User-mode device driver development

WAMOS 2015, 06.08.2015

Annalena Gutheil

Benjamin Weißer

User-mode device driver development

WAMOS 2015, 06.08.2015

Annalena Gutheil

Benjamin Weißer

I. Introduction
II. Device drivers
III. User-mode device drivers in microkernel systems
IV. User-mode device drivers in monolithic systems
V. User-mode device driver frameworks and architectures for Linux
VI. Conclusion

„Why are device drivers evil in kernel-mode?“

- Drivers have a high bug frequency
  - → Difficult development process
  - → Highly hardware specific
- Sharing kernel address space offers huge error potential
- Drivers make up to 70% of Linux kernel
- Drivers are often responsible for system crashes

Device drivers

Device drivers

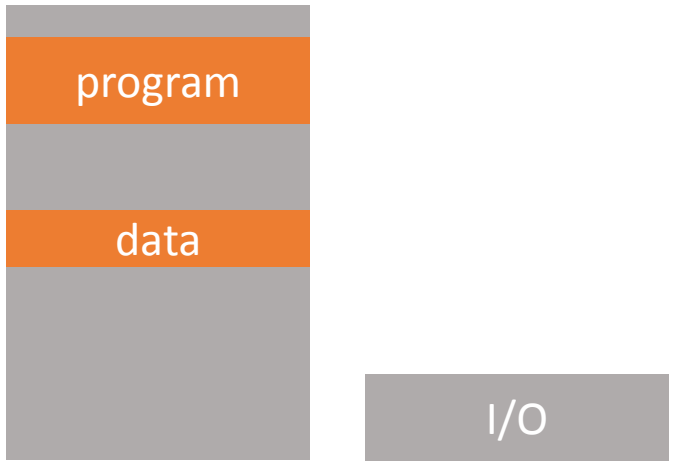
- Interact with and manage a particular type of hardware device
- Main tasks:
  - Accessing device memory
  - Managing data transfer

Accessing device memory

- Memory-mapped I/O
  - Mapping physical device memory into virtual address space
  - I/O device is accessed like it is part of the memory

Accessing device memory

- Port-mapped I/O
  - Intel x86 architectures
  - Special commands for read / write → in / out



Managing data transfer

- Programmed I/O
  - *Polling*
  - CPU repeatedly checks status of device

Managing data transfer

- Interrupt-driven I/O
  - Hardware emits Interrupt Requests (IRQ)
  - Interrupt handler receives interrupts
  - Asynchronous

Managing data transfer

- Direct Memory Access (DMA)
  - DMA controller bypasses CPU, uses the system bus
  - Fast transfer of large data

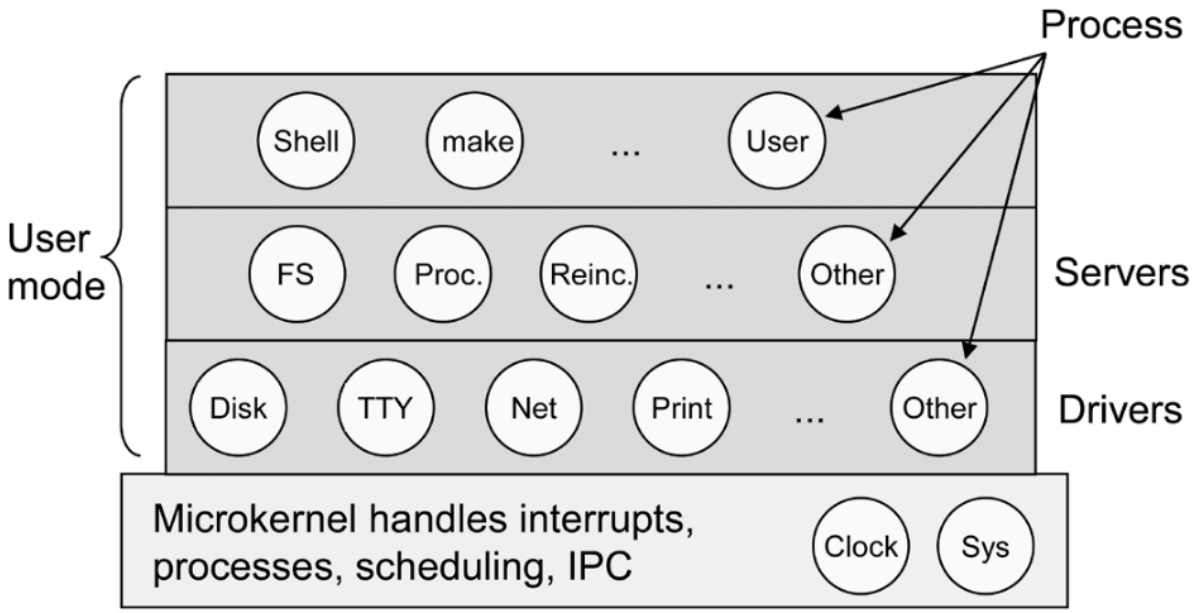
User-mode device drivers in microkernel systems

Microkernels

- User-mode drivers per definition
- Examples: MINIX 3, seL4

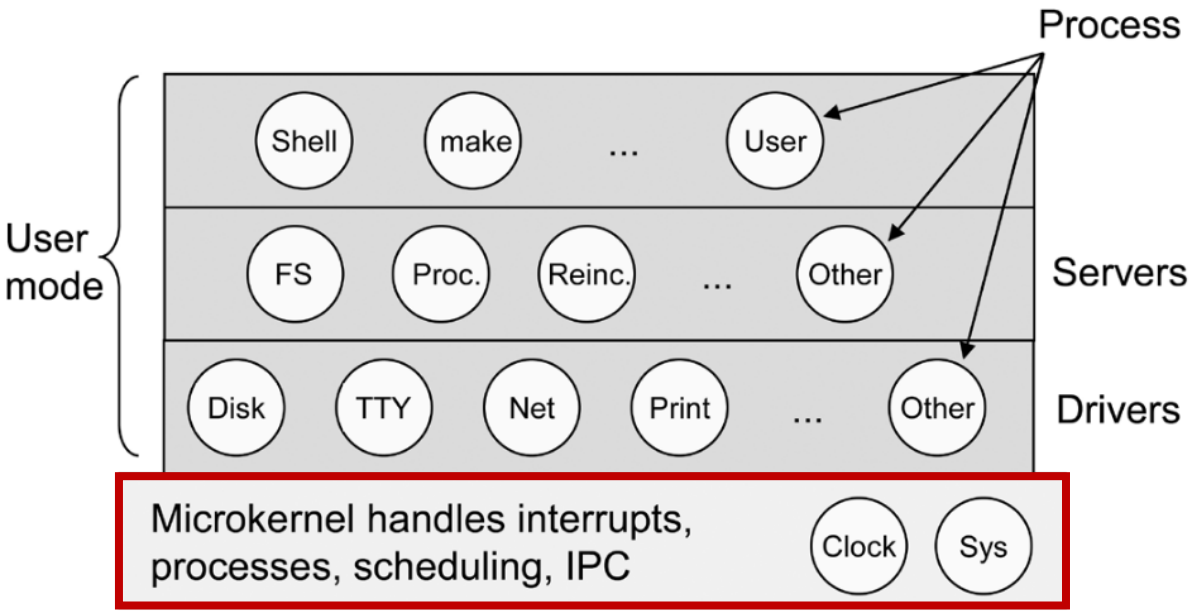
MINIX 3

- POSIX-compliant, UNIX-like, open-source
- Layer architecture



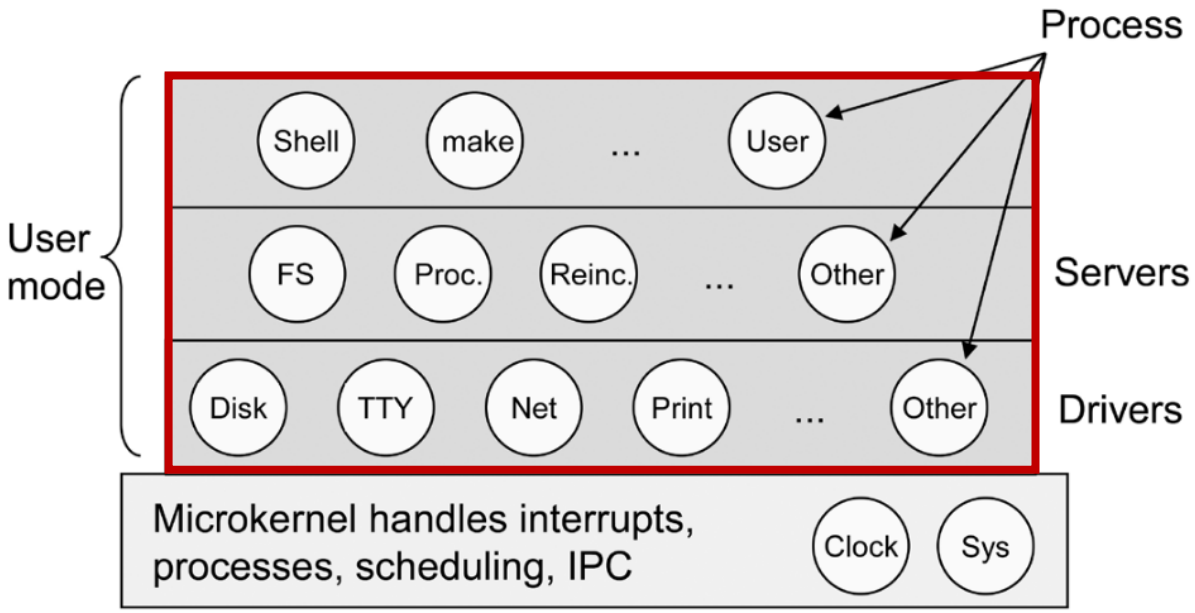
MINIX 3 – Kernel layer

- Kernel calls
- Notifications



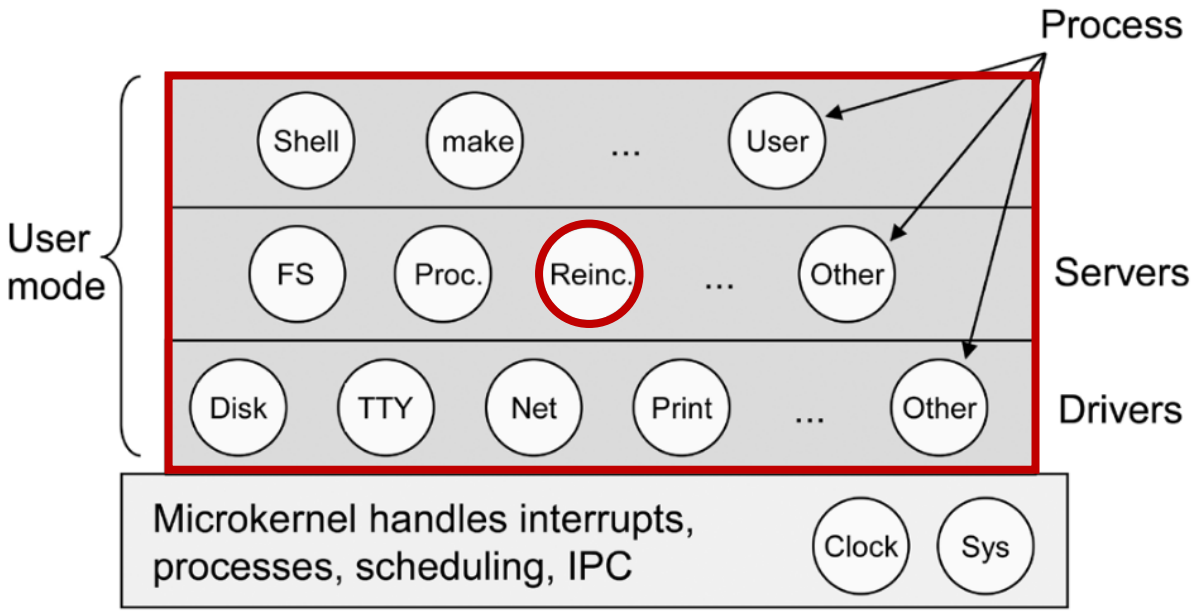
MINIX 3 – User-mode layers

- Message passing
- Memory grants



MINIX 3 – User-mode layers

- Reincarnation Server
  - → Detects crashed processes and restarts them
  - → Feature of device drivers in user space



MINIX 3 – Tasks for driver development

- **I/O access**

  → read/write I/O ports via kernel calls (Intel x86)

- **Interrupts** are passed to the driver by kernel notifications

seL4

- L4 microkernel familiy → high performance microkernels
- seL4: secure → formally verified
- Capability-based

  → Ensure authorization of all operations

seL4 – Tasks for driver development

- **I/O access**
  - → Intel x86: *IO Port* capability authorizes reading/writing of I/O ports
  - → Mapping device memory into virtual address space
- **Interrupt handling**
  - → Implemented as asynchronous IPC messages / notifications

seL4 – IPC

- Uses synchronous and asynchronous endpoints
- *IRQHandler* capability authorizes definition of an endpoint
- Synchronous IPC

  → Data, capabilities
- Asynchronous IPC / notifications

  → Only a single word

Microkernel – DMA

- Security risks: bypassing the MMU, malicious driver can initiate access of inappropriate address space
- Supporting I/O Memory Management Unit (IOMMU)
  - → Constrains the regions of memory for the device
  - → Address mapping

User-mode device drivers in monolithic systems

(kernel-mode device driver = KD)
(user-mode device driver = UD)

Early approaches port the UD concept to Linux

- Suffered from poor performance, e.g. mode switch
- Lack of interrupt handling

Early approaches port the UD concept to Linux

- Suffered from poor performance, e.g. mode switch
- Lack of interrupt handling

Recent UD approaches fix this problem by

- Modifying the original UD concept
- Splitting driver into KD and UD component
- KD component: time- and performance-critical functionality
- UD component: non-critical code

Remember: drivers and therefore UDs have to realize two main tasks

- Accessing a device's memory
- Managing data transfer

Accessing a device's memory

- Using **`mmap()`** and **`/dev/mem`**

Managing data transfer

- Interrupt handling

Interrupt handling with UDs

- Cannot be done within user space
- Several workarounds

Interrupt handling with UDs

- Cannot be done within user space
- Several workarounds

Using a device file

- KD component registers ISR, which listens on special device file
- UD component executes system call **`read()`**
- **`read()`** gets blocked
- Device sends IRQ, call gets unblocked
- UD can handle IRQ

Interrupt handling with UDs

Using `ioctl()`

- Estimates device handle, request code and data argument
- Calling gets blocked
- If device message is available, function gets unblocked and message can be received in user space

Interrupt handling with UDs

Using `ioctl()`

- Estimates device handle, request code and data argument
- Calling gets blocked
- If device message is available, function gets unblocked and message can be received in user space

Using Netlink

- Socket-style kernel interface
- Blocking and non-blocking send/receive functions in user space
- Callback function in kernel space is called when device message appears

Goals of UD concept

- Isolate possible driver bugs

  → Improve stability and reliability

- Decrease kernel footprint

  → Improve maintainability

Two categories of UD approaches

- Split existing KD nearly automatically into the two components
- Write UD component from scratch using provided kernel adapter

UD frameworks and architectures for Linux

Microdriver architecture:

- Splitting pure KD into KD and UD component
- Refactoring tool
  - Splitter: determines critical functions
  - Code generator: generates communication between components

- Evaluation of Microdriver architecture follows after the next approach is introduced
  - Common metrics like CPU utilization and network throughput

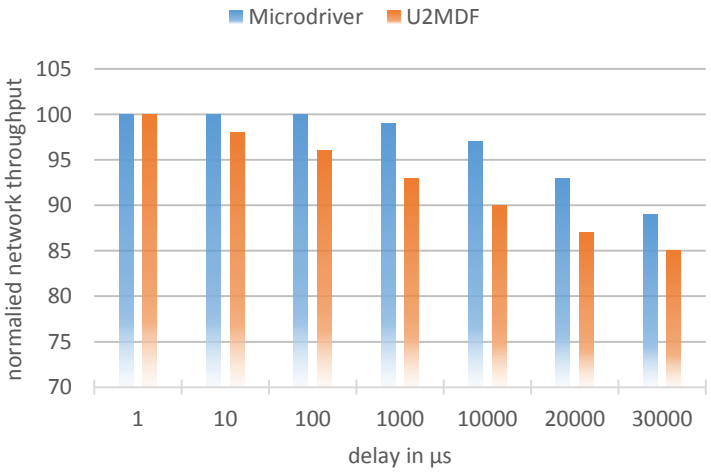
Unified User-Mode Driver Framework (U²MDF):

- Based on the Microdriver architecture
- Aims at high compatibility and simple development
- U²MDF drivers can use
  - I/O ports using **`iopl()`**
  - **`mmap()`** using **`/dev/mem`**
  - Zero-Copy DMA-like technology
- Communication via enhanced Netlink mechanism

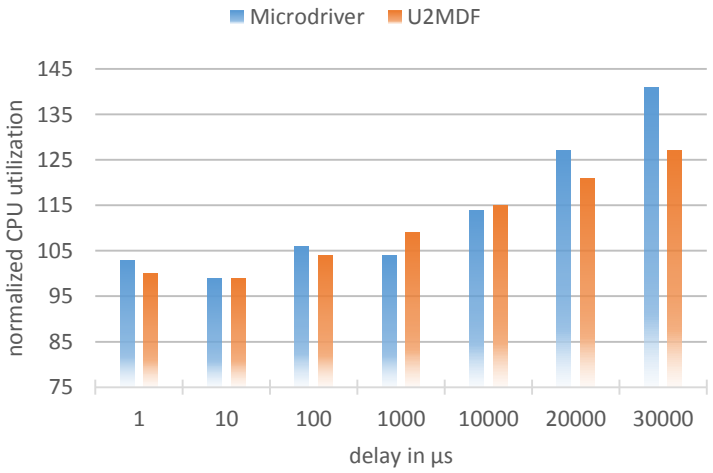
Comparison of Microdriver and U²MDF

- KD and UD component run in kernel space
- Mode switch is simulated with fixed delays

network throughput



CPU utilization



- Performance degradation of U²MDF starts earlier
- CPU utilization of Microdrivers is growing faster with fixed delays higher than 10µs

Userspace I/O (UIO):

- UIO framework in kernel space
- Aims at high performance and simple development
- UIO drivers can use
  - **`mmap()`** using **`/dev/mem`**
- Communication via device file in **`/dev/uioX`**
  - Additional set of status files in **`sysfs`**

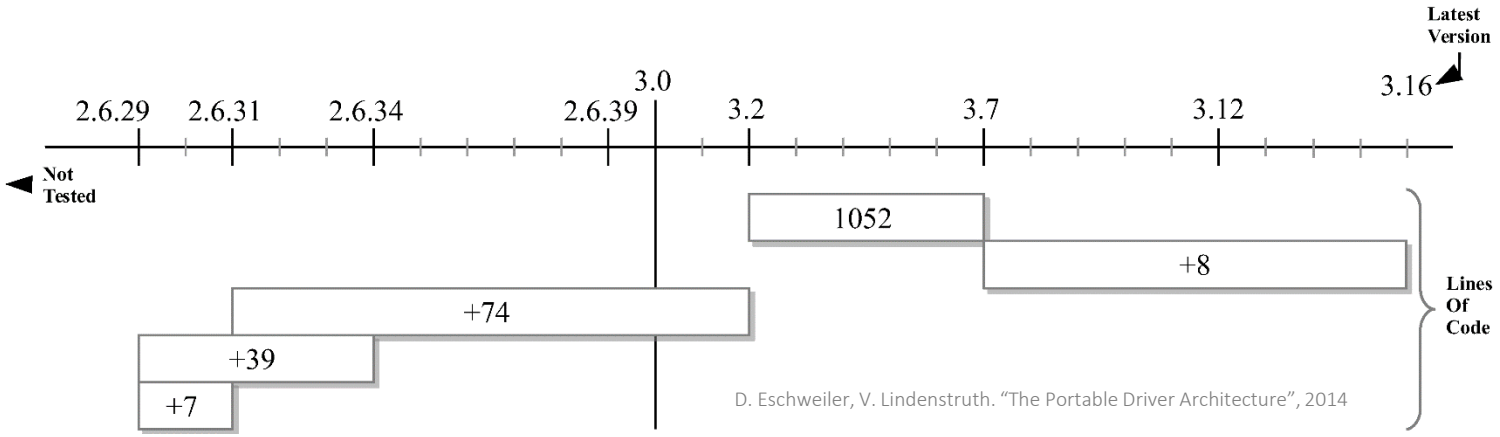
- No evaluation for pure UIO, as the following approach is partially based on UIO and provides a detailed evaluation

Portable Driver Architecture (PDA)

- Partially based on UIO and the PDA predecessor Baracuda
- Aims at high performance, low latency and improved maintainability
- C library provides kernel adapter
  - Interrupt handling
  - Programmed I/O
  - **`mmap()`** using **`/dev/mem`**
  - Zero-Copy DMA-like approach like U²MDF

Evaluation of PDA

- Performance evaluation is done with a high-performance fiber-links network interface card
  - Reached 98% of theoretical throughput of 3.5GiB/s
  - Served the same IRQ rate as pure KD



- PDA library only needs 128 loc to extend compatibility to 27 kernel revisions
  - Greatly improved maintainability

Conclusion

Advantages of UDs

- Developers can use debugging tools
- No restriction to specific programming language
- Decoupled from kernel revision schedule
- Possibility for easy restarting of crashed drivers
- Kernel footprint is decreased
- Improved maintainability
- Removing especially faulty drivers out of kernel improves stability and reliability

1. The small performance loss on average 5% is easily compensated by these advantages.
2. Several UD approaches provide a simple UD development by either splitting pure KDs or providing libraries and frameworks.

Concept of UDs should be taken into consideration when developing drivers