not align perfectly with ABI
  - Extended arguments usually passed as pointers to user memory
    - Need thorough validation (time-of-check to time-of-use races)

System Calls Multiplexing

- **None**
  - Each syscall is a fixed method (more-or-less)
- **Capabilities**
  - Each capability type provide a set of methods (usually fixed)
- **`ioctls`**
  - Each object instance (e.g. file descriptor, netlink socket) provides an arbitrary set of methods or messages

Kernel Object Naming

- **Capabilities**
  - Also file descriptors, sockets, handles, virtual addresses, etc.
  - Local identifiers of objects
  - Implicitly follow the “share nothing” principle
    - No extra effort for partitioning required
- **Global resources**
  - Tasks (processes), threads, users, groups, file names, keys, network devices, network addresses, physical addresses, etc.
  - Explicit partitioning required
    - Namespaces, containers, zones, etc.
      - Class of global resources that group and isolate global resources
      - Non-trivial to achieve a truly “share nothing” state

“Everything Is a File”

- **Original UNIX paradigm**
  - N.B.: Mixes two aspects (naming, handling) together
  - Resources uniformly identified as file names
    - Special files for global “non-files” (e.g. named pipes, device nodes)
    - Internal file systems for local “non-files” (e.g. anonymous pipes, sockets)
    - Special (synthetic) file systems for exposing run-time data (e.g. /proc, /sys)
    - Despite the effort, there were always exceptions (processes, threads, semaphores, etc.)
  - Resources handled uniformly
    - Basic operations (create, destroy, etc.) and input/output stream of bytes
    - Despite the effort, there were always major exceptions
      - Special operations for different types of objects
      - ioctls as a completely unconstrained API

“Everything Is a File”

Source: DALL·E 3 via ChatGPT 4

Everything Is …

- **… a file (for real)**
  - Plan 9
    - No ioctls, just a fixed set of operations (9P protocol)
      - Version, Attach, Auth, Walk, Open, New, Clunk, Delete, Stat, Read, Write, Flush
      - Everything marshalled as streams of bytes
- **… an object**
  - Windows
    - Pragmatic approach without sticking to a paradigm with exceptions
      - “Normal APIs” instead of magic ioctls or magic strings
      - Often some degree of uniformity might be a benefit (e.g. for enumeration)
- **… a capability**
  - Actual local uniform naming (but not uniform handling)
    - Some uniform handing thanks to the generic capability operations
- **… a memory area**
  - All resources represented as (demand mapped) virtual memory
    - Everything marshalled as byte accesses

Device Drivers Interface

- **Device drivers are portable (to a degree)**
  - Platform specifics can be abstracted
    - UART driver accesses hardware registers (I/O ports or MMIO)
    - PCI device driver accesses PCI configuration space
    - USB device driver uses USB controller endpoints
  - Host / device endianess, memory models, etc.
  - Class drivers
    - Supporting many individual devices via a vendor-neutral interface
      - USB HID, Mass Storage, UVC, etc.
  - Tree of device driver instances
  - Follows the hierarchy of devices
    - Example: Root driver, platform driver, interrupt controller driver, DMA controller driver, PCI driver, PCI bridge driver, USB controller driver, USB class driver, custom USB endpoint driver
  - Managing and delegating resources

Device Drivers Framework

- **Implementing common parts of device drivers**
  - Driver instance life cycle
    - Discovery (bus enumeration, hot plug/unplug), probing, attaching, detaching
  - Resource delegation
    - I/O port ranges, MMIO ranges, interrupts, DMA areas, power quotas, etc.
    - IOMMU programming
  - Device soft state management
    - Software mirror of hardware state
    - Device initialization, device / bus reset, device surprise hot removal
  - Device naming
    - Enumeration
    - Persistent instance identification
  - Level-triggered interrupts vs. user space drivers

Classical IPC

- **POSIX signals**
  - Since UNIX Version 4
  - Asynchronous notification sent to a process (thread)
    - Similar to level-triggered interrupts (including masking)
    - Sender uses the `kill(2)` syscall
      - Run-time exceptions and state changes also cause signals (`SIGFPE`, `SIGSEGV`; `SIGPIPE`, `SIGINT`, `SIGSTOP`/`SIGTSTP`, `SIGCONT`, `SIGTRAP`)
    - Receiver thread is interrupted and a signal handler is executed (installed using `signal(2)` or `sigaction(2)`)
      - Race conditions due to nested signals
      - Calling non-reentrant functions (e.g. `malloc()`, `printf()`) is undefined behavior
      - Interruption of some syscalls
    - Real-time signals
      - Queued, guaranteed sending order

Classical IPC

- **Anonymous pipes**
- **Named pipes**
  - Persistent uni-directional pipes
    - Same API as files (anonymous pipes)
    - Pipe identification: File system i-node (bound to a directory entry)
    - No identification of senders on the receiver end
      - Writes of data larger than `PIPE_BUF` bytes can be interleaved
  - Windows named pipes
    - Dedicated namespace (Named Pipe File System `\\.\pipe\`)
    - Non-persistent (removed when all clients close the pipe)
    - Anonymous pipes are named pipes with random names

Classical IPC

- **UNIX domain sockets**
  - Reliable bi-directional stream of bytes (akin to TCP), or …
  - Unordered unreliable datagrams (akin to UDP), or …
  - Reliable ordered stream of datagrams between local processes
    - Same API as BSD sockets
    - Socket identification: File system i-node (bound to a directory entry or to an abstract socket namespace)
    - Sending file descriptors (`sendmsg()`, `rescvmsg()`) as ancillary data
      - Rudimentary capabilities

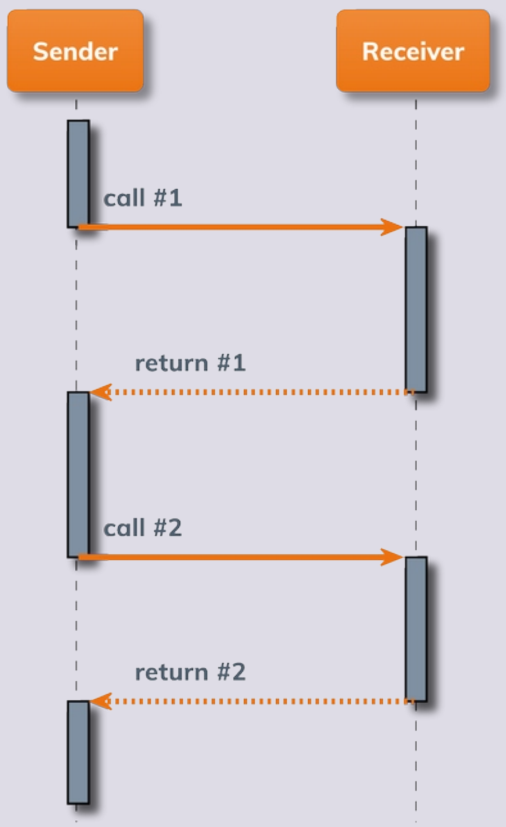
Classical IPC

- **Software shared memory**
  - POSIX Shared Memory, System V Shared Memory
    - Persistent shared memory objects in dedicated namespace
      - In Linux, objects created as tmpfs files (usually `/dev/shm`)
    - `shm_open(3)`, `mmap(2)`, `munmap(2)`, `shm_unlink(3)`
    - `shmget(2)`, `shmat(2)`, `shmdt(2)`
  - Memory mapped files
    - Shared memory backed by a file (or anonymous memory)
    - `mmap(2)`, `munmap(2)`
    - `memfd_create(2)`
      - Removed when no longer referenced
      - File sealing (preventing the other party from changing the configuration)

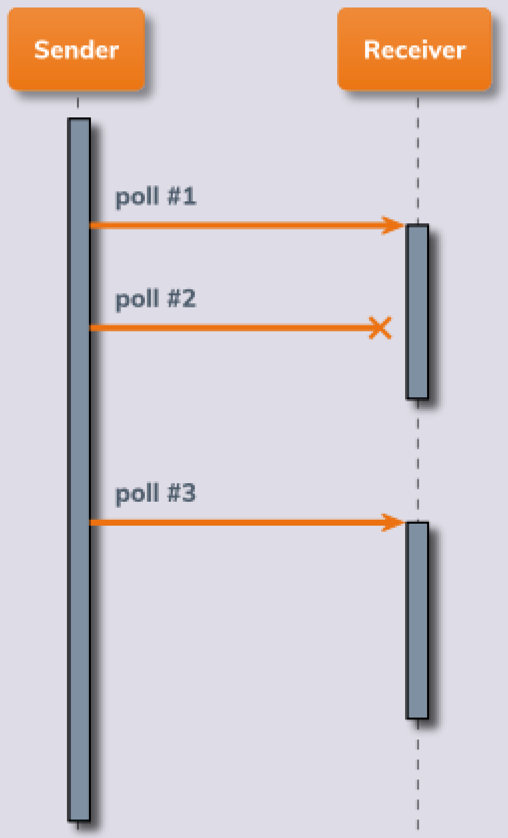
Classical IPC

- **Message passing**
  - Sending
    - Synchronous blocking
      - Sequential processing waiting for reply
    - Synchronous non-blocking
      - Sequential processing not waiting for reply
    - Asynchronous blocking
      - Non-sequential processing waiting for reply
    - Asynchronous non-blocking
      - Non-sequential processing not waiting for reply

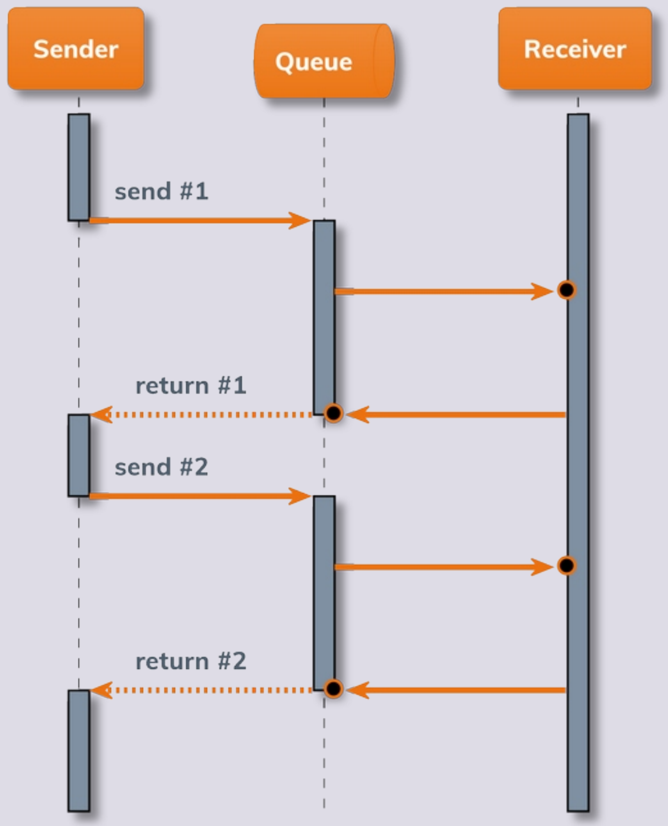
Synchronous Blocking Send



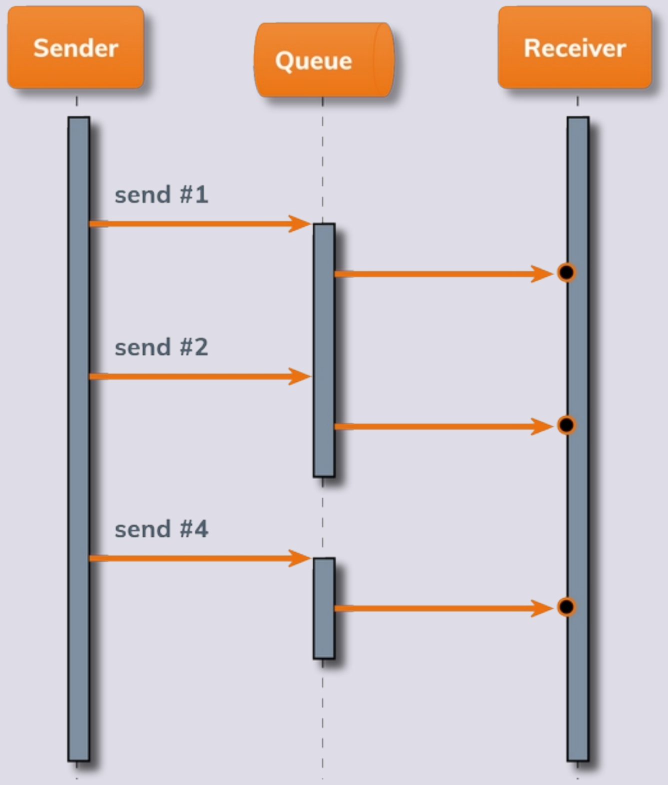
Synchronous Non-blocking Send



Asynchronous Blocking Send



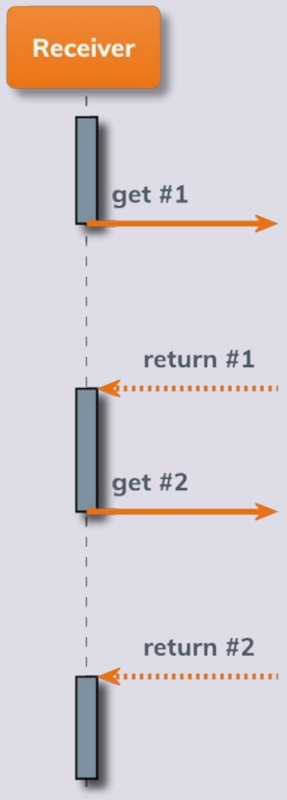
Asynchronous Non-blocking Send



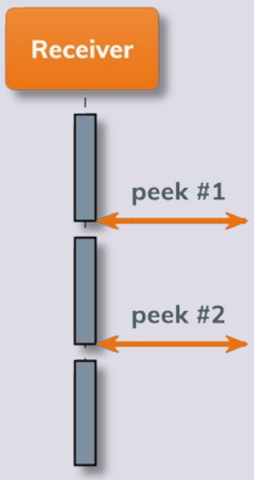
Classical IPC

- **Message passing**
  - Receiving
    - Synchronous blocking
      - Explicit rendez-vous waiting for receiving
    - Synchronous non-blocking
      - Explicit rendez-vous not waiting for receiving
    - Asynchronous
      - Interrupt-style

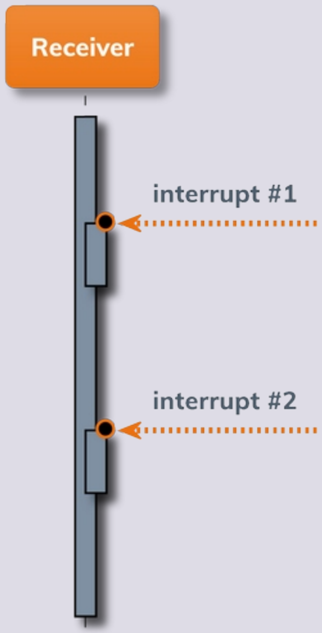
Synchronous Blocking Receive



Synchronous Non-blocking Receive



Asynchronous Receive



Classical IPC

- **Message passing**
  - Addressing
    - Symmetrical
      - Equivalent peers
    - Asymmetrical
      - Explicit client (sender) and receiver (server) roles
    - Direct
      - Peers are explicit endpoints
    - Indirect
      - Peers are hidden behind message queues

Classical IPC

- **Message passing**
  - Transmitting
    - Uniplex
      - Peers take turns in communication
    - Duplex
      - Peers can communicate independently

Classical IPC

- **Message passing**
  - POSIX message queues, System V Message Passing
    - Indirect addressing using a message queue (key for `msgget(2)`, i-node for `mq_open(3)`)
    - `msgsnd(2)`, `mq_send(3)` asynchronous non-blocking (unless the queue is full)
    - `msgrcv(2)`, `mq_receive(3)` synchronous blocking by default
  - Windows Messages
    - Symmetrical addressing using window/thread handles
    - `SendMessage()` synchronous blocking, `SendMessageCallback()`, `SendNotifyMessage()`, `PostMessage()` asynchronous non-blocking
    - `GetMessage()` synchronous blocking, `PeekMessage()` synchronous non-blocking

Mach IPC

- **Prototypical microkernel asynchronous message passing**
  - Ports
    - Receive end-points and associated message queues
  - Port rights
    - Client capabilities for accessing a port (send, receive, send-once)
      - Only a single server can have a receive right
    - Each task has an initial set of port rights
      - Communicating with the kernel, etc.
  - Tagged message structure
    - Kernel enforces type correctness
    - Port rights can be also passed
    - Timeouts

Mach IPC

- **The origin of the "IPC overhead anxiety"**
  - IPC overhead of 50 % compared to monolithic UNIX
    - With a single UNIX server
    - Root causes
      - Complex non-optimized kernel-side code
        - Tagged data type evaluation, handling of timeouts, etc.
        - Dynamic data structures
          - But the implementation only uses linked lists
        - Excessive cache footprint
      - Asynchronicity rarely used for the given workloads
        - User space tasks (mostly ported from UNIX) use synchronous communication and blocking I/O
- **Nowadays, the anxiety is unfounded**
  - Bershad has argued **33 years ago** that the IPC overhead is increasingly irrelevant [1]
    - Real-world performance of computer systems is dominated by other factors
  - Liedtke has shown **30 years ago** that the IPC overhead is negligeable assuming proper microkernel design [2]

The Era of Synchronous IPC

- **L3 (1988), L4 (1993) by Jochen Liedtke**
  - IPC overhead of 3 % compared to monolithic UNIX
    - With a single UNIX server
    - Single IPC call overhead comparable to single syscall overhead in UNIX (approx. 20 times faster than on Mach)
  - Synchronous blocking IPC
    - Explicit client/server rendez-vous and thread migration
      - No need for full context switch (address space switch is sufficient)
      - No buffering, no scheduling, data passed mostly directly in registers
    - Highly target-optimized implementation
      - Small working set, cache-friendly code
      - No complex algorithms or dynamic data structures

The Era of Synchronous IPC

- **L3 (1988), L4 (1993) by Jochen Liedtke**
  - Drawbacks
    - Non-portable microkernel (by design)
      - Poor code readability and maintainability
      - Preoccupation with single-threaded performance conflicts with other goals (e.g. throughput)
    - Design issues of synchronous IPC
      - Unresponsive server blocks the client indefinitely
        - Originally solved using timeouts (in hindsight not a great solution)
      - Asynchronous communication emulated on top of synchronous IPC
        - Abstraction inversion anti-pattern (i.e. requires multithreading)
      - Scalability suffers on modern massively parallel architectures

The Return of Asynchronous IPC

- **The best of both worlds**
  - Synchronous blocking IPC still superior in specific use cases
    - Synchronous blocking semantics, single-core communication
  - Asynchronous IPC reasonably simple, cache-friendly with fast-path kernel code
    - Bounded kernel buffers (additional buffering possible on the client user space side)
    - Intelligent bookkeeping data structures (hash tables, trees)
    - Simple IPC message structure (only integer payload that fits into registers)
      - Additional semantics for memory copying and memory sharing possible
    - Possibility to build rich abstractions in user space
      - Actors, agents, continuations, futures, promises

Case Study: HelenOS IPC

- **Basic design**
  - Message passing over asymmetric connections
    - Sender
      - Asynchronous non-blocking send
      - 6-integer payload (1st integer interpreted as interface/method ID)
      - Bounded kernel buffers
    - Receiver
      - Synchronous blocking receive
      - Every message paired with a reply (6-integer return value)
    - New connections established via existing connections (capabilities)
      - Security policy delegated to the connection brokers
      - Every client initially connected to the Naming Service (default broker)
  - Message forwarding (recursive)
  - Kernel events and hardware interrupts converted to IPC messages (no reply)

Case Study: HelenOS IPC

- **Kernel API**
  - Global method IDs with special semantics
    - `IPC_M_CONNECTION_CLONE` (clone a connection capability from the client to the server)
    - `IPC_M_CONNECT_TO_ME` (establish a callback connection)
    - `IPC_M_CONNECT_ME_TO` (establish a new connection)
      - When forwarded, the connection is potentially established to the next receiver
        - Broker (Naming Service, Location Service, Device Manager, VFS, etc.) connects the client to the target server
    - `IPC_M_SHARE_IN` / `IPC_M_SHARE_OUT` (receive/send a shared virtual address space area)
    - `IPC_M_DATA_READ` / `IPC_M_DATA_WRITE` (receive/send bulk data)
    - `IPC_M_STATE_CHANGE_AUTHORIZE` (update a server state on behalf of a different client)
      - Three-way handshake
    - `IPC_M_PHONE_HUNGUP` (connection close)

Case Study: HelenOS IPC

- **User space API**
  - Async framework
    - Goal: Writing single-threaded sequential client code that makes effective use of the asynchronous IPC
      - User space-scheduled cooperative threads (fibrils)
        - Efficient parallelism (preempted only when blocking on waiting for IPC replies)
    - Abstracting the low-level IPC connections into **sessions**
      - Each session can have a different threading model
    - Abstracting the atomic low-level IPC messages into logical **exchanges**
      - Easily implementing complex communication protocols

Case Study: HelenOS IPC

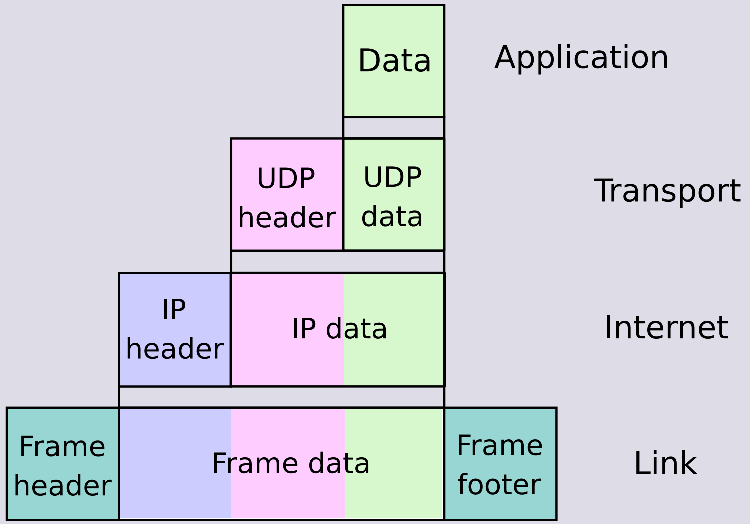
```
async_exch_t *ns_exch = async_exchange_begin(session_ns);

async_sess_t *sess =
    async_connect_me_to_iface(ns_exch, INTERFACE_VFS, SERVICE_VFS, 0);

async_exchange_end(ns_exch);

async_exch_t *exch = async_exchange_begin(sess);

ipc_call_t answer;
aid_t req =
    async_send_3(exch, VFS_IN_OPEN, lflags, oflags, 0, &answer);

async_data_write_start(exch, path, path_size);

async_exchange_end(exch);

// Do some other useful work in the meantime

sysarg_t rc;
async_wait_for(req, &rc);

if (rc == EOK)
    fd = (int) IPC_GET_ARG1(answer);
```

Networking in Operating Systems

- **Socket abstraction**
  - Communication endpoint abstraction
    - In case of IP: [protocol, address, port]
      - Address and port might be implicit or wildcard on the API level
      - Listening socket vs. accepting socket
    - In case of Berkeley API: Socket descriptor as file descriptor
      - Other competing APIs (e.g. STREAMS) almost disappeared
    - Connection-oriented sockets: Socket pair
      - In case of IP: [protocol, source address, source port, destination address, destination port]

Networking in Operating Systems

- **Stacking and encapsulation**
  - RFC 3439: Layering considered harmful



Networking in Operating Systems

- **Breaking layering**
  - Hardware off-loading
    - Checksum calculation, pre-parsing and hashing
      - NIC already touching each octet anyway
  - XDP (eXpress Data Path)
    - Early eBPF packet hook (before any networking stack)
      - Raw data inspection
    - Possibility for hardware off-loading
  - Packet descriptor
    - Structure describing packet data & metadata
      - Pointers to various fields
        - `start`, `size`: Pointing to the raw buffer with focus on current headers

Networking in Operating Systems

- **Performance considerations**
  - DMA scatter-gather into TX/RX ring buffers
    - Packet sizes and page sizes are not divisible
    - Header pushing / popping is not necessarily fixed (IPv6 header chaining)
      - Linear segment with headroom (zero-copy if possible)
      - Non-linear segments
  - Interrupt coalescing
    - Throughput vs. latency
      - Explicit polling architecture
      - Adaptive polling under heavy load
      - Flow aggregation
  - Deferred interrupts

Networking in Operating Systems

- **Performance considerations**
  - TCP state machine
    - Real-time timeouts
    - Local buffering vs. congestion control
      - Original approach: Large RX buffers increase throughput
        - Increase data latency, but also signaling latency
      - Current approach: Latency-bound RX buffers
        - Better queuing discipline
  - Shared resources
    - Flow caches, defragmentation buffers
      - Remotely exploitable
    - Global quotas, zero copy
      - Locally exploitable

References

[1] Bershad B. N.: *The Increasing Irrelevance of IPC Performance for Micro-Kernel-Based Operating Systems*, in Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures, USENIX, 1992, https://dl.acm.org/doi/10.5555/646405.692226

[2] Liedtke J.: *On Micro-Kernel Construction*, in Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), ACM, 1995, https://dl.acm.org/doi/10.1145/224056.224075

Thank you!

Questions?