Advanced Operating Systems
Summer Semester 2023/2024

Martin Děcký
3 Interfaces, Abstractions and Portability
Software & Hardware Interface

- **Address space**
  - Universal abstraction for accessing data (code is a form of data)
  - Physical memory
    - Bytes, words, instructions (or similar)
  - Virtual memory (software / device)
    - Pages (or similar)
  - I/O memory
    - Bytes, words, ports (or similar)
    - Can be embedded in physical memory (memory-mapped I/O)
  - Persistent memory
    - Blocks, pages (or similar)
    - Can be combined with physical memory (non-volatile memory)
  - Object space
    - Keys, capabilities (or similar)
RAM
SSD
L4 Cache
L5 Cache
Physical Memory Myths

- **Random access performance**
  - Seems to be $O(1)$ in time units, but in reality it is closer to $O(\sqrt{n})$
    - Where $n$ is the size of the working set
    - Performance effects of the cache hierarchy

- **Canonical physical address space**
  - Different views of the physical address space
    - Local APIC and SMM on x86, secure/non-secure TrustZone on ARM
    - Embedding of the I/O address space into the MMIO address space on x86
  - Completely disjoint address spaces
    - No central interconnect, but a network of nodes and address translations
Figure 2. Subset of the NXP i.MX8 memory layout. The LPUART is accessed using a different addresses from the M4 cores, the >2GiB memory is only accessible from the A35 cores, the PPB is only reachable from the M4 cores.

Non-Uniform Memory Access

- **Explicitly exposed hardware topology**
  - Processing units, cores, packages
  - NUMA nodes (directly byte-addressable memory)
  - Caches
    - Transparent cache coherency (ccNUMA)
      - MSI, MESI, MESIF, MOSI, MOESI, Dragon, Firefly protocols
      - Directory-based cache coherency
  - Buses and I/O devices

- **Guiding heuristics for placing execution near its working set**
  - `numactl`, `libnuma`
Device Virtual Memory

- Mapping of device-visible addresses to bus-visible addresses
  - Similar purpose to software virtual memory
    - Isolation (i.e. safety, security)
    - Mitigating fragmentation (i.e. scatter-gather functionality)
    - Mitigating address range issues
  - Integrated in the device DMA engine
    - Graphics Address/Aperture Remapping Table
  - Separate IOMMU
    - Device memory paging
    - Usually also implementing interrupt remapping
IOMMU

- **AMD-Vi, ARM SMMU**
- **Intel VT-d**
  - Usually located in the peripheral interconnect (a.k.a. north bridge)
  - Address space is usually associated with a protection domain
    - Endpoint is usually associated with a source ID
    - Data structure that maps source IDs to protection domains
    - Memory mapping using hierarchical page tables
      - First-stage translation page tables essentially equivalent to the CPU page tables
      - Second-stage translation for hypervisor, with nested first & second-stage translation
    - Device TLB for translation caching, other caches
  - ACPI DMAR (DMA Remapping Reporting) table
Physical Memory Management

• **Zones**
  - Continuous address ranges with specific properties
    • Available, reserved, firmware, kernel code/data, etc.
    • Logical properties
      - E.g. < 1 MiB, < 16 MiB, < 4 GiB on x86

• **Allocations**
  - Tracking of used frames and their owner
  - Bitmaps, free lists, buddy allocation, etc.
Capabilities

• **Motivation**
  - Universal and **pure** kernel mechanism for resource management
    • No specific management policy in the kernel
    • Policy decision delegated to user space
    • Delegation (granting) of authority over resources from the original owner to other parties
      - Including granting revocation
Capabilities

• **Usual terminology**
  - **Capability**
    • Object instance representing (identifying) a specific resource
    • Kernel object representing a kernel-managed resource
    • Kernel proxy object identifying a user-managed resource
    • User space object representing a user space resource
  - **Capability reference**
    • Unforgeable identifier (handle) to a capability
      - Possibility to restrict permissions (e.g. permissible operations) and identify ownership
  - **Capability space**
    • Address space of capability references
      - Typically associated with a task
    • Capabilities as local identifiers within their namespace
Capabilities Put Simply

read(0, ...);

- file descriptor table (capability space)
- file descriptor (capability reference)
- open file (capability)
- vfs_file_t

kernel space

user space
Capability Operations

- **Actions performed with capabilities**
  - Can be restricted by the capability reference
    - Multiple capability references can point to the same capability
  - **Invoke**
    - Execute a “business logic” method on the target object
  - **Clone / Mint**
    - Create a duplicate capability reference (possibly with restricted permissions)
  - **Delegate / Grant**
    - Pass a duplicate capability reference (possibly with restricted permissions) to a different capability space
    - In case of granting, the original ownership is kept
    - Only once or recursively
  - **Revoke**
    - Forcefully removing and granted capability reference from other capability spaces
Capability Delegation

task 1:

```c
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// SCM_RIGHTS ancillary message type ...
memmove(CMSG_DATA(cmsg), &fd, sizeof(fd));
sendmsg(socket, &msg, 0);
```

task 2:

```
0 1 2 3
```

user space

```
0 1 2 3
```

kernel space

```
vfs_file_t
```

0 1 2 3
Capability Delegation

**task 1:**

```c
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// SCM_RIGHTS ancillary message type ...
memmove(CMSG_DATA(cmsg), &fd, sizeof(fd));
sendmsg(socket, &msg, 0);
```

**task 2:**

```c
struct msghdr msg;
struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
// ...
recvmsg(socket, &msg, 0);
int fd;
memmove(&fd, CMSG_DATA(cmsg), sizeof(fd));
```
Physical Memory Management

- **Representing physical memory as capabilities**
  - Chicken & egg problem: Capabilities, capability spaces, page tables and other bookkeeping structures require memory for storage (i.e. capabilities)
  - Recursive solution: Type hierarchy of capabilities
    - Untyped memory capability type
      - Representing a range of physical memory
      - Initially a single capability representing the entire physical memory
      - Untyped capabilities be **derived** ...
        - ... into multiple untyped capabilities (recursively splitting the physical memory)
        - ... into capabilities of other types
          - Providing the memory for capability storage and bookkeeping
          - Providing memory for other kernel objects
Capability References and Spaces

- **Naked capabilities**
  - Capability references identify capabilities directly
    - E.g. physical memory addresses identifying untyped memory capabilities

- **Encapsulated capabilities**
  - Capability references need to be mapped to capabilities
  - Mapping database of capability space
    - Fast lookup of capability references (most frequent operation)
    - Reasonably fast creation / removal of capability references
    - Low memory overhead and fragmentation (sparse capability space)
    - Additional metadata (permissions, delegation, granting)
    - Possibility for in-line storage of actual kernel objects (up to a certain size)
Capability References and Spaces

- **Capability space (cspace)**
  - Directed graph of capability nodes
    - Can be implicit (no explicit object representation)

- **Capability node (cnode)**
  - Array of capability slots
    - Empty slot
    - Slot pointing to a specific capability
    - Slot pointing to a cnode
      - Hierarchical organization of capability nodes
      - Radix tree indexing
Hierarchical Capability Mapping Database

00 01 11 cref_t

cnode_t (10 bit index)

cnode_t (10 bit index)

cnode_t (12 bit index)

mem_region_t

resource

cspace

kernel space

user space
Capabilities Example: seL4

- **Kernel objects**
  - UntypedObject (physical memory range)
  - TCBObject (thread)
  - EndpointObject (IPC calls destination)
  - AsyncEndpointObject (signal recipient)
  - CapTableObject (array of capabilities)
  - X86_4K (4 KiB frame)
  - X86_4M (4 MiB frame)
  - X86_PageTableObject (2\textsuperscript{nd} level page table)
  - X86_PageDirectoryObject (1\textsuperscript{st} level page table)
Capabilities Example: seL4

- Capability derivation

```c
seL4_X86_Untyped_Retype(cnode_selector(phys_addr), seL4_X86_4K, ..., ..., ..., ..., phys_addr >> FRAME_WIDTH, 1);

seL4_X86_Untyped_Retype(cnode_selector(pt_phys_addr), seL4_X86_PageTableObject, ..., ..., ..., ..., pt_phys_addr >> FRAME_WIDTH, 1);

seL4_X86_Untyped_Retype(cnode_selector(pd_phys_addr), seL4_X86_PageDirectoryObject, ..., ..., ..., ..., pd_phys_addr >> FRAME_WIDTH, 1);

seL4_X86_PageTable_Map(cnode_selector(pt_phys_addr), cnode_selector(pd_phys_addr), virt_addr, seL4_X86_Default_VMAttributes);

seL4_X86_Page_Map(cnode_selector(phys_addr), cnode_selector(pd_phys_addr), virt_addr, seL4_AllRights, seL4_X86_Default_VMAttributes);
```
Capabilities Example: seL4

- Resources fully delegated, allows autonomous operation
- Strong isolation, No shared kernel resources

Resource Manager
- RM Data
- Addr Space

Resource Manager
- RM Data
- Addr Space

init Task = Global Resource Manager
- Kernel Data
- GRM Data

Source: Heiser G.: Introduction: Using seL4
Courtesy of Gernot Heiser, UNSW Sydney, CC BY 4.0,
Physical Memory Management Comparison

- **Traditional**
  - Straightforward API
  - High-level abstraction
  - Portable
  - Implicit policy
  - Accounting out of scope
  - Delegation out of scope

- **Capability-based**
  - No implicit policy (policy set completely by the client)
  - Accounting and delegation within the scope
  - Low-level API
  - Potential abstraction inversion
  - Non-portable
Note on Memory Accounting

- **Strict memory reservation**
  - Sum of virtual memory sizes < Sum of physical memory sizes
    - Swap space counted as physical memory
  - In-bound out-of-memory condition
  - More predictable
  - Potential inefficient resource usage

- **Memory overcommit**
  - Sum of resident memory sizes < Sum of physical memory sizes
    - Decoupling memory mapping from memory allocation
  - Support for large sparse virtual address spaces
    - Potentially more efficient resource usage
  - Out-of-bound out-of-memory condition
    - Victim finding
  - Less predictable
Note on Caches

- **Separate instruction and data caches**
  - Self-modifying code (N.B.: including code loading)

- **Virtually-indexed caches**
  - Mostly used for L1 instruction caches nowadays
  - Cache homonyms (same VPN referring to different PFN)
    - Flush on each address space switch costly
    - Distinct virtual addresses unpractical
    - ASID tagging (ASID management by operating system)
  - Cache synonyms (different VPN referring to same PFN)
    - Shared memory or multiple mappings leading to stale data
    - Synonym detection, cache coloring
    - Hardware synonym detection
Non-Volatile Memory

• Historically biased towards rotational media
  - Cylinder / Head / Sector → Linear (Logical) Block Addressing
    • Originally interface abstraction not very high
      - Hard sectored → Soft sectored (with remapping)
    • 512 B blocks → 4096 B blocks (floppy/hard drives)
    • 2048 B blocks (optical drives), 2353 B blocks (raw optical drives)
  - Latency several orders of magnitude larger than volatile memory
    • Originally interface I/O efficiency not very important
      - Single tenant
      - Single request stream
Non-Volatile Memory

- Historically biased towards rotational media
Non-Volatile Memory

- Historically biased towards rotational media
  - Multi-tenant performance dominated by physical seek time
  - Still mostly via single request stream
    - Software I/O scheduling (shortest seek first, elevator/sweep, shortest deadline first, etc.)
      - Might not have the most accurate physical storage information (i.e. remapping)
    - I/O command batching (queuing)
      - Leaving the optimal I/O order (within the batch) to hardware
      - Incorporates interrupt coalescing
Non-Volatile Memory

- **Solid-state drives**
  - Differing characteristics from rotational drives
    - Physical characteristics mostly unimportant
    - Addressing characteristics
      - Different native read/write and erase blocks
        - Write amplification
      - Physical addressing more like volatile memory
    - Latency much closer to volatile memory
      - Performance dominated by interface I/O efficiency
  - High degree of internal parallelism
  - Unique wear characteristics
Non-Volatile Memory

• **Solid-state drives**
  - Reflection in the I/O interface (e.g. NVMe)
    • Generally provides the common LBA abstraction
      - Wear leveling, block remapping and garbage collection in hardware Flash Transition Layer (FTL)
        • Frequently implemented as multi-level log-based storage
        • Software trim hint to indicate unused (erased) blocks
        • Trade-offs between write amplification, performance, idle characteristics
  - Low latency and parallel access
    - “Unlimited” request queues with lock-less access
    - “Unlimited” command queuing
    - Interrupt coalescing & multiple interrupt groups
    - Full-duplex scatter-gather DMA
Non-Volatile Memory

- **Solid-state drives**
  - Exposing more of the hardware architecture to software
  - **Addressing**
    - Open-channel SSD
    - NVMe Zoned Namespace
      - Note: Zones also useful for Shingled Magnetic Recording (SMR)
  - Compute off-loading
    - Basic NVMe I/O commands: Compare, Write Zeroes, Copy
    - NVMe Key Value command set
    - Near data computing (proposed)
Storage Near Data Computing

- **Off-loading computation to storage controller**
  - Decrease latency, improve throughput, decrease energy consumption
  - Improve performance
    - Trade-off: Lower performance of embedded cores
      - Still a performance boost when compute cores are already loaded

Memory Near Data Computing

Control logic

Row decoder

Address

Memory matrix

Sense amps

Y-gating

Data

RAS

CAS

WE
Memory Near Data Computing

- Address
- Control logic
- RAS
- CAS
- WE

Decoder

Opcode

Memory matrix

Sense amps

Filtering / computing

Data

Addressing / computing

Op-code

Sense amps

Filtering / computing

Data
Memory Near Data Computing

- Address
- Control Logic
- RAS
- CAS
- WE
- Opcode
- Row Decoder
- Memory Matrix
- Sense Amps with Gates
- Y-gating
- Data
Generic Near Data Computing

Host

Memory

Processing
In-Memory
(PIM)

Storage

In-Storage
Computing
(ISC)

Network

On-Stream
Processing
(OSP)
Generic Near Data Computing

• **Current work-in-progress**
  - Universal open interface standards
    - Currently extensions of existing I/O interfaces (NVMe)
    - Compute Express Link (CXL)

• **Open questions**
  - Universal programming model
    - Stream / flow processing
    - Association of compute units with data
  - Universal compute model
    - ISA
    - Safety and security considerations
  - Off-loading vs. distributed computing
Kernel Interfaces

- Most monolithic kernels and microkernels 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, interfaces, etc.
  - All software engineering best practices apply here
    - 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
    - Etc.
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 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)

- **ioctl**s
  - Each object (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
      - **ioctl**s 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 ioctl, 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 ioctl 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)

• ... a memory area
  - All resources represented as (demand mapped) virtual memory
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 Driver 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 system calls
  - 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()`, `recvmsg()`) 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 / asynchronous, blocking / non-blocking
  - Receiving: synchronous / asynchronous, blocking / non-blocking
  - Addressing: symmetrical / asymmetrical / indirect
  - Transmitting: uniplex / duplex
  - 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 non-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 31 years ago that the IPC overhead is increasingly irrelevant [1]
    • Real-world performance of computer systems is dominated by other factors
  - Liedtke has shown 28 years ago that the IPC overhead is negligible assuming proper microkernel design [2]
The Era of Synchronous IPC

  - 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 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

  - **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 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 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
HelenOS IPC

- **Basic design**
  - Asynchronous message passing over uni-directional connections
    - 6 INTEGER payload (1st integer interpreted as interface/method ID)
    - Bounded kernel buffers
    - 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)
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)
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
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);
References


Thank you!

Questions?