MICROKERNEL-BASED AND CAPABILITY-BASED OPERATING SYSTEMS

Martin Děcký
martin.decky@huawei.com

March 2021
About the Speaker

**Charles University**
- Graduated (Ph.D.) in 2015
- Co-author of the HelenOS ([http://www.helenos.org/](http://www.helenos.org/)) microkernel multiserver operating system

**Huawei Technologies**
- Senior Research Engineer, Munich Research Center (2017 – 2018)
- Principal Research Engineer, Dresden Research Center (2019 – present)
Since 2019, ~20 employees (plus a virtualization team in Munich)
Focuses on R&D in the domain of operating systems

- Microkernels, hypervisors
  - Collaboration with the *OS Kernel Lab* in Huawei HQ
  - Collaboration with TU Dresden, MPI-SWS, ETH Zürich and other institutions
- Formal verification of correctness, weak memory architectures
- Safety and security certification
- Many-core scalability, heterogeneous hardware
- Flexible OS architecture
We Are Hiring

- **Operating System Engineer / Researcher** (Dresden)
  - [https://apply.workable.com/huawei-16/j/3BAC3458E6/](https://apply.workable.com/huawei-16/j/3BAC3458E6/)

- **Formal Verification Engineer / Researcher** (Dresden)
  - [https://apply.workable.com/huawei-16/j/95CCAD4EC5/](https://apply.workable.com/huawei-16/j/95CCAD4EC5/)

- **Virtualization Engineer / Researcher** (Munich)
  - [https://apply.workable.com/huawei-16/j/51F90678EA/](https://apply.workable.com/huawei-16/j/51F90678EA/)

- **Industrial Ph.D. Student** (Dresden)
  - In collaboration with TU Dresden
March 30th – 31st 2021

https://huawei-events.de/, on-line, no participation fee
MICROKERNELS
Microkernel-based Operating Systems

**Motivation**
- Safety, security, reliability, dependability
  - Proper software architecture
  - Formal verification of correctness
- Modularity, customization
- Virtualization, paravirtualization
  - Tasks and virtual machines are quite similar types of entities
- Partitioning, support for mixed criticality
Monolithic OS Design Is Flawed


- “While intuitive, the benefits of the small TCB have not been quantified to date. We address this by a study of critical Linux CVEs, where we examine whether they would be prevented or mitigated by a microkernel-based design. We find that almost all exploits are at least mitigated to less than critical severity, and 40% completely eliminated by an OS design based on a verified microkernel, such as seL4.”

  - https://dl.acm.org/doi/10.1145/3265723.3265733
Some Data Points from History

- **Compatible Time-Sharing System (CTSS)**
  - John McCarthy, MIT Computation Center, 1961
    - Probably one of the earliest “real” operating system
      - Not just a loader, jobs manager or batch manager

- **RC 4000 Multiprogramming System**
  - Per Brinch Hansen, Regnecentralen, 1969
    - Separation of mechanism and policy, modularity via isolated concurrently running processes, message passing

- **Multics**
  - MIT, General Electric, Bell Labs, 1969
    - Traceable influence on UNIX
Some Data Points from History (2)

- **HYDRA**
  - William Wulf, Carnegie Mellon University, 1971
    - Capability-based, object-oriented, separation of mechanism and policy
    - Probably the earliest peer-reviewed publication of the design principles

- **UNIX**
  - Ken Thompson, Dennis Ritchie, Brian Kernighan et al., Bell Labs, 1973
    - Architecture and design traceable in many current monolithic systems

- **VMS**
  - Digital Equipment, 1977
    - Architecture and design traceable in Microsoft Windows
Some Data Points from History (3)

- **EUMEL / L2**
  - Jochen Liedtke, University of Bielefeld, 1979
    - Proto-microkernel based on bitcode virtual machines

- **QNX**
  - Gordon Bell, Dan Dodge, 1982
    - Earliest commercially successful microkernel multiserver OS
      - Still in active use and development today

- **CMU Mach**
  - Richard Rashid, Avie Tevanian, Carnegie Mellon University, 1985
    - Arguably the most widespread microkernel code base
      - Still a core part of macOS, iOS and other OS clones by Apple today (but not in a microkernel configuration)
      - Despite its well-publicized shortcomings, it remains highly influential
Microkernel-based Operating Systems

**Definition**

- Operating system that follows specific design principles that, in effect, minimize the amount of code running in the privileged (kernel) mode
  - Hence the name
- Every microkernel-based OS follows slightly different specific design principles
  - Two design principles are probably universally common
    - Minimality principle
    - Split of mechanism and policy principle
Minimality Principle

- **The obvious criterion**
  - The kernel needs to implement the functionality than cannot be possibly implemented in user space
  - On typical commodity hardware, this includes
    - Bootstrapping
    - Fundamental part of hardware exception and interrupt handling
    - Configuration of certain control registers (possibly including MMU)
    - Fundamental part of mode switching (e.g. related to hardware virtualization, trusted execution environments, etc.)
Minimality Principle (2)

- **The necessary criterion**

  - The kernel needs to implement the functionality than cannot be delegated only to a trusted user space component without also delegating it to any untrusted user space component (thus undermining the fundamental guarantees that the operating system provides)

  - On typical commodity hardware, this includes
    - Configuration of the forced preemption mechanism (e.g. timer interrupt routing)
    - Fundamental part of interacting with a hypervisor, firmware and some hardware components
      - Hardware components are tricky: Without IOMMU, almost any interaction with hardware might potentially undermine the OS guarantees
Minimality Principle (3)

- **The practicality criterion**
  - The kernel *might* also implement the functionality that would be unpractical (while still technically possible) to be safely delegated to user space
  - This is where microkernels differ, but there are still some universal examples
    - Context switching
    - Basic scheduling
    - System timer configuration
    - Observability and (optional) debugging support
Split of Mechanism and Policy Principle

- **Orthogonal to the minimality principle**
  - The microkernel is not an indivisible entity
    - Composed of instructions, basic blocks, language constructs, etc.
    - The code inevitably follows some patterns that form architecture, design, abstractions, parametrization, etc.
  - Separation of concerns
    - The kernel implements only pure and universal mechanisms ("the what") while the policies ("the how/when") are delegated to user space
      - This is where microkernels differ
        - Does "arbitrary policy" equal "no policy"?
        - Is it fine to have a default (but replaceable) policy?
Practical Differences

- **Monolithic kernel**
  - Configurability via compile-time options and parametrization
  - Modularity via run-time dynamic linking
  - Tight module coupling, weak module cohesion
  - Structure is implicit and not enforced (especially at run time)

- **Microkernel**
  - Configurability via different use (policy in user space)
  - Modularity via extension in user space
  - Loose module coupling, strong module cohesion
  - Structure is explicit and enforced (even at run time)
Design Space of Operating Systems
Design Space of Operating Systems

- monolithic components
- fine-grained components
Design Space of Operating Systems

- Monolithic components
- Fine-grained components
- Raw performance
- Safety via isolation

Microkernel-based and Capability-based Operating Systems
Design Space of Operating Systems

- Static deployment
- Safety via isolation
- Monolithic components
- Raw performance
- Fine-grained components
- Dynamic deployment
- Static deployment
- Safety via isolation
- Monolithic components
- Raw performance
- Fine-grained components
- Dynamic deployment
Design Space of Operating Systems
Monolithic OS

- Application
- Monolithic kernel
- Memory management
- Scheduler
- IPC
- Device drivers
- File system drivers
- User management
- Network stack
- Hardware

Unprivileged mode

Privileged mode
Architecture of User Space

Single-server microkernel OS

- Application
- System server
- Microkernel
- Memory management
- Scheduler
- IPC
- Device drivers
- File system drivers
- User management
- Network stack

Unprivileged mode
Privileged mode

Hardware
Multiserver microkernel OS

- Application
- Network stack
- Security server
- Naming server
- Location server
- Device multiplexer
- Device driver server
- File system multiplexer
- File system driver server
- Memory management
- Scheduler
- IPC
- Microkernel
- Hardware

Unprivileged mode
Privileged mode
Architecture of User Space

- **Type-1 hypervisor**

![Diagram](image)
Type-1 hypervisor (in common deployment)
Architecture of User Space

- Hypervisor with unikernels

```
unikernel
  app component
  kernel component

unikernel
  app component
  kernel component

unikernel
  app component
  kernel component

memory mgmt
scheduler
comm
hypervisor

hardware

privileged mode
hyper-privileged mode
```
Architecture of User Space

- Multiserver microkernel with unikernels for device drivers
Architecture of User Space

**Multikernel**

- **Application**
- **Server**
- **Kernel**
- **CPU**

Diagram showing the architecture of user space with multiple kernels and applications in both privileged and unprivileged modes.
CAPABILITIES
Capabilities

- **Motivation**
  - A *universal* and *pure* mechanism in the kernel to safely manage (all) operating system resources
    - Without implementing any specific management policy in the kernel (i.e. delegating the management policy completely to user space)
  - Potential secondary goal
    - Possibility to grant or delegate (parts of) the authority over resources from the original owner of a resource to other users
      - In a controllable fashion (i.e. including the possibility of revocation)
Capabilities

Definition

- **Capability**
  - Object (instance of a given object type) identifying some specific (operating system) resource
    - Kernel object identifying a kernel-managed resource
    - Kernel object (proxy object) identifying a user space resource
    - User space object identifying a user space resource

- **Capability reference**
  - Unforgeable identifier (handle) to a capability
    - Might be associated with permissions (e.g. permissible operations, methods) and ownership

- **Capability space**
  - Each capability reference is local to a specific namespace (typically associated with a specific task, process) and does not have any meaning in other namespaces
    - Akin to (virtual) address space
What Are Capabilities, Anyway?

read(0, ...);

user space

kernel space

0 1 2 3

file descriptor table
(capabilities in capability space)

vfs_file_t

operating system resource
(open file)
Capability Granting

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

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

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

user space

kernel space

vfs_file_t

0 1 2 3

0 1 2 3 4
Chicken & Egg Problem

What if we want to represent all resources as capabilities?

- Even the resource (memory) needed to store the capabilities and capability references is a capability
  - We start with some basic capability (untyped capability) that represents (physical) memory
    - Encapsulated capability vs. naked capability
    - This capability can be retyped to a different capability or converted to multiple capabilities
      - Allocating kernel objects
      - Allocating capability nodes that bind capability references to capabilities
  - Bookkeeping objects (e.g. memory for page tables) might also be represented as capabilities
Capability Derivation Tree

Permissible ways of retyping capabilities

```
untyped cap
  └── untyped cap
    ├── untyped cap
    │   └── cnode cap
    └── untyped cap
        └── TCB cap

untyped cap
  └── untyped cap
    └── L1 PT cap

untyped cap
  └── untyped cap
    └── L2 PT cap

10 pages
  6 pages
  2 pages
  1 page
  1 page
```
Representing Capability Space

- **Effective and efficient storage for capability nodes**
  - **Criteria**
    - Low memory overhead and fragmentation even for sparse capability spaces
    - Fast lookup of capability references (typically the most frequent operation)
    - Reasonably fast creation and removal of new capability references
    - Possibility to store metadata (e.g., permissions, ownership/delegations) and even actual kernel objects (up to a certain size) in-line
  - **Typical candidates**
    - Arrays
    - Hash tables
    - Radix trees
Hierarchical Capability Space

- `00 01 11` refers to `cref_t`
- `cnode cap` with `cnode_t (10 bit index)`
- `untyped cap` with `cnode_t (10 bit index)`
- `endpoint cap` with `cnode_t (10 bit index)`
- `page cap` with `cnode_t (10 bit index)`
- `untyped cap` with `cnode_t (10 bit index)`
- `mem_region_t` with `resource`
Capability Operations

- **Actions that can be performed with capabilities**
  - The permissible set of operations might be defined/restricted by the capability reference itself
    - Each capability reference might permit different methods despite pointing to the same object
  - **Invoke**
    - Executing some “business logic” operation on the target object
  - **Clone**
    - Creating a duplicate capability reference
  - **Mint**
    - Creating a duplicate capability reference, but with restricted permissions
Capability Operations (2)

- **Derive**
  - Retyping the capability to a different capability type or converting it to multiple capabilities
    - Permissible retyping/conversions defined by the capability derivation tree
- **Delegate**
  - Passing the ownership of the capability reference to different capability space
- **Grant**
  - Creating a duplicate capability reference (possibly with restricted permissions) in a different capability space (while keeping ownership)
    - Might be done only once or recursively
- **Revoke**
  - Removing a granted capability reference from a different capability space
GET TO KNOW MICROKERNELS
Microkernels are operating systems that outsource the traditional operating system functionality to ordinary user processes while providing them with mechanisms requisite for implementing it. Microkernel-based operating systems come in many different flavours, each having a distinctive set of goals, features and approaches. Some of the most often cited reasons for structuring the system as a microkernel is flexibility, security and fault tolerance. Many microkernels can take on the role of a hypervisor too. Microkernels and their user environments are most often implemented in the C or C++ programming languages with a little bit of assembly, but other implementation languages are possible too. In fact, each component of a microkernel-based system can be implemented in a different programming language.

Here is a list of active free, open source microkernel projects. If your project is missing or this page needs fixing, please create a pull request!

**Escape**

A UNIX-like microkernel operating system, that runs on x86, x86_64, ECO32 and MMIX. It is implemented from scratch and uses nearly no third-party components. To fit nicely into the UNIX philosophy, Escape uses a virtual file system to provide drivers and services. Both can present themselves as a file system or file to the user. ([github.com/Nils-TUD/Escape](https://github.com/Nils-TUD/Escape))

**M³**

A microkernel-based system for heterogeneous manycores, that is developed as a hardware/OS co-design at the TU Dresden. It aims to support arbitrary cores (general purpose cores, DSPs, FPGAs, ASICs, ...) as first-class citizens. This is achieved by abstracting the heterogeneity of the cores via a new hardware component per core, called data transfer unit. ([github.com/TUD-OS/M³](https://github.com/TUD-OS/M³))
Microkernel-based and Capability-based Operating Systems
HelenOS Microkernel Functional Blocks
HelenOS User Space Architecture

- **File System Drivers**
  - TMPFS
  - Location FS
  - ISO 9660
  - UDF
  - MINIX FS
  - FAT
  - exFAT
  - ext4

- **Device Drivers**
  - vterm
  - bdsh

- **Device Manager**
  - vfs

- **Networking Management**
  - nconfsrv
  - dnrsv
  - dhcp

- **Link Layer Protocols**
  - slip
  - loopip
  - ethip

- **Transport Layer Protocols**
  - tcp
  - udp

- **Human Interface**
  - remote console
  - remote framebuffer
  - console
  - compositor
  - clipboard
  - audio
  - input
  - output

- **Kernel**
  - location service
  - logger
  - klog
  - loader
  - task monitor
  - init
  - kernel

- **Other Components**
  - inetsrv
  - location service
  - service loader
  - task monitor
  - init
Genode OS Framework

Further Reading


- Matthias Lange: *The impact of Meltre and Specdown on microkernel systems (*)&, Microkernel Devroom, FOSDEM, 2019
  - (*) Deliberate misspelling of Meltdown and Spectre
Q&A
THANK YOU!