

Microkernel Construction

Introduction

Nils Asmussen

04/12/2018

- **Introduction**

- Goals
- Administration

- Monolithic vs. Microkernel

- Overview About L4/NOVA

- ➊ Provide deeper understanding of OS mechanisms
- ➋ Look at the implementation details of microkernels
- ➌ Make you become enthusiastic microkernel hackers
- ➍ Propaganda for OS research at TU Dresden

- Thursday, 4th DS, 2 SWS

- Slides:

`www.tudos.org` → Teaching → Microkernel Construction

- Subscribe to our mailing list:

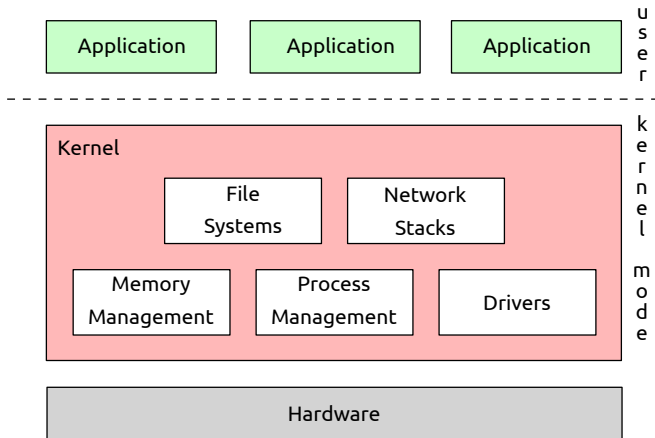
`www.tudos.org/mailman/listinfo/mkc2018`

- In winter term:

- Microkernel-based operating systems (MOS)
- Various labs

- Introduction
- **Monolithic vs. Microkernel**
 - Kernel design comparison
 - Examples for microkernel-based systems
 - Vision vs. Reality
 - Challenges
- Overview About L4/NOVA

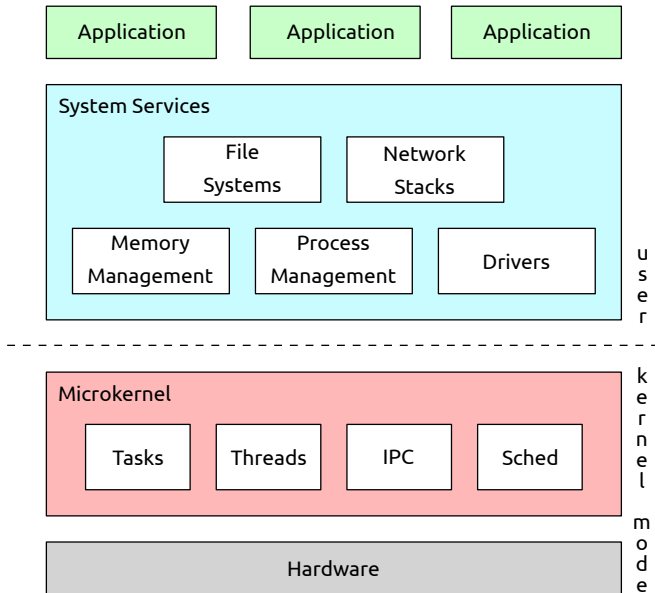
Monolithic Kernel System Design



Monolithic Kernel OS (Propaganda)

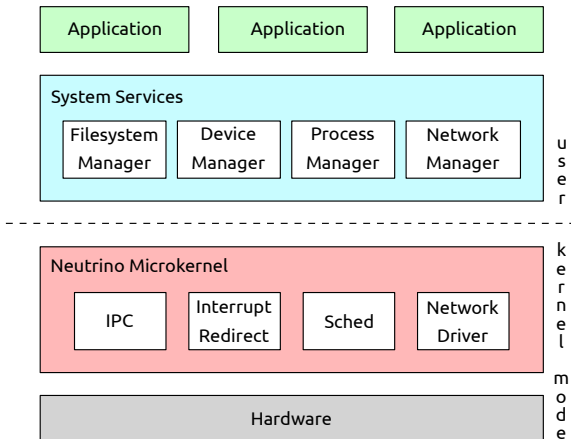
- **System components run in privileged mode**
 - No protection between system components
 - Faulty driver can crash the whole system
 - Malicious app could exploit bug in faulty driver
 - More than 2/3 of today's OS code are drivers
 - No need for good system design
 - Direct access to data structures
 - Undocumented and frequently changing interfaces
 - Big and inflexible
 - Difficult to replace system components
 - Difficult to understand and maintain
- Why something different?
 - Increasingly difficult to manage growing OS complexity

Microkernel System Design



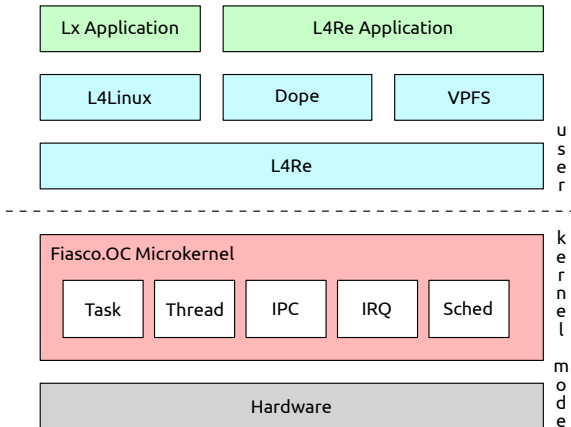
Example: QNX on Neutrino

- 1 Commercial, targets embedded systems
- 2 Network transparency



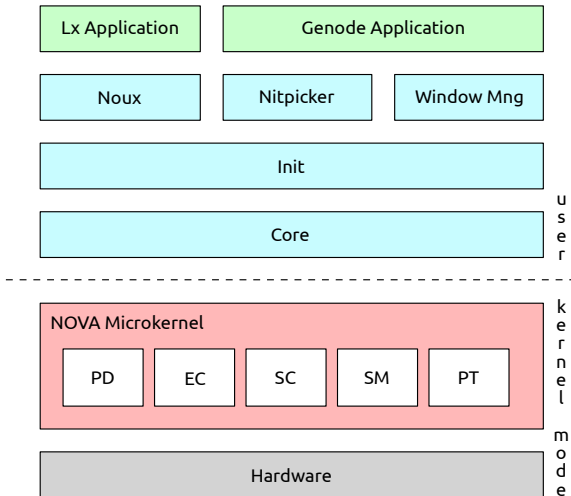
Example: L4Re on Fiasco.OC

- 1 Developed at our chair, now at Kernkonzept
- 2 Belongs to the L4 family



Example: Genode on NOVA

- 1 Genode is a spin-off of the chair
- 2 NOVA was built at our chair



Vision vs. Reality

- Flexibility and Customizable

- Monolithic kernels are typically modular

- Maintainability and complexity

- Monolithic kernels have layered architecture

- ✓ Robustness

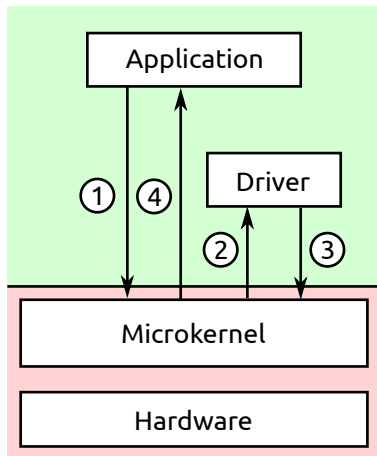
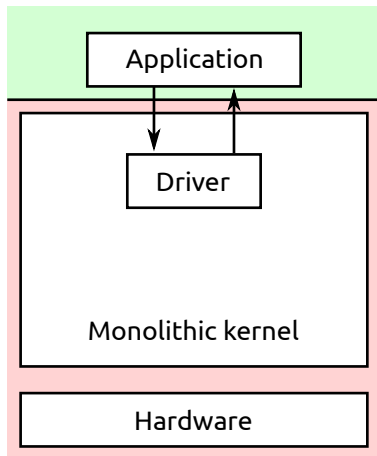
- Microkernels are superior due to isolated system components
- Trusted code size
 - NOVA: 9.000 LOC
 - Linux: > 1.000.000 LOC (without drivers, arch, fs)

- ✗ Performance

- Application performance degraded
- Communication overhead (see next slides)

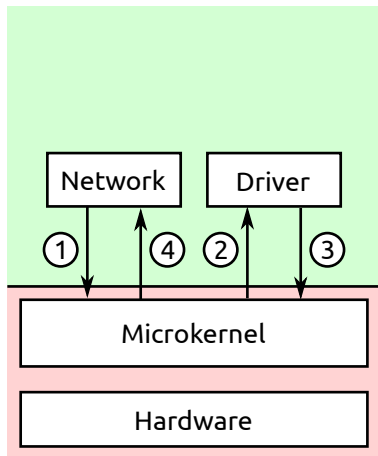
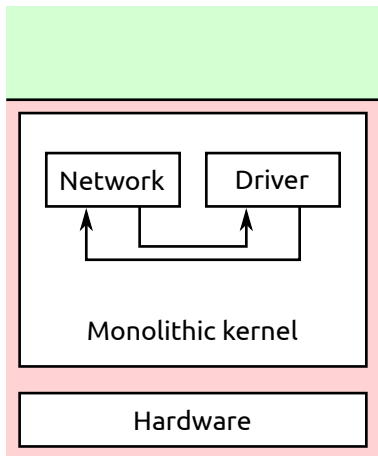
Performance vs. Robustness (1)

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



Performance vs. Robustness (2)

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



① Build functionally powerful and fast microkernels

- Provide abstractions and mechanisms
- Fast communication primitive (IPC)
- Fast context switches and kernel entries/exits

→ Subject of this lecture

② Build efficient OS services

- Memory management
- Synchronization
- Device drivers
- File systems
- Communication interfaces

→ Subject of lecture "Microkernel-based operating systems"

- Introduction
- Monolithic vs. Microkernel
- **Overview About L4/NOVA**
 - Introduction
 - Kernel Objects
 - Capabilities
 - IPC

- ❶ Originally developed by Jochen Liedtke
(GMD / IBM Research)
- ❷ Current development:
 - UNSW/NICTA/OKLABS: OKL4, seL4
 - TU Dresden: Fiasco.OC, NOVA, (M³)
- ❸ Support for hardware architectures:
 - x86, ARM, ...

More Microkernels (incomplete)

① Commercial kernels:

- Singularity @ Microsoft Research
- K42 @ IBM Research
- velOSity/INTEGRITY @ Green Hills Software
- Chorus/ChorusOS @ Sun Microsystems
- PikeOS @ SYSGO AG

② Research kernels

- EROS/CoyotOS @ John Hopkins University
- Minix @ FU Amsterdam
- Amoeba @ FU Amsterdam
- Pebble @ Bell Labs
- Grasshopper @ University of Sterling
- Flux/Fluke @ University of Utah
- Pistachio @ KIT
- Barrelfish @ ETH Zurich

- ① Jochen Liedtke: “A microkernel does no real work”
 - Kernel provides only inevitable mechanisms
 - No policies implemented in the kernel
- ② Abstractions
 - Tasks with address spaces
 - Threads executing programs/code
- ③ Mechanisms
 - Resource access control
 - Scheduling
 - Communication (IPC)

Protection Domain (PD)

- PD is a resource container
 - Object capabilities (e.g., PD, execution context, ...)
 - Memory capabilities (pages)
 - I/O port capabilities (NOVA runs only on x86)
- Capabilities can be exchanged between PDs
- Typically, PD contains one or more execution contexts
- Not hierarchical (in the kernel)

NOVA to Fiasco.OC

Protection Domain \simeq Task

Execution Context (EC)

- EC is the entity that executes code
 - User code (application)
 - Kernel code (syscalls, pagefaults, IRQs, exceptions)
- Has a user thread control block (UTCB) for IPC
- Belongs to exactly one PD
- Receives time to execute from scheduling contexts
- Pinned on a CPU (not migratable)
- Three variants: Local EC, Global EC and VCPU

NOVA to Fiasco.OC

Execution Context + Scheduling Context \simeq Thread

Scheduling Context (SC)

- SC supplies an EC with time
- Has a budget and a priority
- NOVA schedules SCs in round robin fashion
- Scheduling an SC, activates the associated EC

NOVA to Fiasco.OC

Execution Context + Scheduling Context \simeq Thread

Portal (PT)

- A portal is an endpoint for synchronous IPC
- Each portal belongs to exactly one (Local) EC
- Calling a portal, transfers control to the associated EC
- Data and capability exchange via UTCB
- No cross-core IPC

NOVA to Fiasco.OC

Portal \simeq IPC Gate

Semaphore (SM)

- A semaphore offers asynchronous communication (one bit)
- Supports: up, down and zero
- Can be used cross-core
- Hardware interrupts are represented as semaphores

NOVA to Fiasco.OC

Semaphore \simeq IRQ

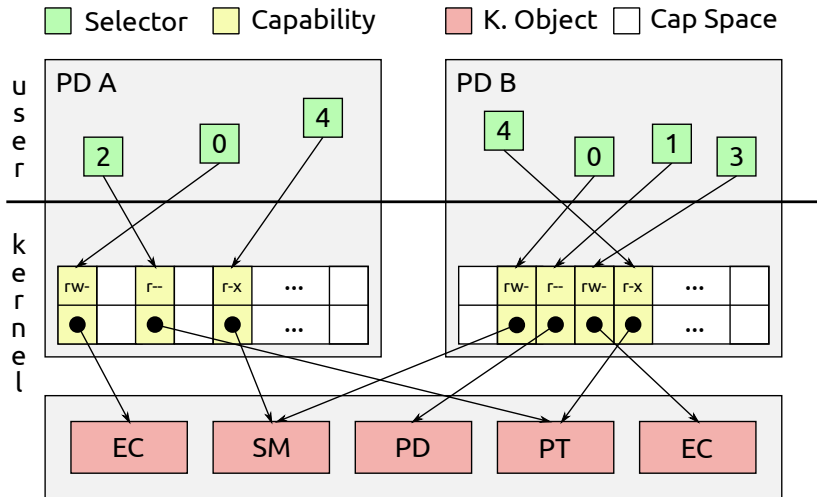
Capabilities

- Access to kernel objects is provided by capabilities
- Capability is a pair: (pointer to kernel object, permissions)
- Every PD has its own capability space (local, isolated)
- Capabilities can be exchanged:
 - Delegate: copy capability from one Cap Space to the other
 - Revoke: remove capability, recursively
- Applications use selectors to denote capabilities

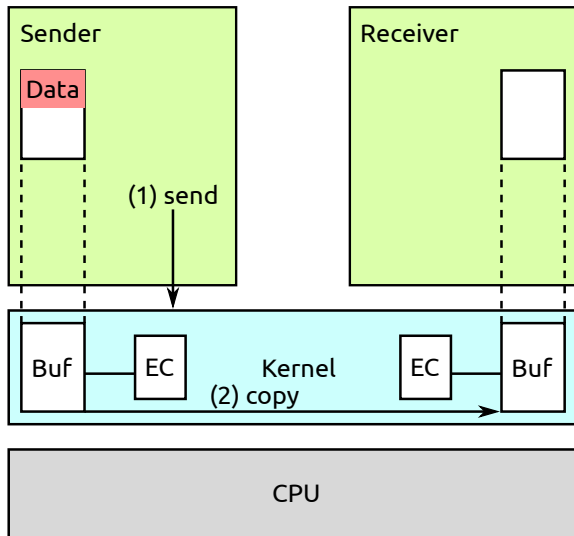
NOVA to Fiasco.OC

Delegate = Map

Capabilities Overview



Interprocess Communication



- **Introduction**

- Threads and address spaces
- Kernel entry and exit
- Interprocess communication
- Capabilities
- Exercise: kernel entry, exit
- Exercise: Linkerscript, Multiboot, ELF
- Exercise: Thread switching
- Case study: M³
- Case study: Fiasco.OC?
- Case study: Escape