TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Hardware and Device Drivers

Till Miemietz
Maksym Planeta
Björn Döbel

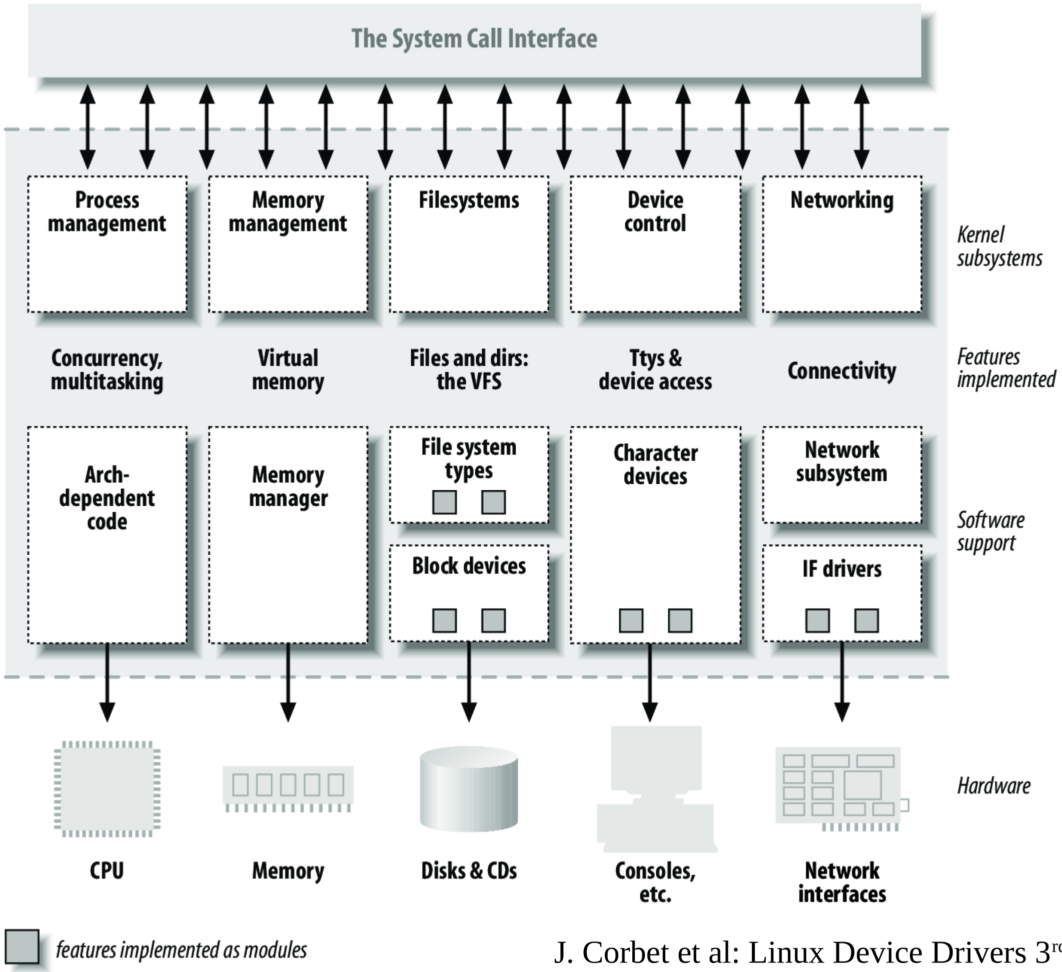
Dresden, November 19, 2024

TECHNISCHE UNIVERSITÄT DRESDEN

Outline

- How do Linux drivers look like?
- What's so different about device drivers?
- How to access hardware?
- L4 services for writing drivers
- Reusing legacy drivers
- Device virtualization

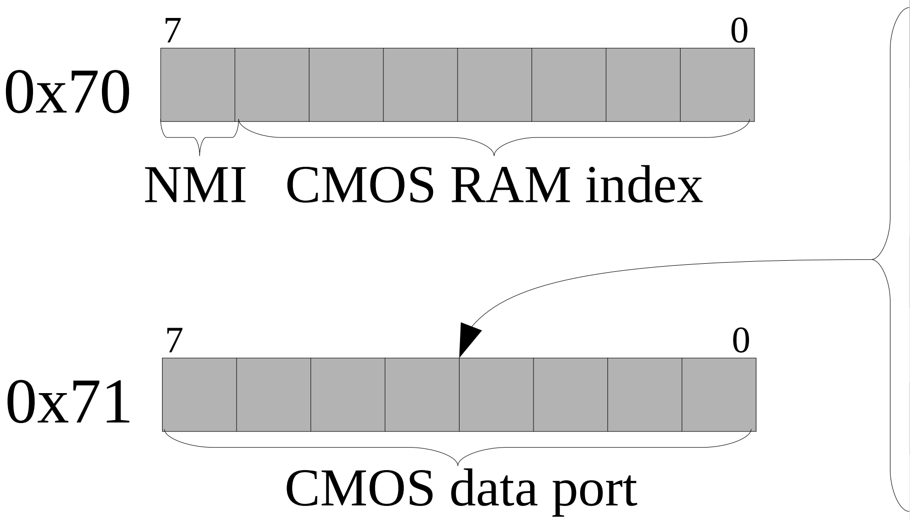
Drivers in Linux



J. Corbet et al: Linux Device Drivers 3rd edition, Chapter 1, page 6

Simple character device driver

- Sketch out how a Linux driver looks like
- A module which allows to read RTC value
- Use IO-ports to access RTC (CMOS map)



RTC registers

00	Current second in BCD
02	Current minute in BCD
04	Current hour in BCD
06	Day of week in BCD
07	Day of month in BCD
08	Month in BCD
09	Year in BCD

Using the device driver

- File in the /dev filesystem

- Read the value

  $ cat /dev/rtctest
  14:05:44 24.11.2020

Simple character device driver

```
/* Global variables definitions. Forward declarations. */

static struct file_operations fops = {
  .open = dev_open,
  .read = dev_read,
  … };

static int __init rtctest_init(void) {...}
static void __exit rtctest_exit(void){...}

static int dev_open(struct inode *inodep, struct file *filep){}
static ssize_t dev_read(struct file *filep, char *buffer,
                        size_t len, loff_t *ppos){...}
module_init(rtctest_init);
module_exit(rtctest_exit);
```

Simple character device driver

```
static int __init rtctest_init(void){
  majorNumber = register_chrdev(0, DEVICE_NAME, &fops); // /dev/rtctest
  if (majorNumber<0)  goto err_major;

  rtctestClass = class_create(THIS_MODULE, CLASS_NAME); // lsmod → rtctest
  if (IS_ERR(rtctestClass)) goto err_class;

  rtctestDevice = device_create(rtctestClass, NULL,
                                MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
  if (IS_ERR(rtctestDevice)) goto err_device;

  rtc_resource = request_region(RTC_PORT_START, RTC_PORT_NUM, "RTC");
  if (!rtc_resource) goto err_region;
  return 0;

  err_region: device_destroy(rtctestClass, MKDEV(majorNumber, 0));
  err_device: class_unregister(rtctestClass); class_destroy(rtctestClass);
  err_class: unregister_chrdev(majorNumber, DEVICE_NAME);
  err_major: return -EFAULT;
}
```

Simple character device driver

```
static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *ppos){
  if (*ppos) goto out;

  get_time(&time);
  ret = snprintf(time_str, MAX_STRLEN, "%d:%d:%d %d.%d.%d",
                 time.hour, time.minute, time.second,
                 time.day_of_month, time.month, time.year);
  if (ret < 0) goto err;

  ret += 1; // Account zero-terminator
  len = len < ret ? len : ret;

  error_count = copy_to_user(buffer+*ppos, time_str+*ppos, len-*ppos);
  if (error_count) goto err;

  *ppos += len;
 /* ... */
}
```

Simple character device driver

```
static void get_time(struct time_struct *time)
{
  int old_NMI;
  local_irq_disable();
  old_NMI = NMI_get();

  time->second       = read_reg(0x00);
  time->minute       = read_reg(0x02);
  time->hour         = read_reg(0x04);
  time->day_of_week  = read_reg(0x06);
  time->day_of_month = read_reg(0x07);
  time->month        = read_reg(0x08);
  time->year         = read_reg(0x09);

  NMI_restore(old_NMI);
  local_irq_enable();
}
```

```
static int from_bcd(int bcd) {
  return ((bcd&0xf0) >> 4)*10+(bcd&0xf);
}

static int read_reg(int reg) {
  outb_p(reg, 0x70);
  int val = inb_p(0x71);
  return from_bcd(val);
}
```

Simple character device driver

```
static void __exit rtctest_exit(void){
  release_region(RTC_PORT_START, RTC_PORT_NUM);
  device_destroy(rtctestClass, MKDEV(majorNumber, 0));     // remove the device
  class_unregister(rtctestClass);                          // unregister the device class
  class_destroy(rtctestClass);                             // remove the device class
  unregister_chrdev(majorNumber, "rtctest");               // unregister the major number
  printk(KERN_INFO "RTCtest: Goodbye from the LKM!\n");
}
```

Monolithic architecture problems

- Which problems do you see?

- What I see
  - Security problems
  - Safety problems
  - Concurrency considerations
  - Requires implicit knowledge
  - Volatile interfaces

Some statistics

- [Swift03]: Drivers cause 85% of Windows XP crashes.
- [Chou01]:
  - Error rate in Linux drivers is 3x (maximum: 10x) higher than for the rest of the kernel
  - Bugs cluster (if you find one bug, you're more likely to find another one pretty close)
  - Life expectancy of a bug in the Linux kernel (~2.4): 1.8 years
- [Rhyzyk09]: Causes for driver bugs
  - 23% programming error
  - 38% mismatch regarding device specification
  - 39% OS-driver-interface misconceptions
- [Xiao19]: “bugs related […] Drivers and ACPI, account for 51.6% of all classified bugs”

Anecdote: Linux e1000 NVRAM bug

- **Aug 8th 2008** Bug report: e1000 PCI-X network cards rendered broken by Linux 2.6.27-rc
  - overwritten NVRAM on card
- **Oct 1st 2008** Intel releases quickfix
  - map NVRAM somewhere else
- **Oct 15th 2008** Reason found:
  - dynamic ftrace framework tries to patch __init code, but .init sections are unmapped after running init code
  - NVRAM got mapped to same location
  - Scary cmpxchg() behavior on I/O memory
- **Nov 2nd 2008** dynamic ftrace reworked for Linux 2.6.28-rc3

Traditional approach does not work

- **Problem**
  Fault in a driver quickly propagates to the whole system
- **Reason**
  Kernel and device drivers are too tightly coupled
- **Solutions**
  - Verification (e. g. Singularity [Hunt07])
  - Hardware assisted isolation (e.g. Intel's MPK)
  - Specialized fault tolerance techniques (e. g. Otherworld [Dep10])
  - Safe languages (Rust)

Idea: User-level Drivers

- **Isolate components**
  - device drivers (disk, network, graphic, USB cruise missiles, ...)
  - stacks (TCP/IP, file systems, …)
- **Separate address spaces each**
  - More robust components
- **Problems**
  - Overhead
    - HW multiplexing
    - Context switches
  - Need to handle I/O privileges

TECHNISCHE UNIVERSITÄT DRESDEN

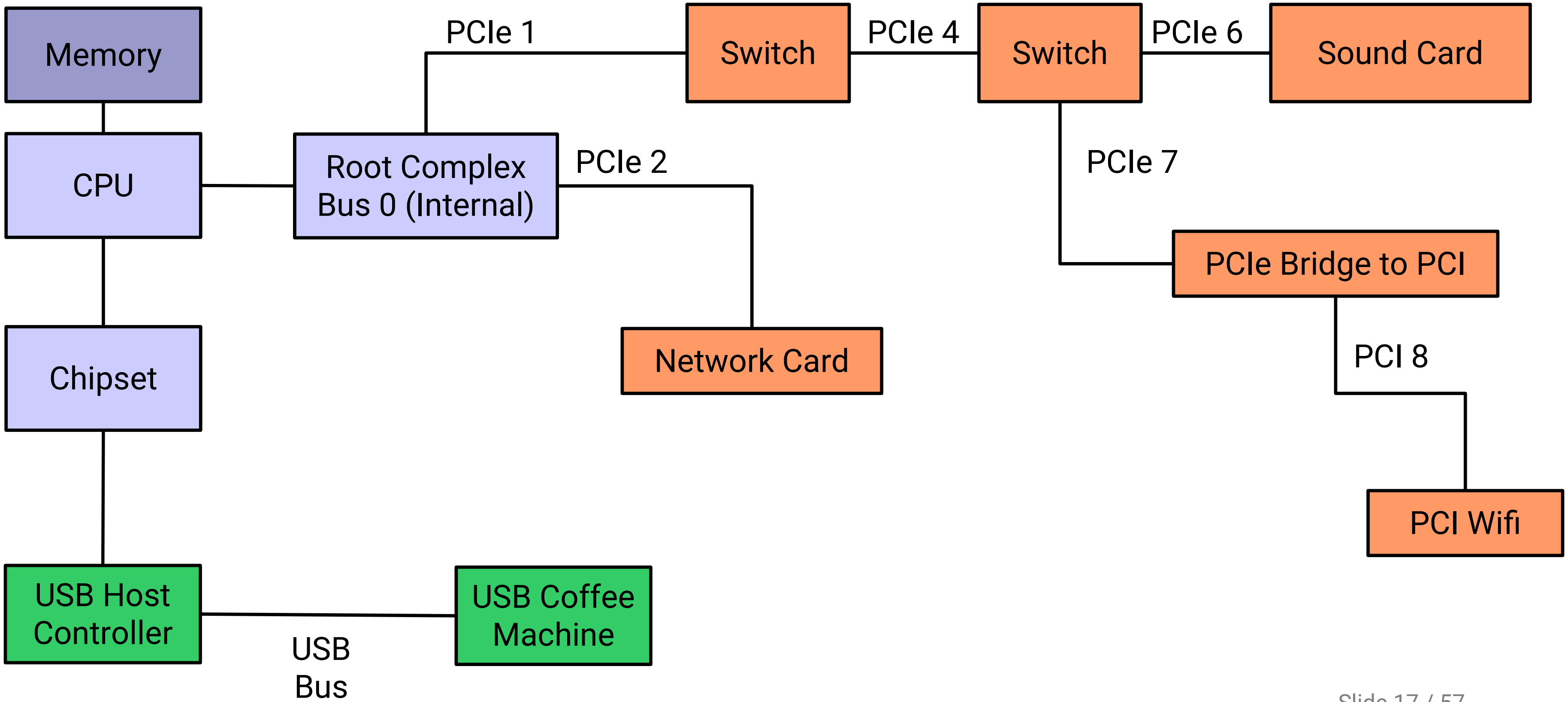
A closer look

- Organization of device hierarchy
  - CPU
  - Chipset
  - Buses
- How devices interact with OS
  - Ports
  - IO memory
  - Interrupts

TECHNISCHE UNIVERSITÄT DRESDEN

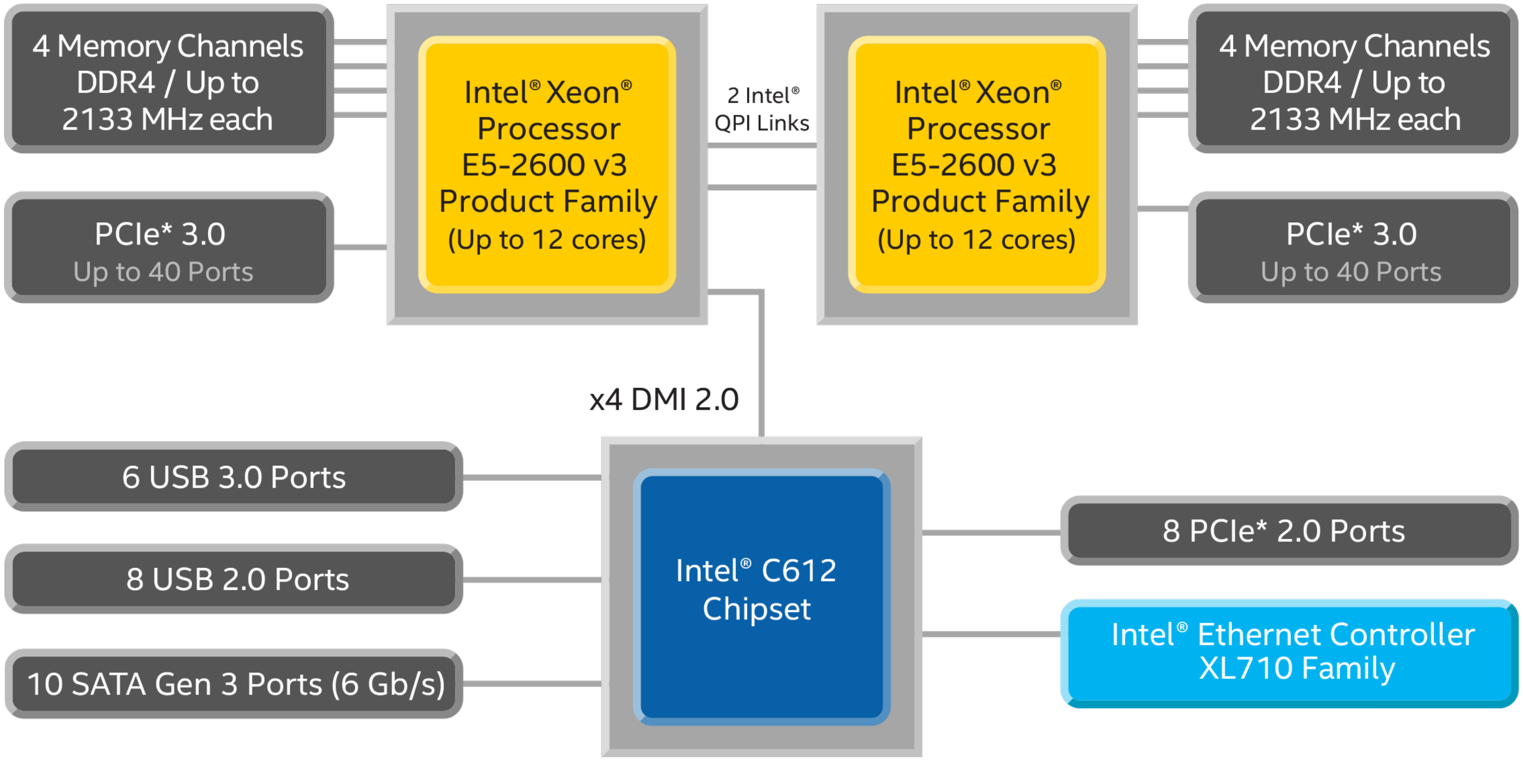
System Layout

- Devices connected by buses (USB, PCI, PCIe)
- Host chipset (DMA logic, IRQ controller) connects buses and CPU



Source: pcisig.com

Real World Example



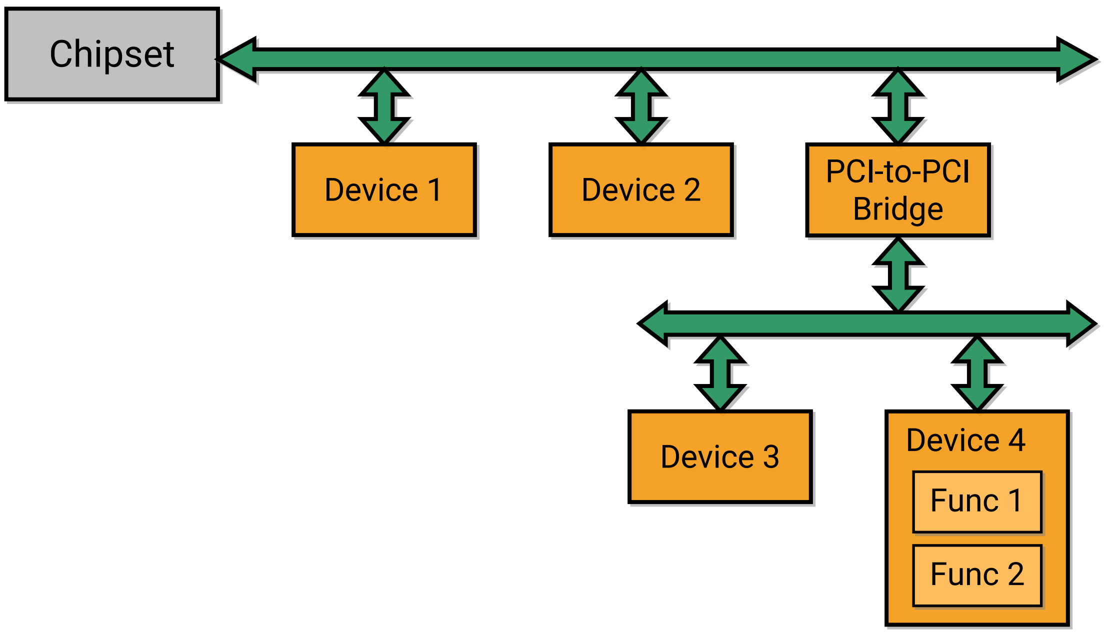
Intel c612 Chipset
(source: intel.com)

Buses and Devices

- A long long time ago:
  - device architecture hard-coded
- Problem: more and more devices
  - need means of dynamic device discovery
- Probing
  - try out every driver to see if it works
- Plug&Play:
  - first try of dynamic system description
  - device manufacturers provide unique IDs
- PCI: dedicated config space
- ACPI: system description without relying on underlying bus/chipset

Buses: PCI

- Peripheral Component Interconnect
- Hierarchy of buses, devices and functions
- Configuration via I/O ports
  - Address + data register (0xcf8-0xcff)

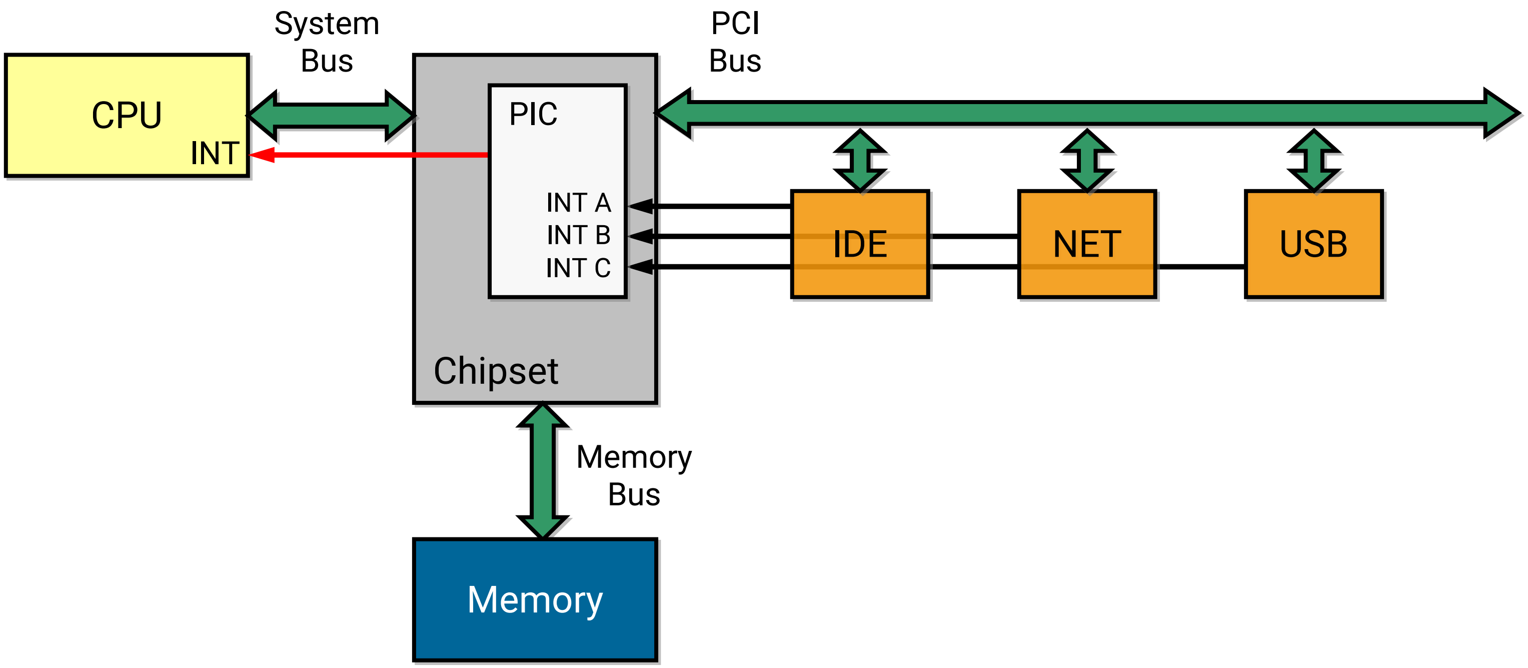


Buses: PCI (2)

- PCI configuration space
- 64 byte header
  - Busmaster DMA
  - Interrupt line
  - I/O port regions
  - I/O memory regions
  - + 192 byte additional space
- must be provided by every device function
- must be managed to isolate device drivers

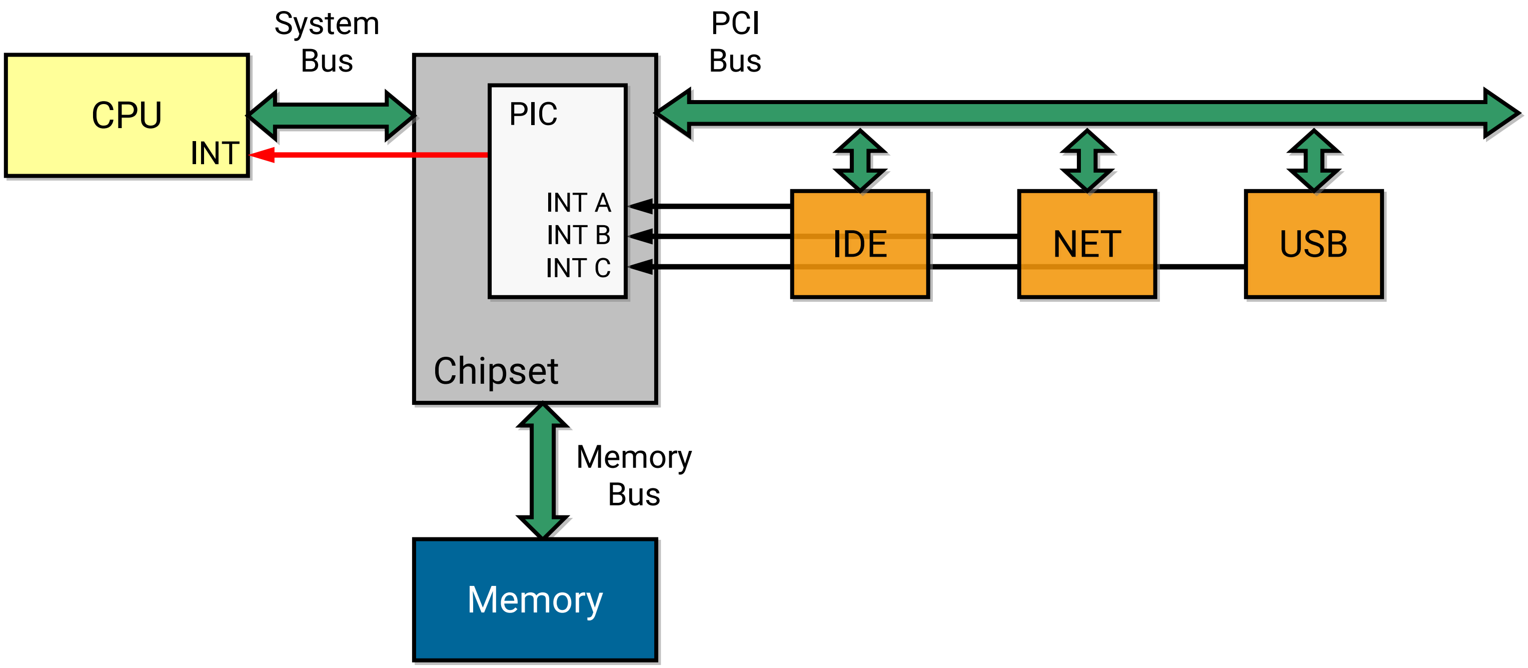
Interrupts

- Signal device state change
- Programmable Interrupt Controller (PIC, APIC)
  - map HW IRQs to CPU's IRQ lines
  - prioritize interrupts



Interrupts (2)

- Handling interrupts involves
  - examine / manipulate device
  - program PIC
    - acknowledge/mask/unmask interrupts



L4: Interrupt handling

- IRQ kernel object
  - Represents arbitrary async notification
  - Kernel maps hardware IRQs to IRQ objects
- Exactly one waiter per object
  - call `l4_irq_attach()` before
  - wait using `l4_irq_receive()`
- Multiple IRQs per waiter
  - attach to multiple objects
  - use `l4_ipc_wait()`
- IRQ sharing
  - Many IRQ objects may be `chain()`ed to a master IRQ object

Disabling interrupts

- CLI – only with IO Privilege Level (IOPL) 3
- Should not be allowed for every user-level driver
  - untrusted drivers
  - security risk
- Observation: drivers often don't need to disable IRQs globally, but only access to their own IRQ
  - Just don't receive from your IRQ

Linux: Interrupt handling

- Catching interrupts in a driver
  - Setup a handler with request_irq() in open()
  - Release interrupt line with free_irq in close()
- Disabling interrupts is also bad in kernel
  - Handler should be quick
  - If it is not quick, split the handler
- Top and bottom halves
  - Top half catches invoked immediately, and schedules “real” handler
  - Bottom half is executed by the kernel in preemptable context, but can be slow

I/O ports

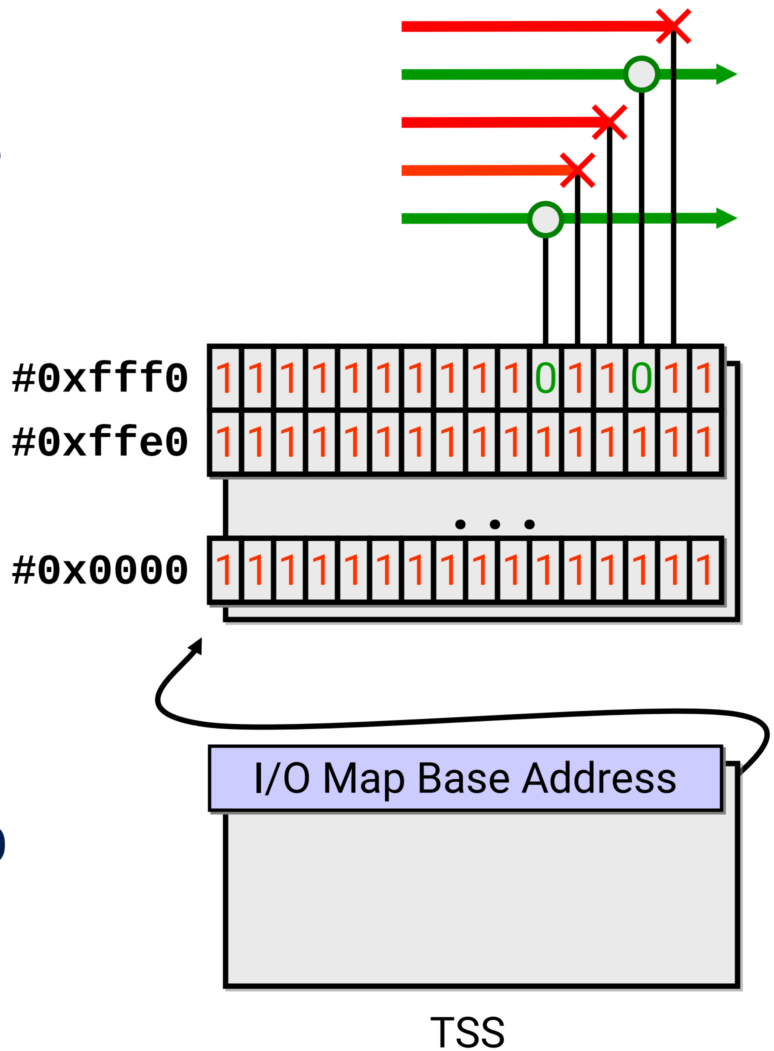
- x86-specific feature
- I/O ports define own I/O address space
  - Each device uses its own area within this address space
- Special instruction to access I/O ports
  - in / out: I/O read / write
  - Example: read byte from serial port

```
mov $0x3f8, %edx
in  (%dx), %al
```

- Need to restrict I/O port access
  - Allow device drivers access to I/O ports used by its device only

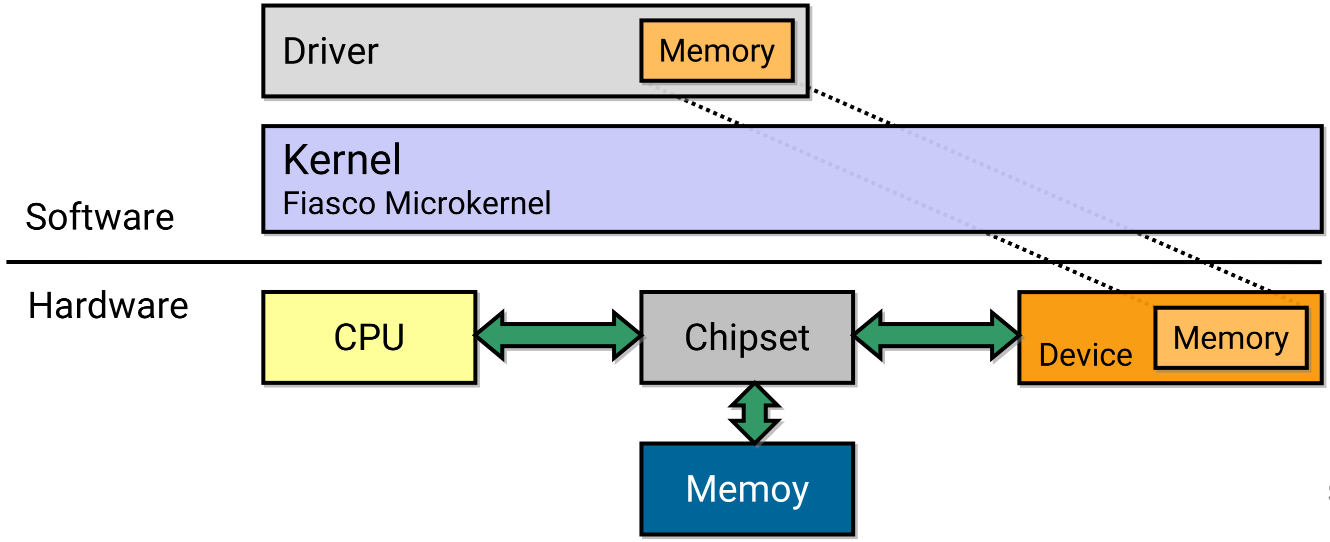
I/O Bitmap

- Per task IO privilege level (IOPL)
- If IOPL > current PL, all accesses are allowed
  (kernel mode)
- Else: I/O bitmap is checked
- 1 bit per I/O port
  - 65536 ports -> 8kB
- Controls port access
  (0 == ok, 1 == GPF)
- L4: per-task I/O bitmap
  - Switched during task switch
  - Allows per-task grant/deny of I/O port access



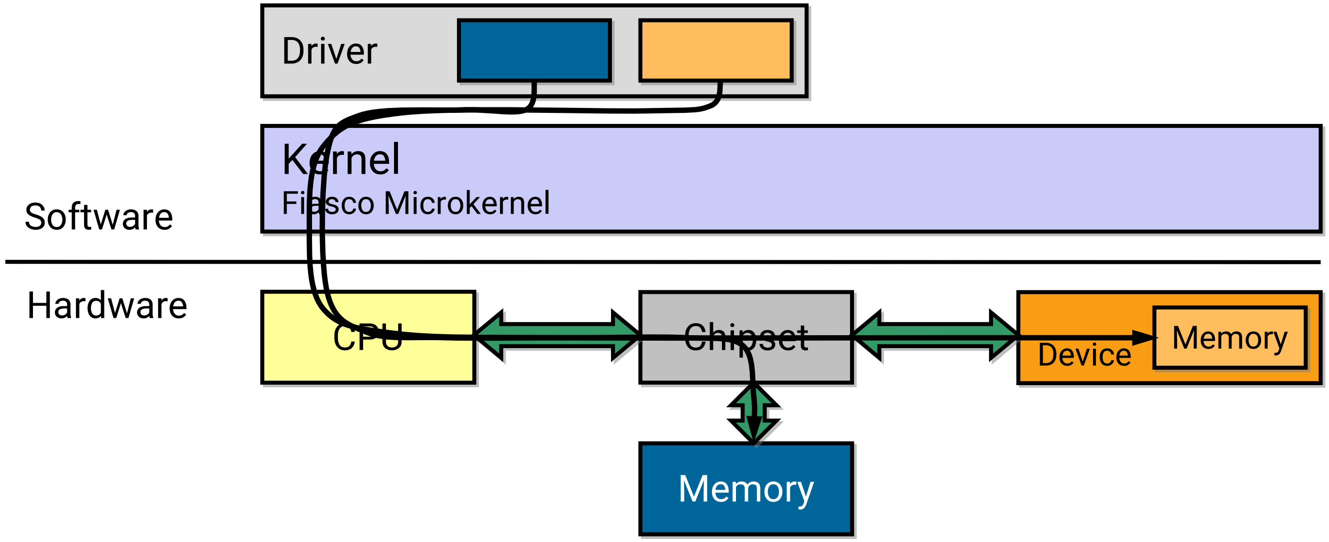
I/O Memory

- Devices often contain on-chip memory (NICs, graphics cards, ...)
- Drivers can map this memory into their address space just like normal RAM
  - no need for special instructions
  - increased flexibility by using underlying virtual memory management



I/O memory (2)

- Device memory looks just like phys. memory
- Chipset needs to
  - map I/O memory to exclusive address ranges
  - distinguish physical and I/O memory access

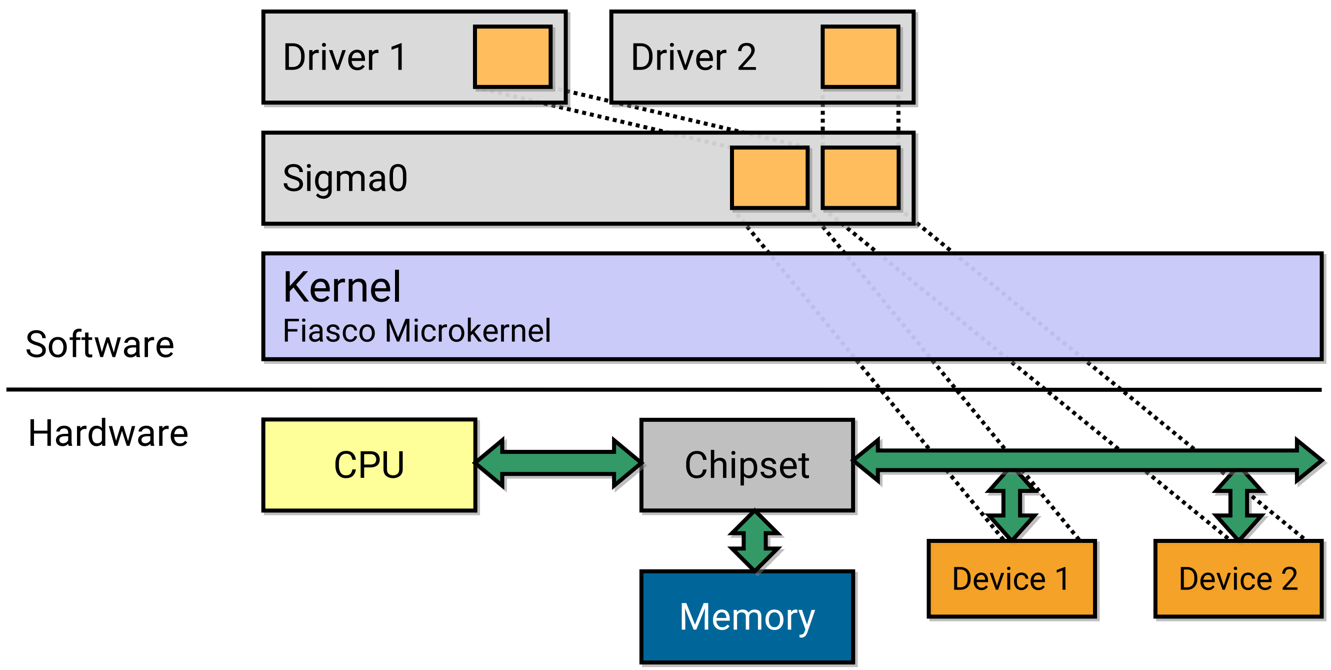


Capabilities in Driver Context

- A driver can grant, share or receive a capability for every object
- Flexpage is a descriptor for capabilities in L4
- Flexpage types:
  - Memory
  - IO ports
  - Objects

I/O memory in L4

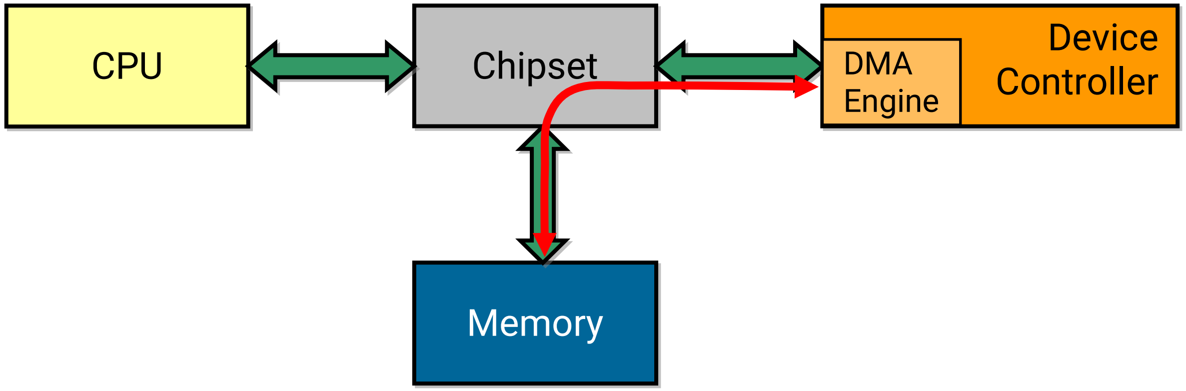
- Like all memory, I/O memory is owned by sigma0
- Sigma0 implements protocol to request I/O memory pages
- Abstraction: Dataspaces containing I/O memory



Direct Memory Access (DMA)

- Bypass CPU by directly transferring data from device to RAM
  - improved bandwidth
  - relieved CPU

- DMA controller either programmed by driver or by device's DMA engine (Busmaster DMA)



TECHNISCHE UNIVERSITÄT DRESDEN

Problems with DMA

- DMA uses physical addresses.
  - I/O memory regions need to be physically contiguous → supported by L4Re dataspace manager
  - Buffers must not be paged out during DMA → L4Re DS manager allows “pinning” of pages

- DMA with phys. addresses bypasses VM management
  - Drivers can overwrite any phys. Address

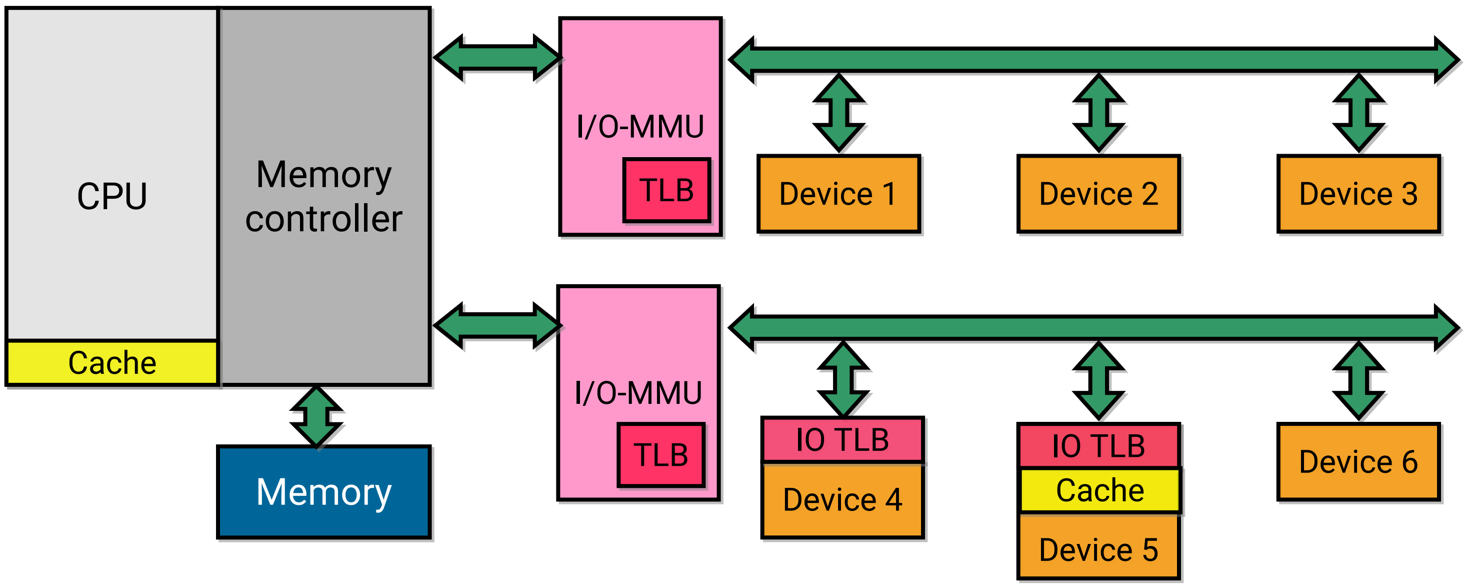
- DMA is both a safety and a security risk.

- Which mechanism do you know to protect untrusted software from accessing physical memory?

Idea: I/O MMU

- Like traditional MMU maps virtual to physical addresses
  - implemented in PCI bridge
  - manages a page table
  - I/O-TLB
- Drivers access buffers through virtual addresses
  - I/O MMU translates accesses from virtual to IO-virtual addresses (IOVA)
  - restrict access to phys. memory by only mapping certain IOVAs into driver's address space
- Interrupt remapping and virtualization

I/O MMU architecture

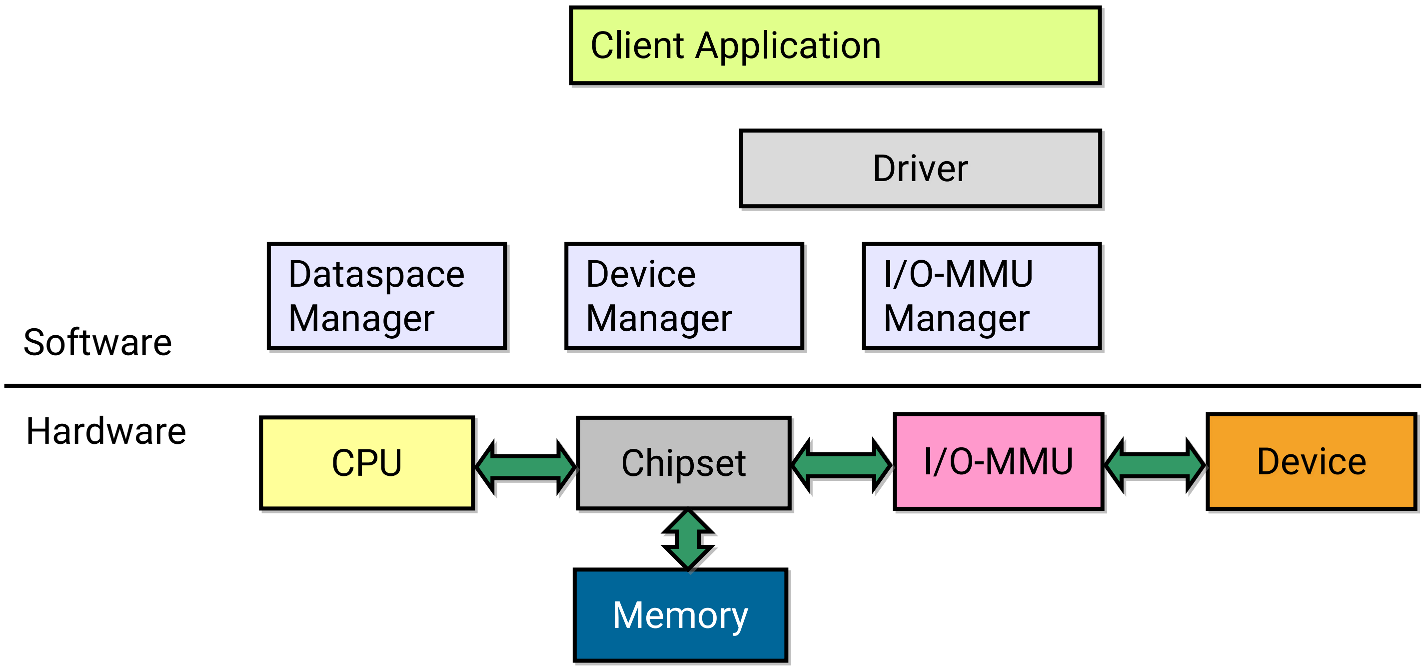


Source: amd.com

- Do you see a security problem?
  - Device TLB and caches bypass IO-MMU

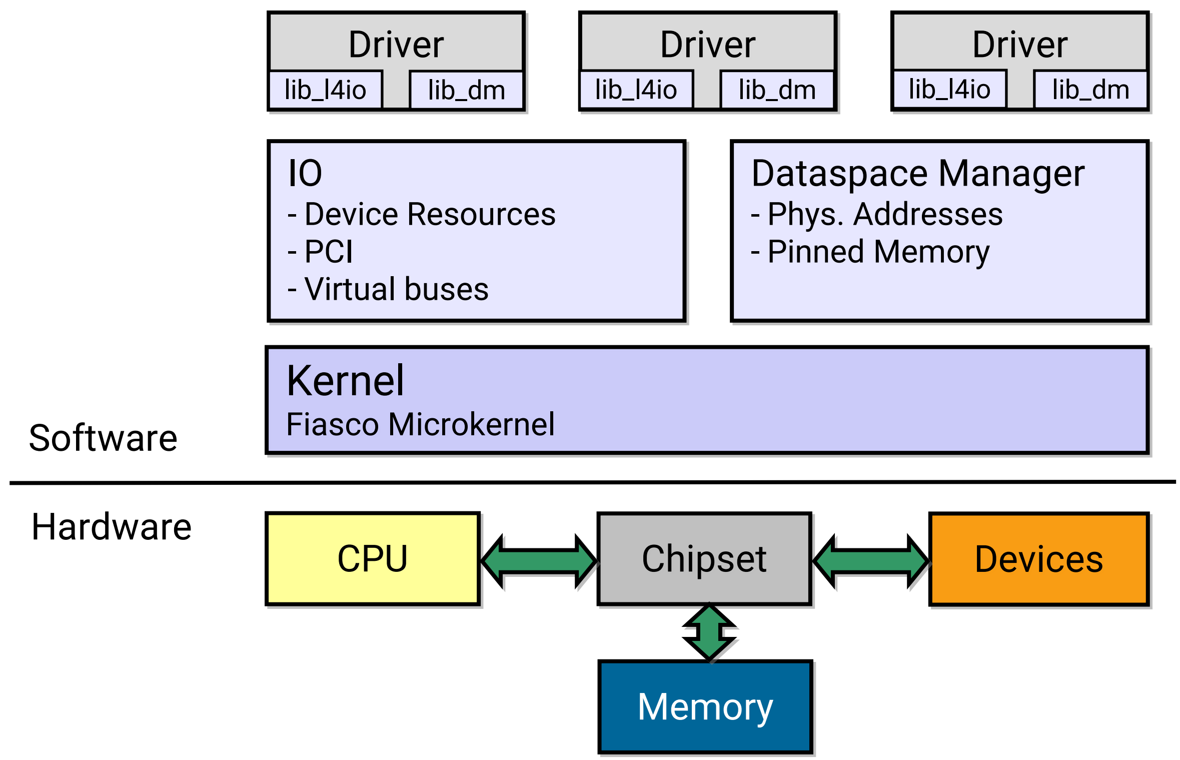
I/O MMU Architecture

- I/O MMU managed by yet another resource manager
- Before accessing I/O memory, drivers use manager to establish a virt → phys mapping



Summary: Driver support in L4

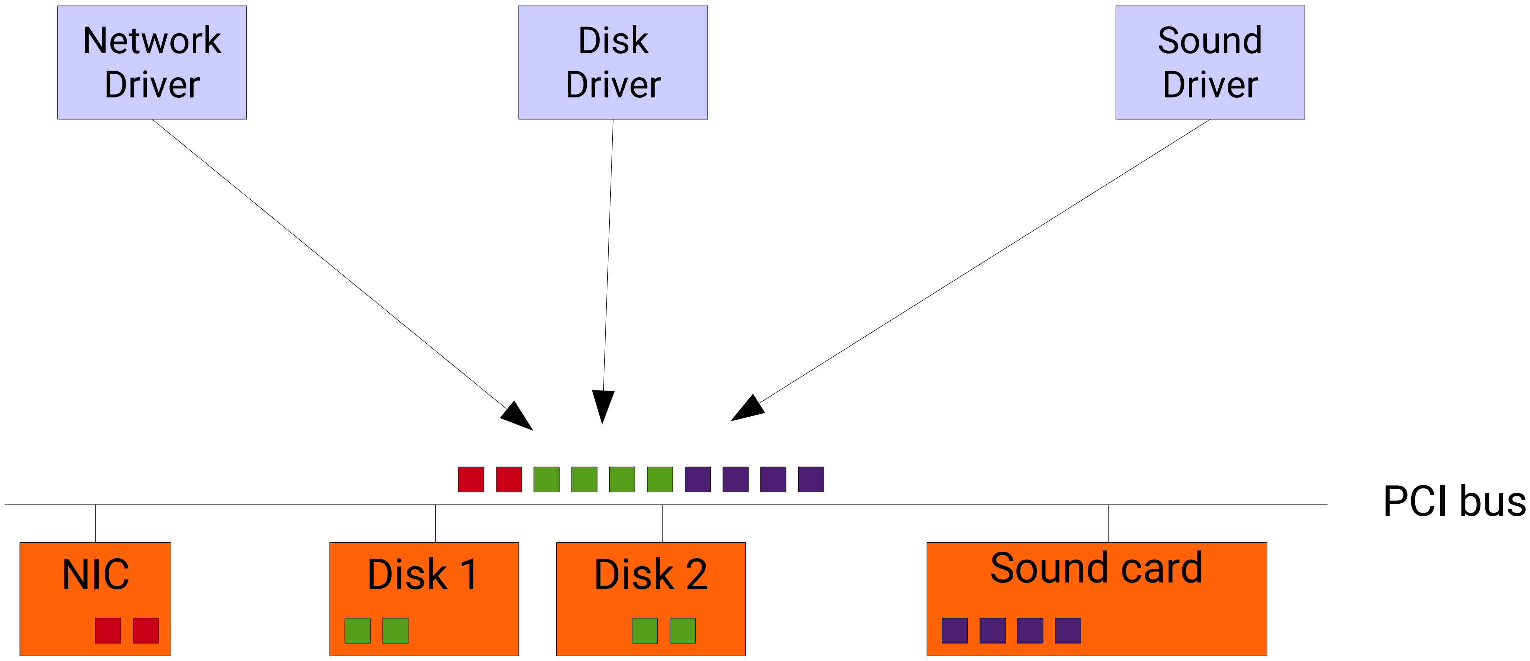
- Interrupts -> Kernel object + IPC
- I/O ports and memory -> flexpage mappings
- User-level resource manager -> IO



TECHNISCHE UNIVERSITÄT DRESDEN

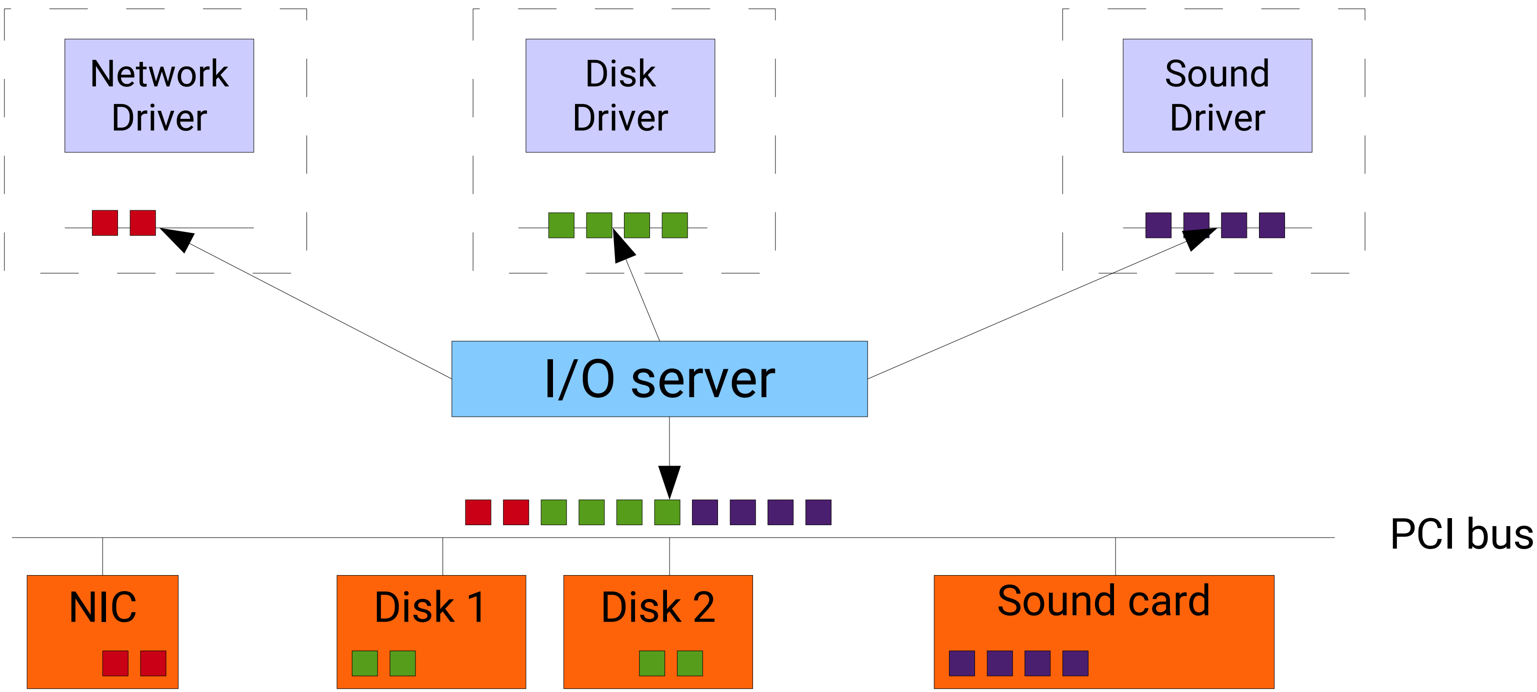
Untrusted Device Drivers

- How to enforce device access policies on untrusted drivers?



Untrusted Device Drivers

- How to enforce device access policies on untrusted drivers?
- I/O manager needs to manage device resources
  - Virtual buses



Break

- Device drivers are hard.
- Hardware is complex.
- Virtual buses for isolating device resources

- Next: Implementing device drivers on L4 without doing too much work

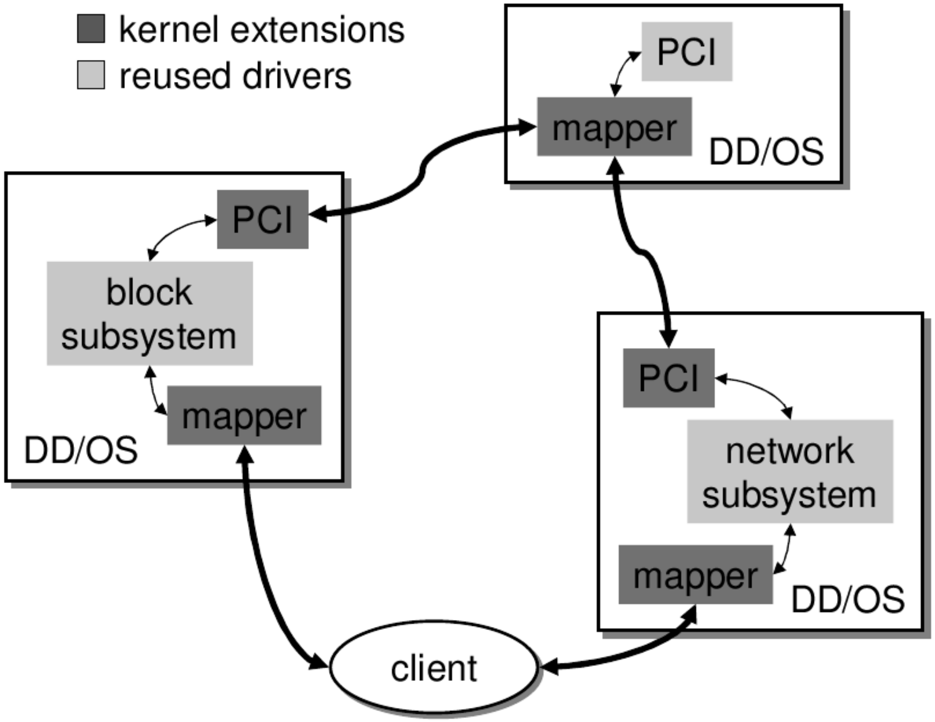
Implementing Device Drivers

- Just like in any other OS:
  - Specify a server interface
  - Implement interface, use the access methods provided by the runtime environment
- Highly optimized code possible
- Hard to maintain
- Implementation time-consuming
- Unavailable specifications
- Why reimplement drivers if they are already available on other systems?
  - Linux, BSD – Open Source
  - Windows – Binary drivers

TECHNISCHE UNIVERSITÄT DRESDEN

Reusing legacy device drivers

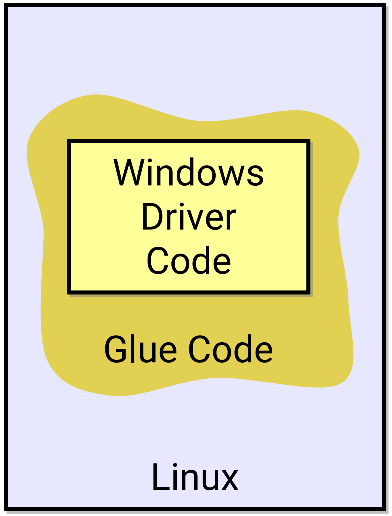
- Exploit virtualization: Device Driver OS



Source: LeVasseur et. al.: *"Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines"*, OSDI 2004

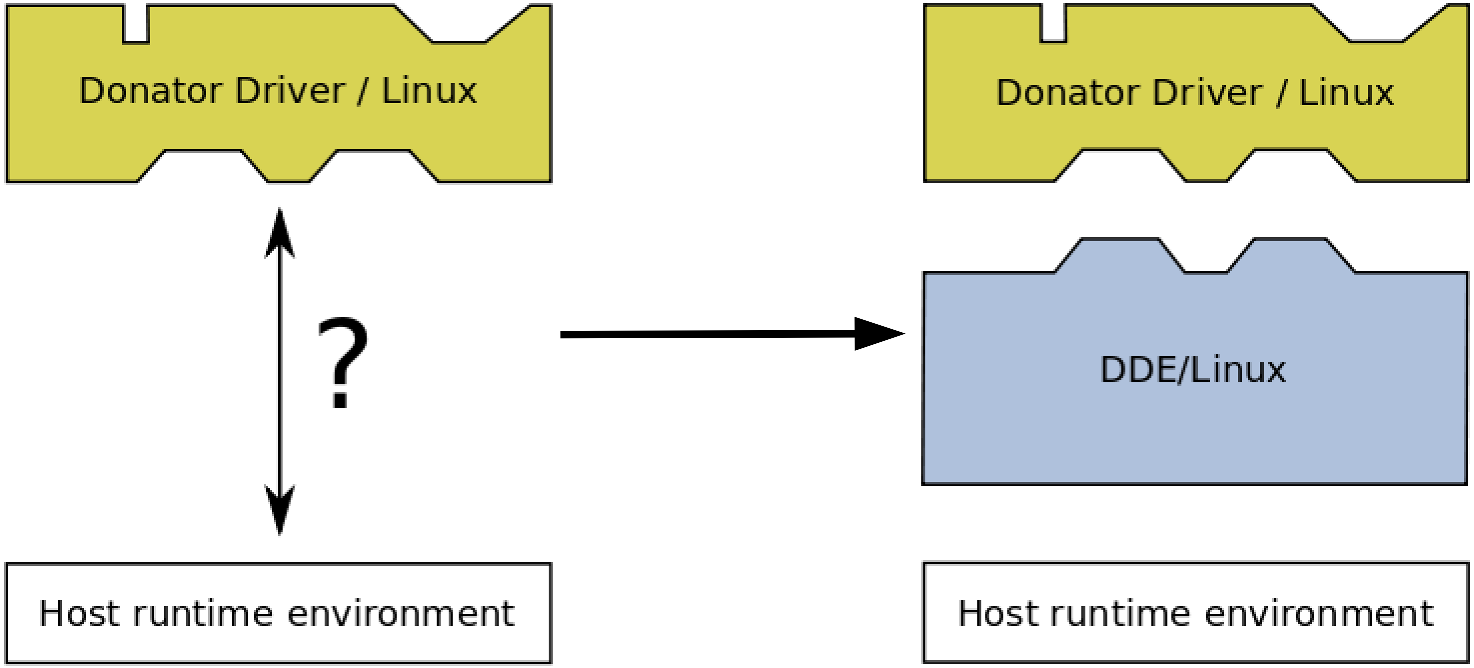
Reusing Legacy Device Drivers

- NDIS-Wrapper: Linux glue library to run Windows WiFi drivers on Linux
- Idea is simple: provide a library mapping Windows API to Linux
- Implementation is a problem.



Reusing Legacy Device Drivers (2)

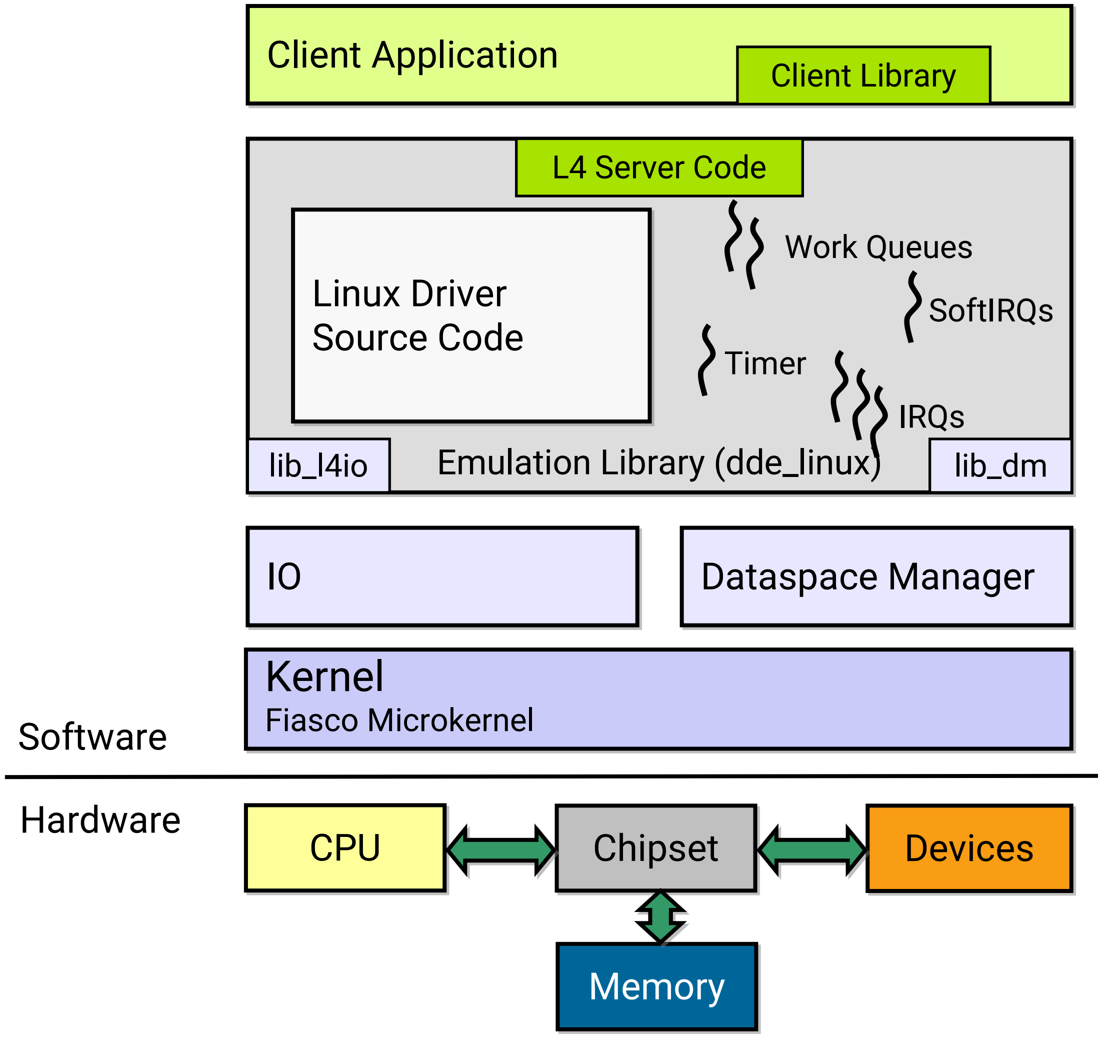
- Generalize the idea: provide a Linux environment to run drivers on L4
  → Device Driver Environment (DDE)



Emulating Linux: DDE/Linux

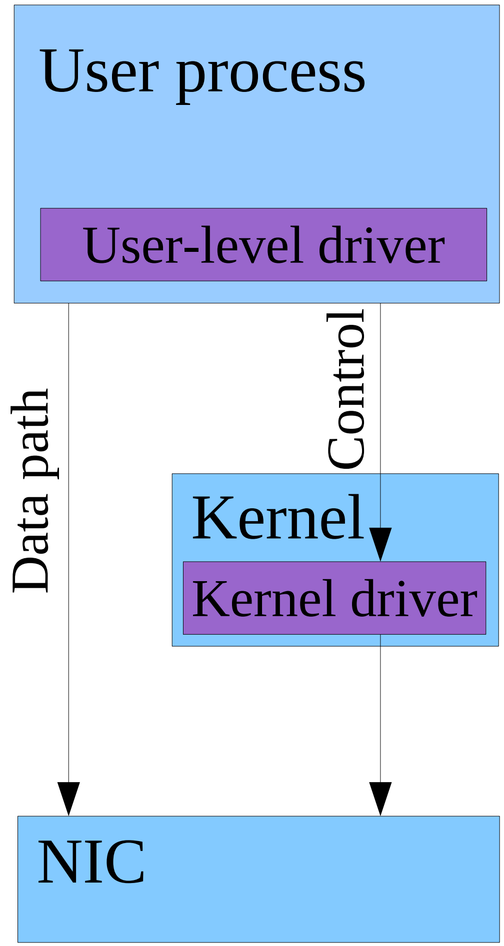
- Multiple L4 threads provide a Linux environment
  - Workqueues
  - SoftIRQs
  - Timers
  - Jiffies
- Emulate SMP-like system (each L4 thread assumed to be one processor)
- Wrap Linux functionality
  - kmalloc() → L4 Slab allocator library
  - Linux spinlock → pthread mutex
- Handle in-kernel accesses (e.g., PCI config space)

DDE Structure

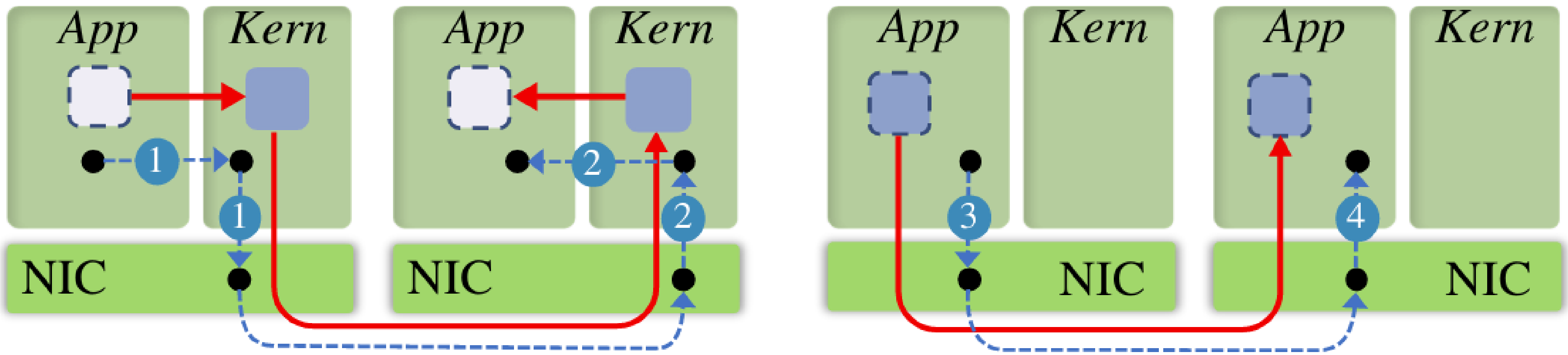


RDMA Networks

- Remote Direct Memory Access
- Separate control and data path
- Control path:
  - Connection setup
  - Goes through kernel
- Data path:
  - Data exchange
  - Directly exchange with NIC (DMA)
- Network Interface Controller (NIC)
  - Specialized interface
  - No need for SR-IOV



RDMA-network vs TCP/IP-network



Traditional (left) and RDMA (right) network stacks.

Features of RDMA-networks

- Kernel-bypass
  - No user-kernel boundary crossing
- Zero-copy
  - No message copy through the kernel
- Offloading
  - RDMA API is message-level
  - NIC splits messages into packets

Arrakis: the dataplane OS

- Kernel code can add significant overhead
- Put the device driver in the application
  - LibOS
- Hardware virtualization for isolation
  - SR-IOV
  - VNIC
  - IOMMU

Using device in Linux

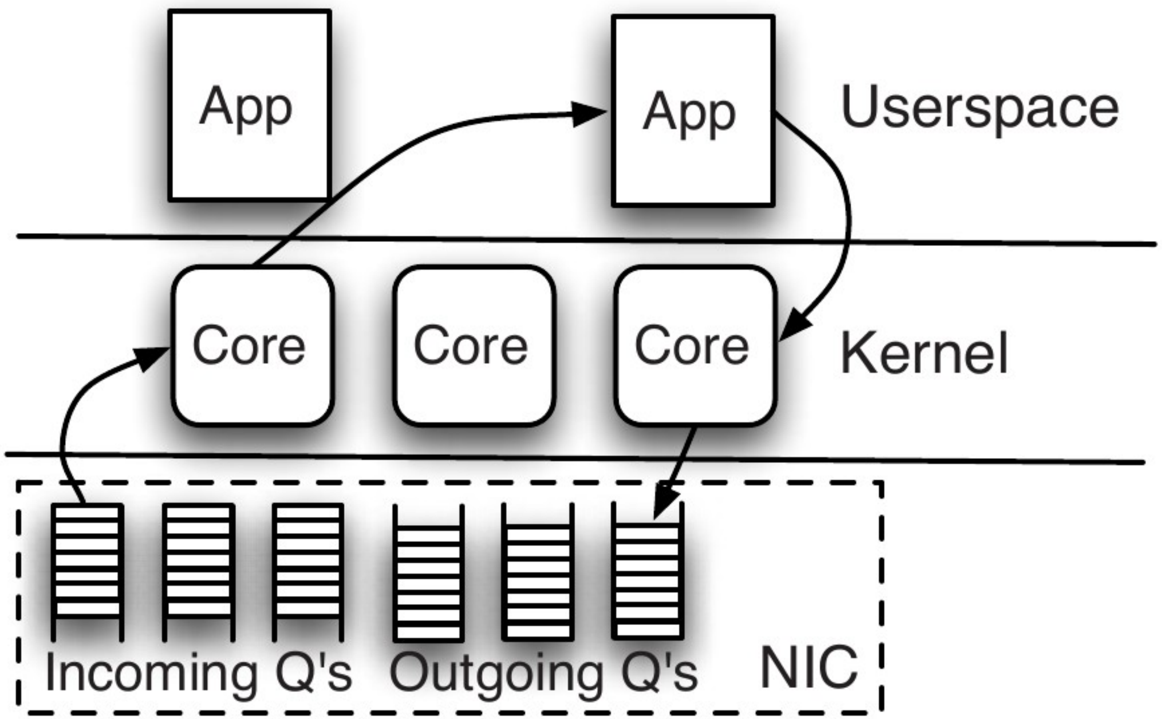


Figure 1: Linux networking architecture and workflow.

Source: The Morning Paper: Arrakis - the operating system is the control plane, S. Peter, et al. OSDI 2014

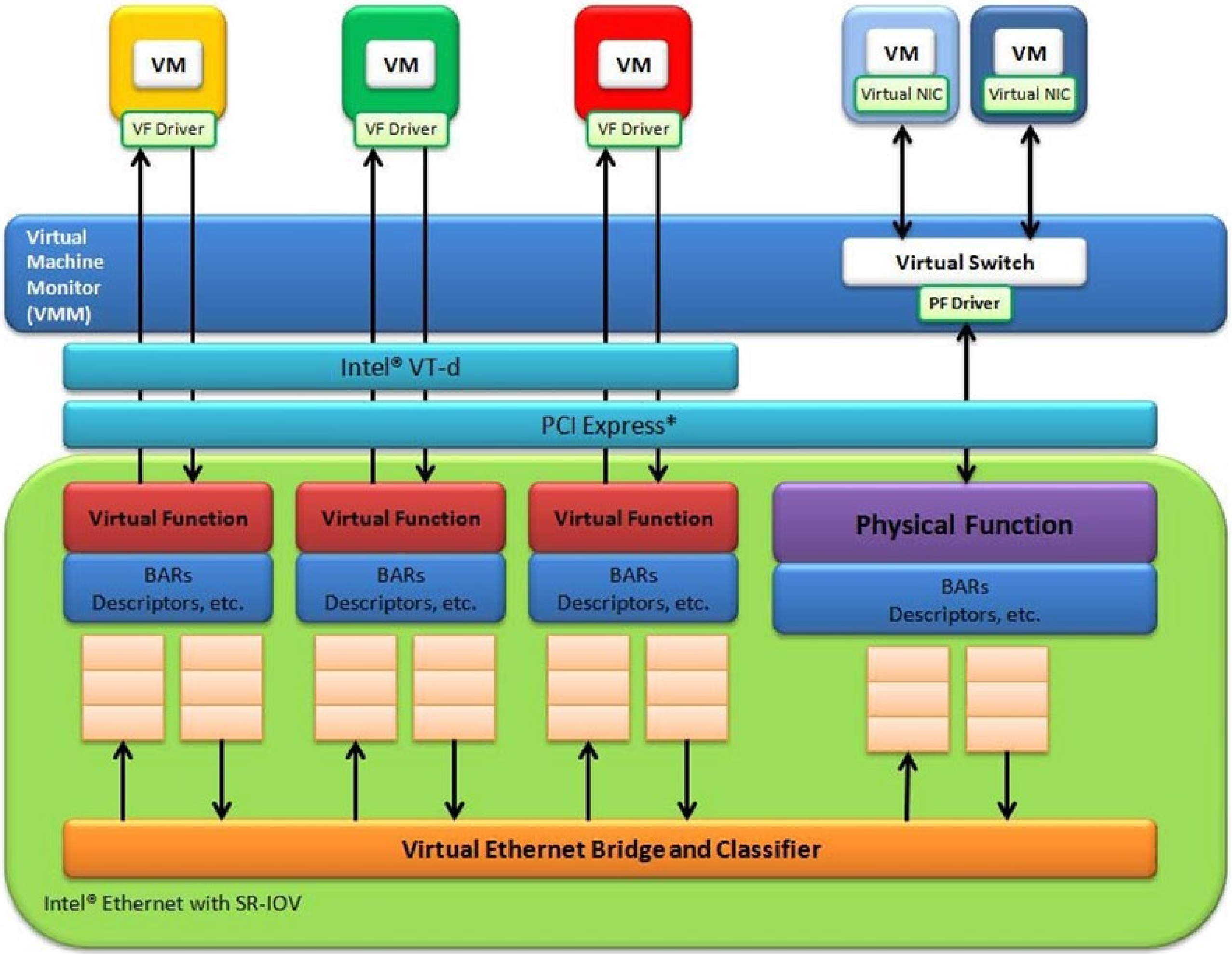
Linux overhead

		Linux				Arrakis			
		Receiver running		CPU idle		Arrakis/P		Arrakis/N	
Network stack	in	1.26	(37.6%)	1.24	(20.0%)	0.32	(22.3%)	0.21	(55.3%)
	out	1.05	(31.3%)	1.42	(22.9%)	0.27	(18.7%)	0.17	(44.7%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)	-		-	
Copy	in	0.24	(7.1%)	0.25	(4.0%)	0.27	(18.7%)	-	
	out	0.44	(13.2%)	0.55	(8.9%)	0.58	(40.3%)	-	
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)	-		-	
	syscall	0.10	(2.9%)	0.13	(2.1%)	-		-	
Total		3.36	($\sigma = 0.66$)	6.19	($\sigma = 0.82$)	1.44	($\sigma < 0.01$)	0.38	($\sigma < 0.01$)

Table 1: Sources of packet processing overhead in Linux and Arrakis. All times are averages over 1,000 samples, given in μs (and standard deviation for totals). Arrakis/P uses the POSIX interface, Arrakis/N uses the native Arrakis interface.

Source: The Morning Paper: Arrakis - the operating system is the control plane, S. Peter, et al. OSDI 2014

SR-IOV Architecture



Source: https://doc.dpdk.org

Using device in Arrakis

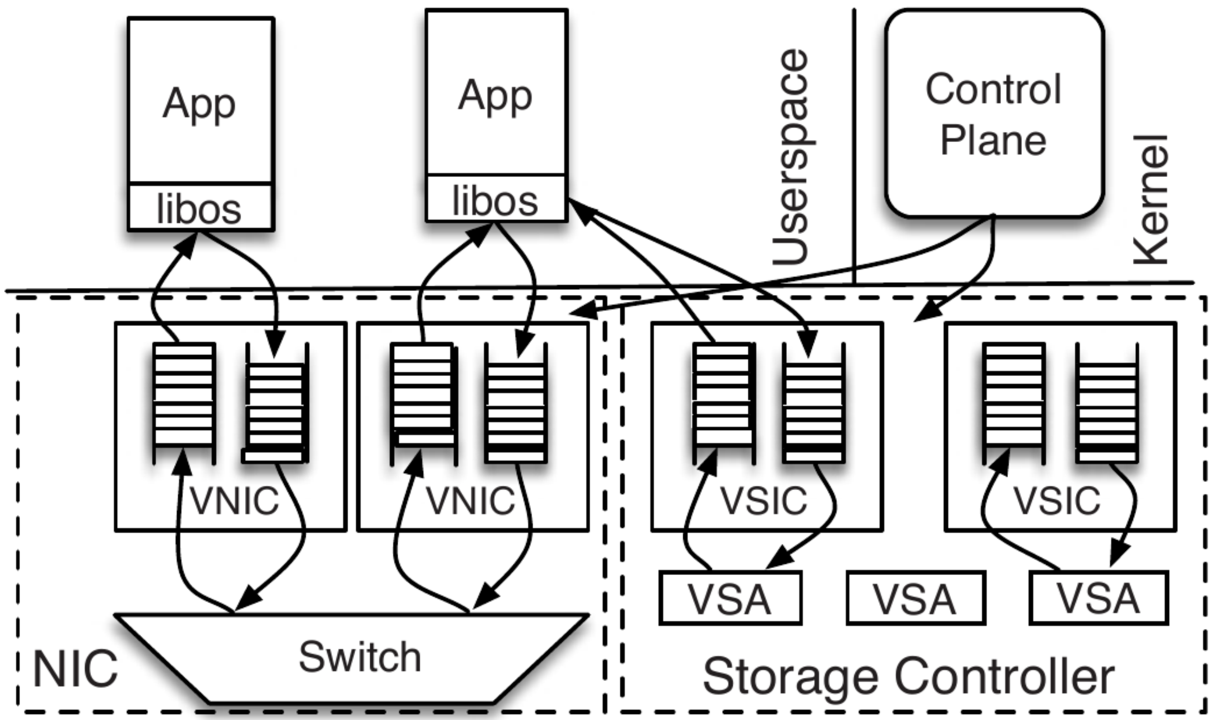


Figure 3: Arrakis architecture. The storage controller maps VSAs to physical storage.

Source: The Morning Paper: Arrakis - the operating system is the control plane, S. Peter, et al. OSDI 2014

Literature

- **Device drivers, problems, and solutions**
  - Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson R. Engler: *"An Empirical Study of Operating System Errors"*, SOSP 2001
  - Michael M. Swift, Brian N. Bershad, Henry M. Levy: *"Improving the Reliability of Commodity Operating Systems"*, SOSP 2003
  - Michael M. Swift, Brian N. Bershad, Henry M. Levy, Muthukaruppan Annamalai : *"Recovering Device Drivers", OSDI 2004*
  - Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz: *"Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines",* OSDI 2004
  - Galen Hunt, James Larus "Singularity: Rethinking the Software Stack", SIGOPS 2007
  - Leonid Ryzhyk et al.: *"Automatic Device Driver Synthesis with Termite"*, SOSP 2009
  - V. Chipounov, G. Candea: *"Reverse Engineering of Binary Device Drivers with RevNIC"*, EuroSys 2010
  - Alex Depoutovitch, Michael Stumm, "Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes", EuroSys 2010
  - N. Palix et al.: *"Faults in linux – 10 years later"*, ASPLOS 2011
  - Kantee, Antti. "Rump device drivers: Shine on you kernel diamond." AsiaBSDCon
  - H. Weisbach, B. Döbel, A. Lackorzynski: "Generic User-Level PCI Drivers", Real-Time Linux Workshop 2011
  - S. Peter: "Arrakis - the operating system is the control plane", OSDI 2014
  - A. Belay" "IX: a protected dataplane operating system for high throughput and low latency", OSDI 2014
  - DPDK, https://www.dpdk.org/
  - J. Corbet, A. Rubini, G Kroah-Hartman, "Linux Device Drivers, 3rd edition"
  - Robert Love, "Linux Kernel Development", 3rd edition

TECHNISCHE UNIVERSITÄT DRESDEN

Coming soon

- Today
  - Exercise: Paper discussion (Singularity)

- Nov 26th
  - Lecture: Real-Time and Microkernels