barkhausen institut

Hardware And Device Drivers

Lectures on Microkernel-Based Operating Systems (WS'25)

till.miemietz@barkhauseninstitut.org, with contents from Björn Döbel and Maksym Planeta

What Is A Device Driver?

- Definition of Tanenbaum [T09]:
  - *“The software that talks to a controller, giving it commands and accepting responses, is called a device driver.”*

- Software that renders hardware usable for software
  - Lowest level of system software
  - Interfaces directly with hardware
  - Provides access to hardware-specific functions

A Quick Tour of (Modern) Computer Hardware

What Is Inside a Modern Computer?

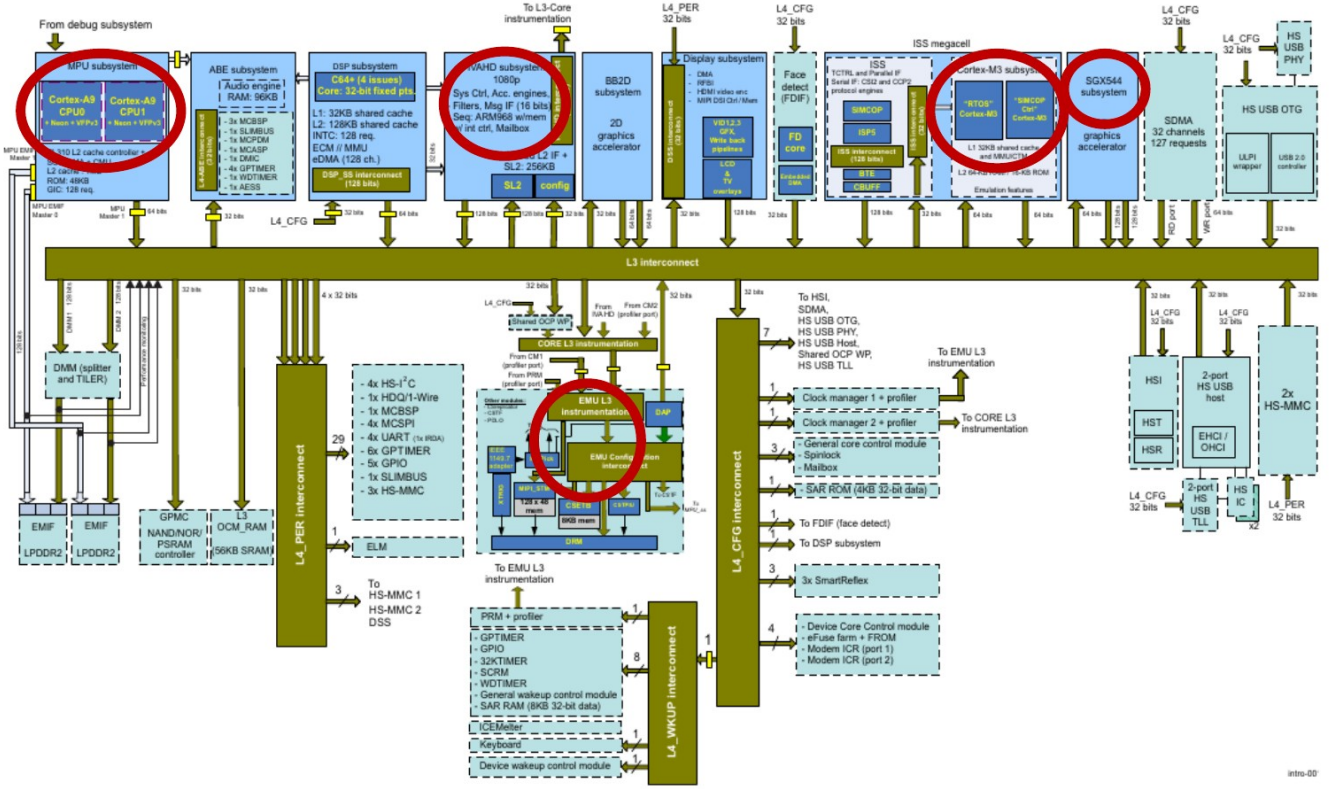
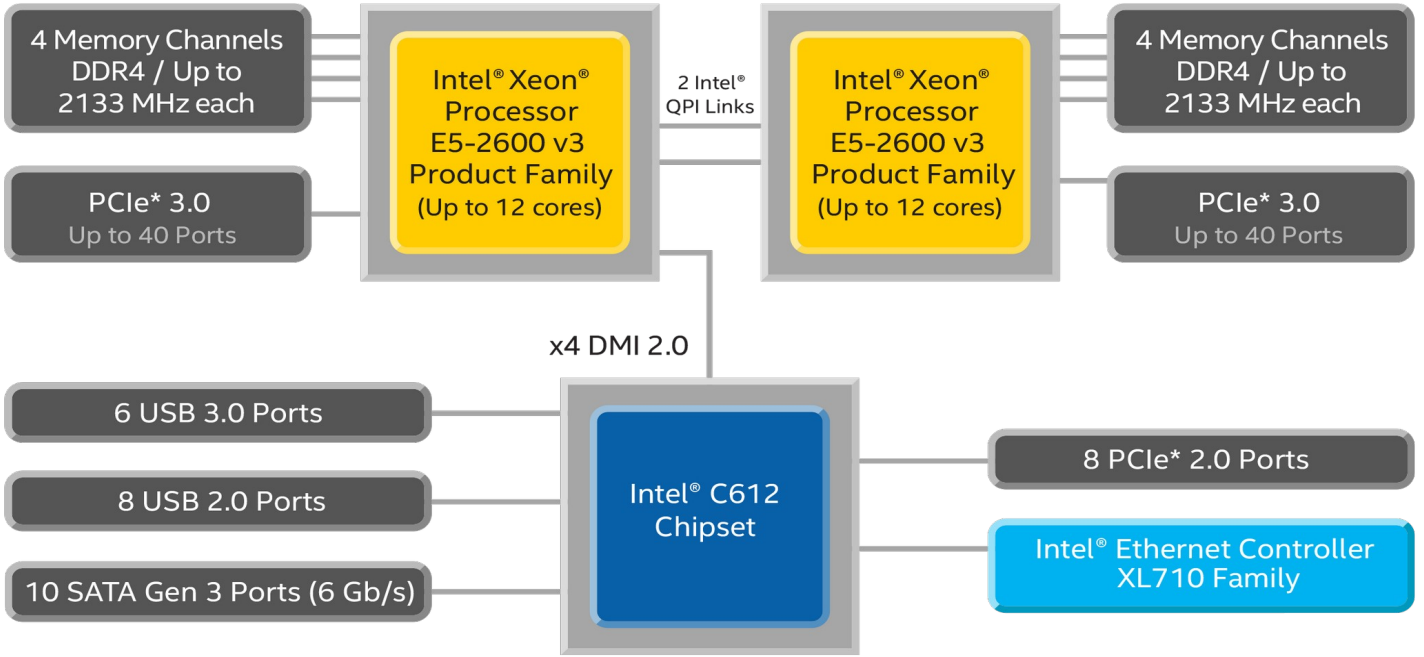


Image taken from: Timothy Roscoe - It's time for Operating Systems to rediscover Hardware, Keynote at OSDI'21

What Is Inside a Modern Computer?

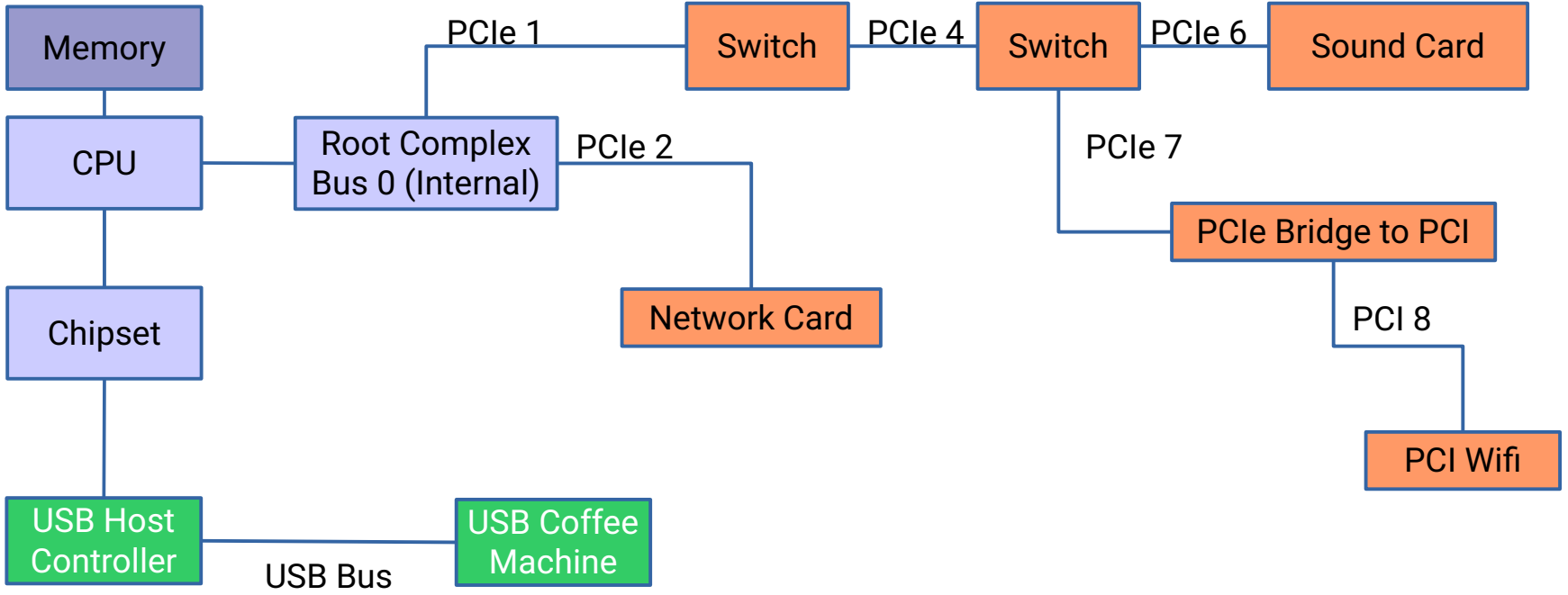
- Let’s simplify hardware for education purposes:



Intel c612 Chipset (source: intel.com)

System Buses

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

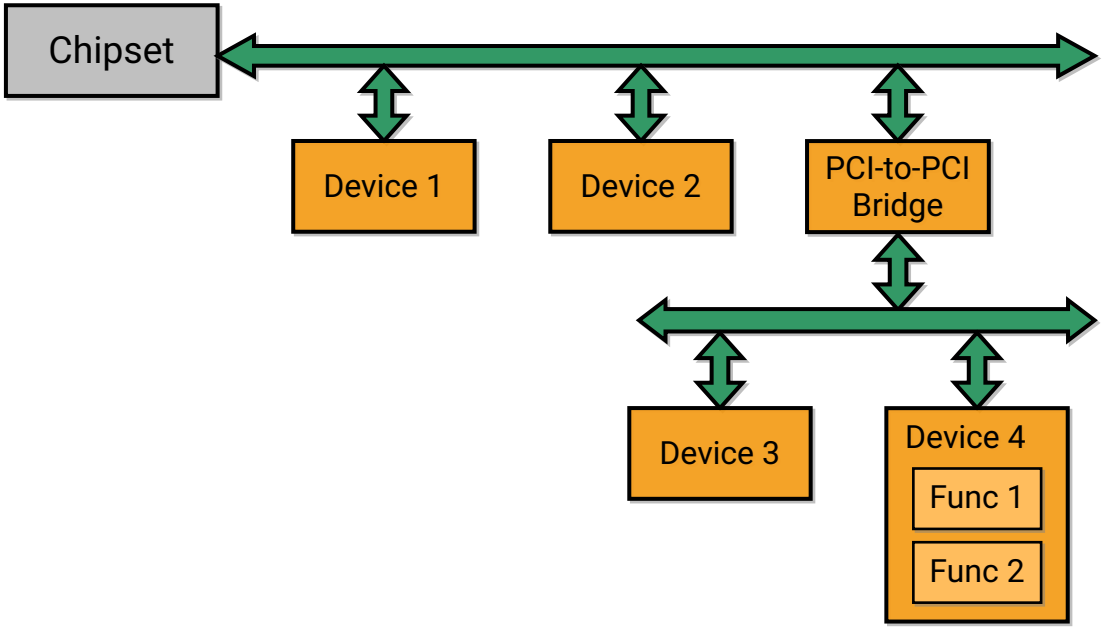


Buses And Devices

- 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

Peripheral Component Interconnect (PCI)

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



Peripheral Component Interconnect (PCI)

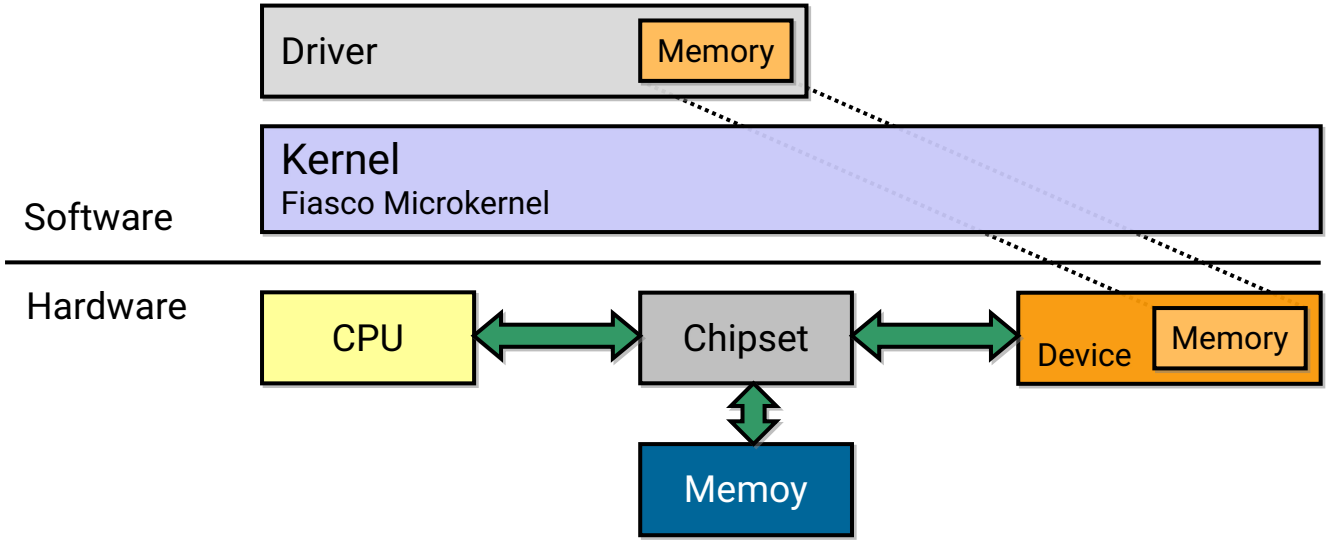
- PCI configuration space

- 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

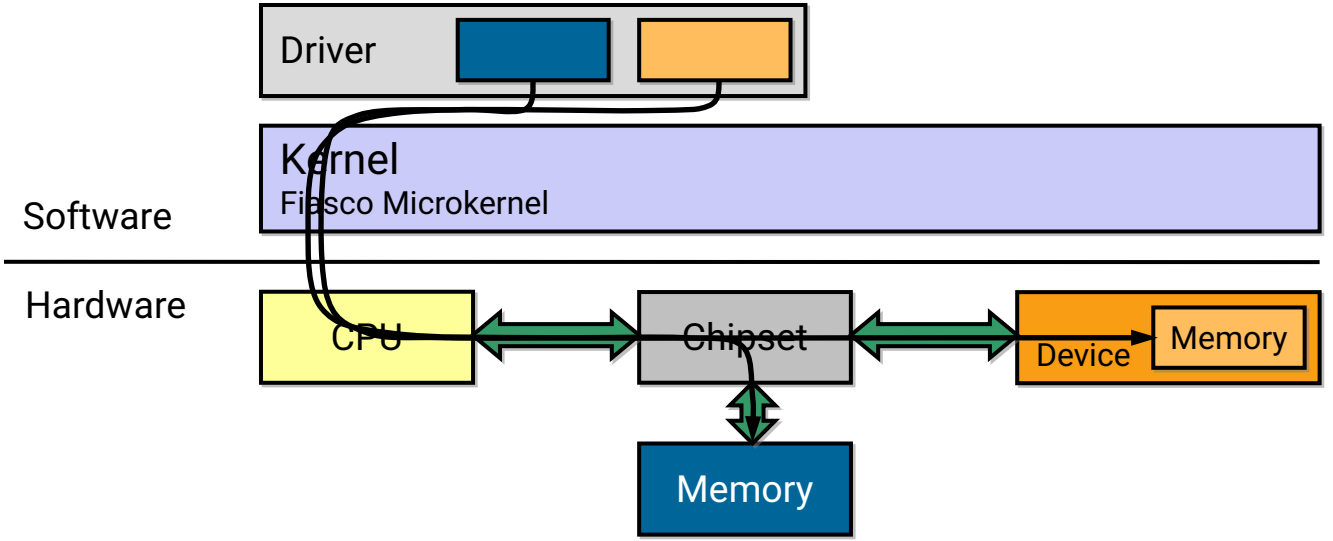
I/O Memory

- 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

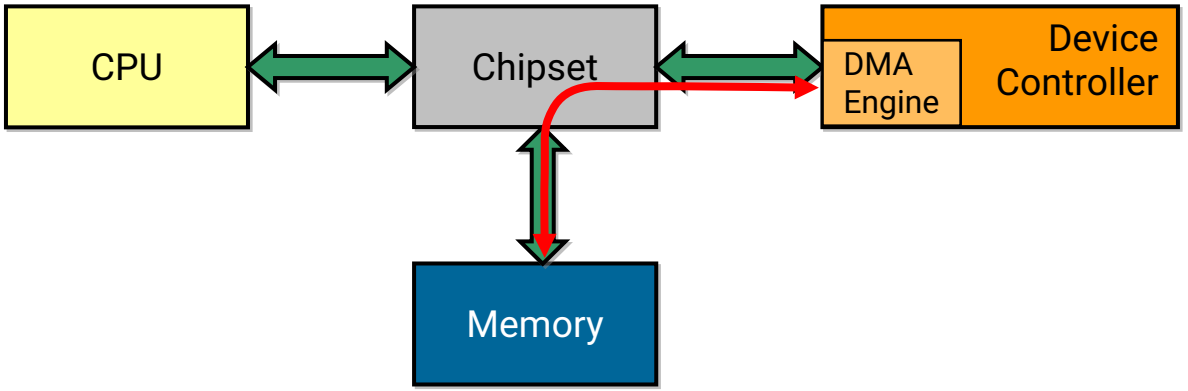
- Device memory looks just like phys. memory
- Chipset needs to
  - map I/O memory to exclusive address ranges
  - distinguish physical and I/O memory access



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)



I/O Ports

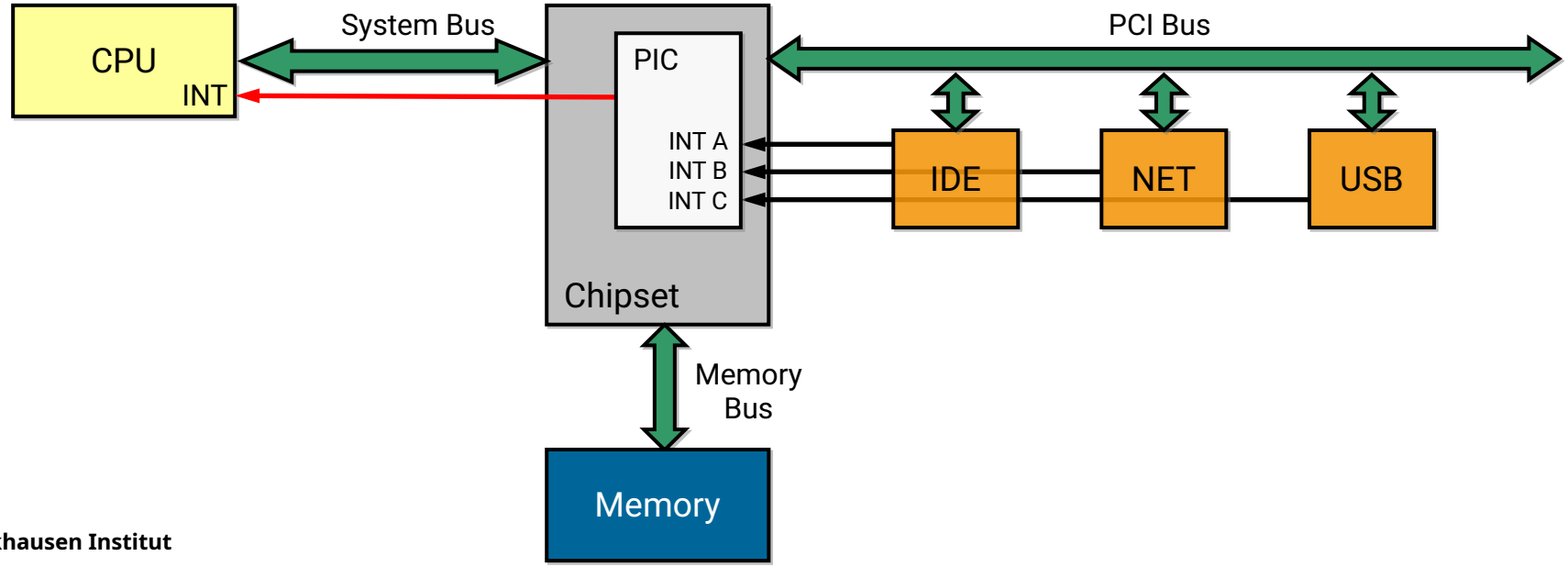
- x86-specific feature
- I/O ports define own I/O address space
  - Each device uses its own area within this address space
- Special instruction to access I/O ports
  - in / out: I/O read / write
- Example: read byte from serial port

```
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

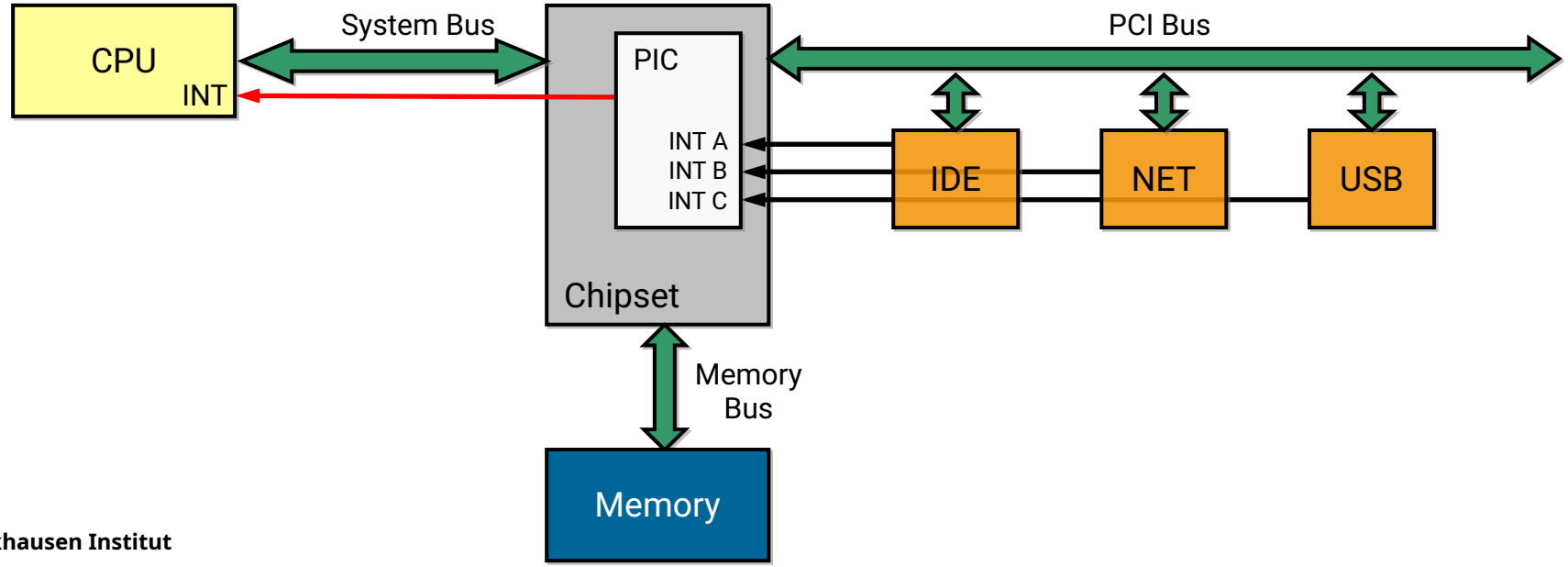
Interrupts

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



Interrupts

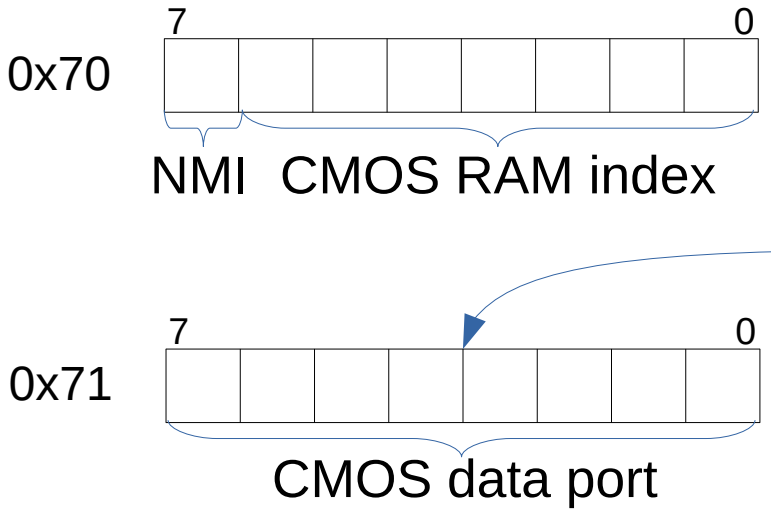
- Handling interrupts involves
  - examine / manipulate device
  - program PIC
  - acknowledge/mask/unmask interrupts



Device Drivers On Monolithic Kernels (Linux)

CMOS Clock Example

- 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

CMOS Clock Example

- File in the /dev filesystem

- Read the value

  $ cat /dev/rtctest

  14:05:44 24.11.2020

CMOS Clock Example

```
/* 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);
```

CMOS Clock Example

```
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;
}
```

CMOS Clock Example

```
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;
 /* ... */
}
```

CMOS Clock Example

```
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);
}
```

CMOS Clock Example

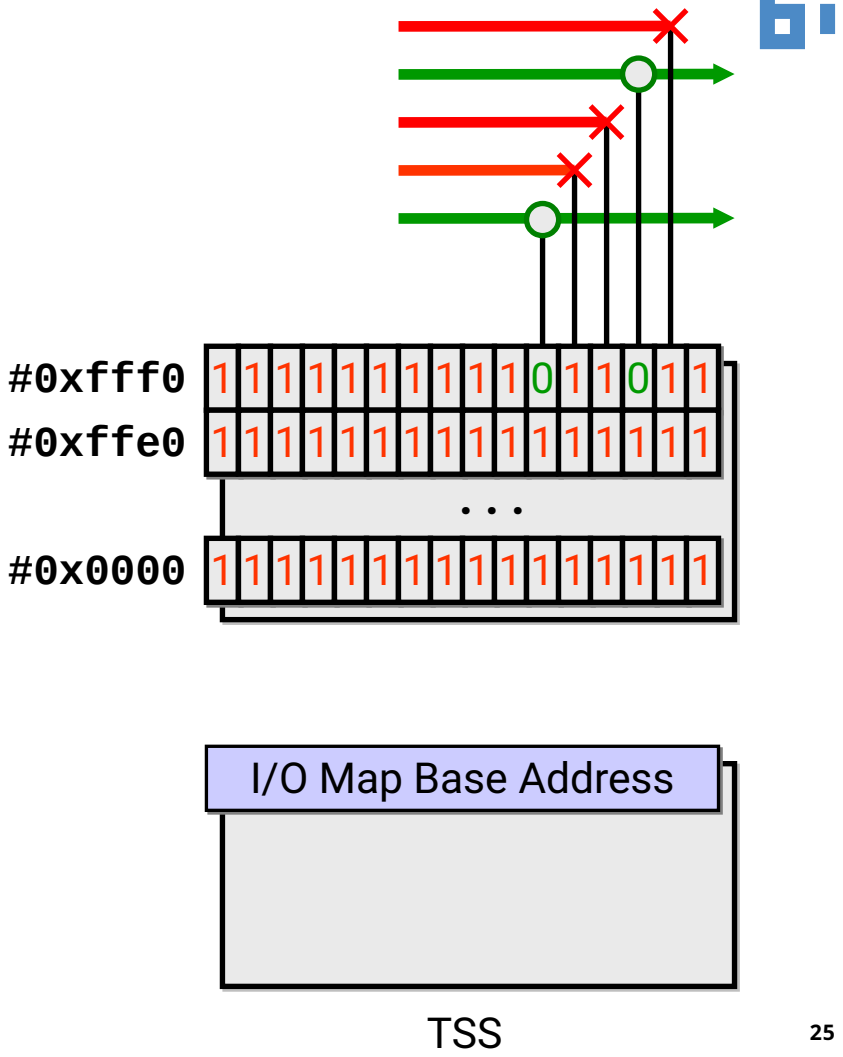
```
static void __exit rtctest_exit(void){
  release_region(RTC_PORT_START, RTC_PORT_NUM);
  device_destroy(rtctestClass, MKDEV(majorNumber, 0));     // remove the device
  class_unregister(rtctestClass);                          // unregister the device class
  class_destroy(rtctestClass);         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

IRQ Handling in Linux

- 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

IRQ Handling in Linux

- 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)



DMA Security Problems

- DMA uses physical addresses.
  - I/O memory regions need to be physically contiguous
- Buffers must not be paged out during DMA → Memory pinning

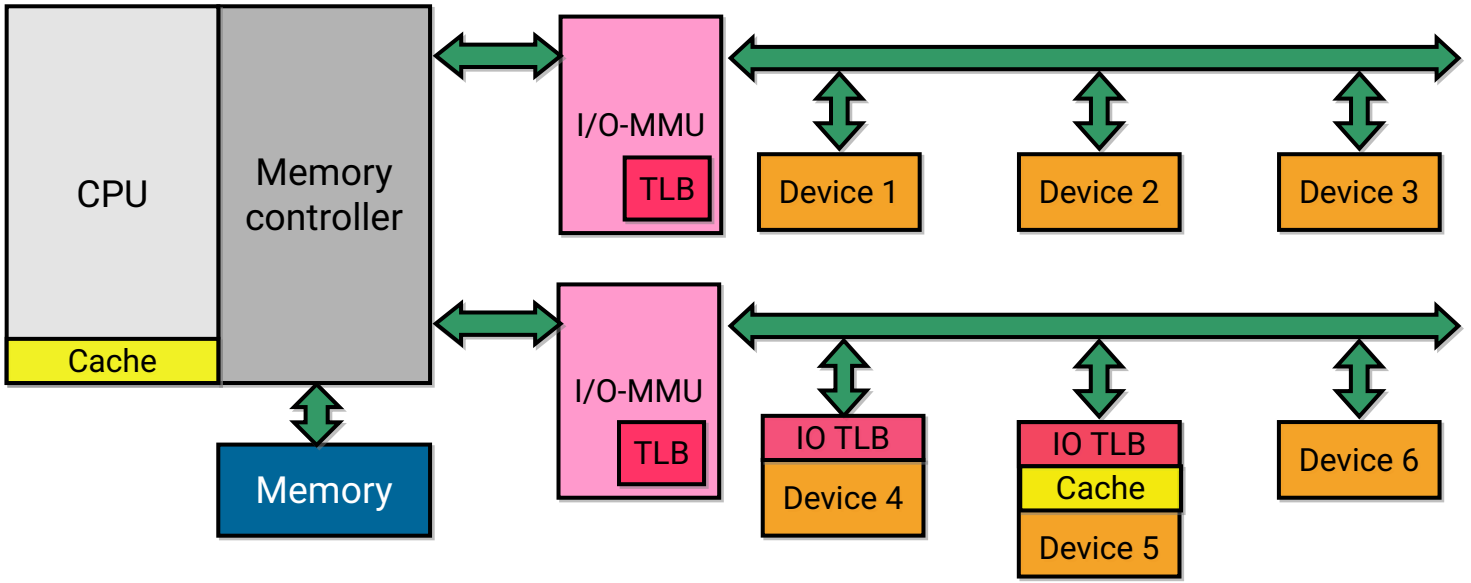
- DMA with phys. addresses bypasses VM management
  - Drivers can overwrite any physical Address

- DMA is both a safety and a security risk.
- Which mechanism do you know to protect untrusted software from accessing physical memory?

I/O MMUs

- 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

Drivers Running in the Kernel – A Smart Idea?

- Which problems do you see?

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

Some Statistics

- [SB+03]: Drivers cause 85% of Windows XP crashes.
- [CY+01]: 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 close)
- Life expectancy of a bug in the Linux kernel (~2.4): 1.8 years
- [R+09]: Causes for driver bugs
  - 23% programming error
  - 38% mismatch regarding device specification
  - 39% OS-driver-interface misconceptions
- [XZ+19]: “bugs related [...] Drivers and ACPI, account for 51.6% of all classified bugs”

Anecdote: The e1000 NVRAM Bug of Linux

- 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

The Traditional Approach has Issues

- 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)

Device Drivers On Microkernels (L4Re)

Idea: Drivers in Userspace

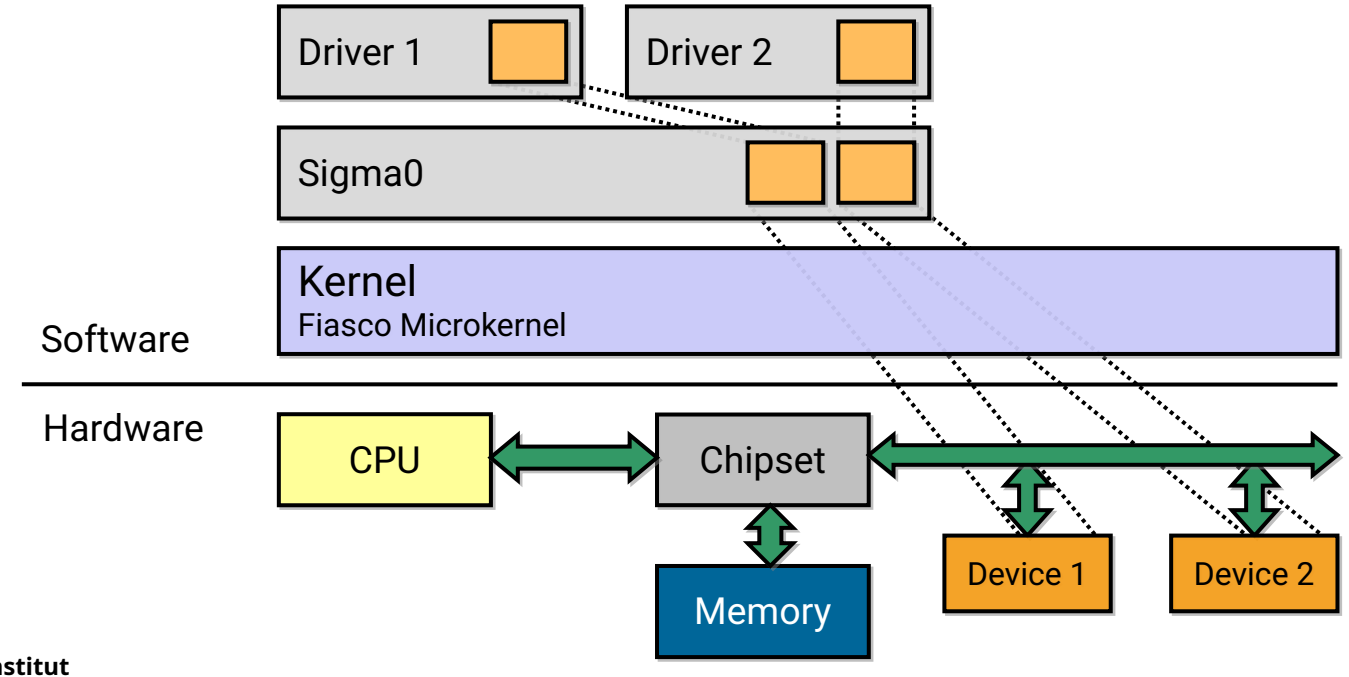
- Isolate components
  - device drivers (disk, network, graphic, USB cruise missiles, …)
  - stacks (TCP/IP, file systems, …)
- Separate address spaces each: More robust components
- Problems
  - Overhead
  - HW multiplexing
  - Context switches
  - Need to handle I/O privileges

How To Access Hardware Resources?

- 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 L4Re

- 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



Interrupt Handling in L4Re

- 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

Interrupt Handling in L4Re

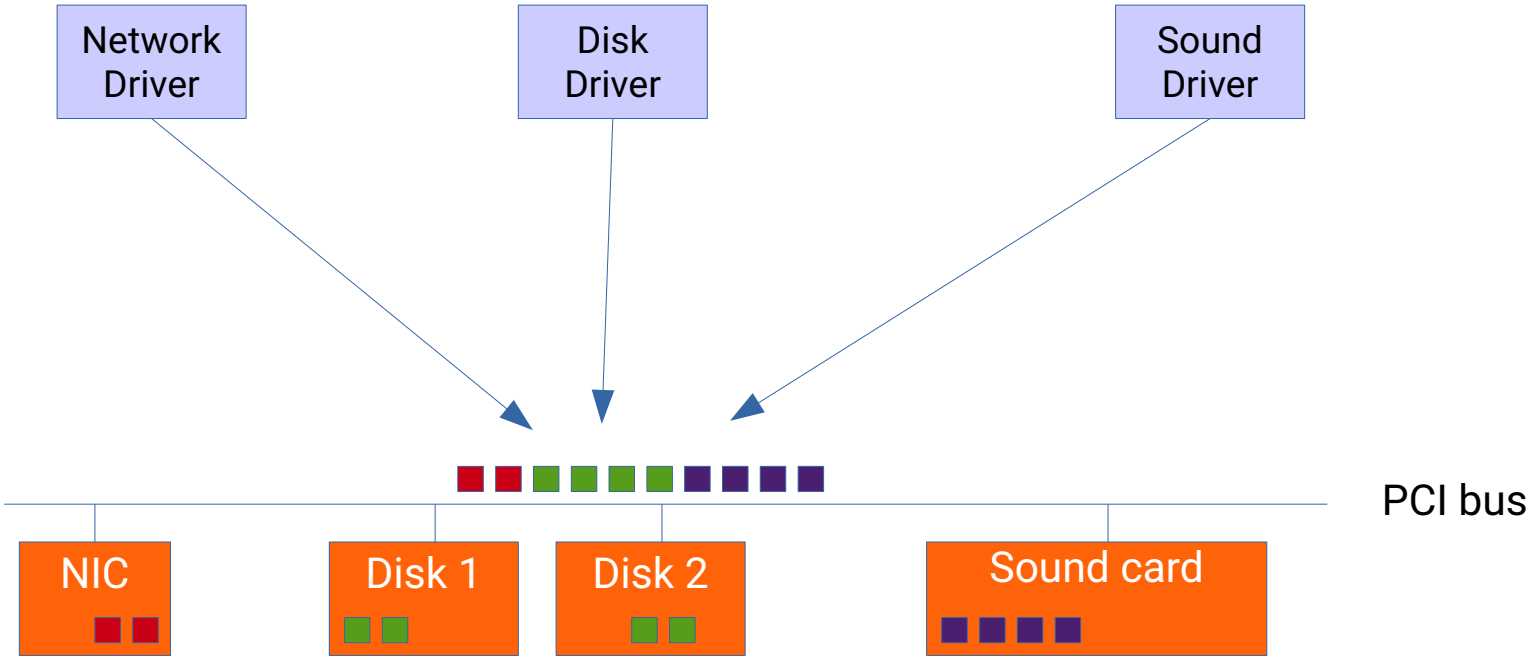
- CLI – only with IO Privilege Level (IOPL) 3
- Should not be allowed for every user-level driver
  - untrusted drivers
  - security risk
- Observation: drivers often don't need to disable IRQs globally, but only access to their own IRQ
  - Just don't receive from your IRQ

Restricting I/O Port Access in L4Re

- Per-task I/O bitmap
- Switched during task switch
- Allows per-task grant/deny of I/O port access

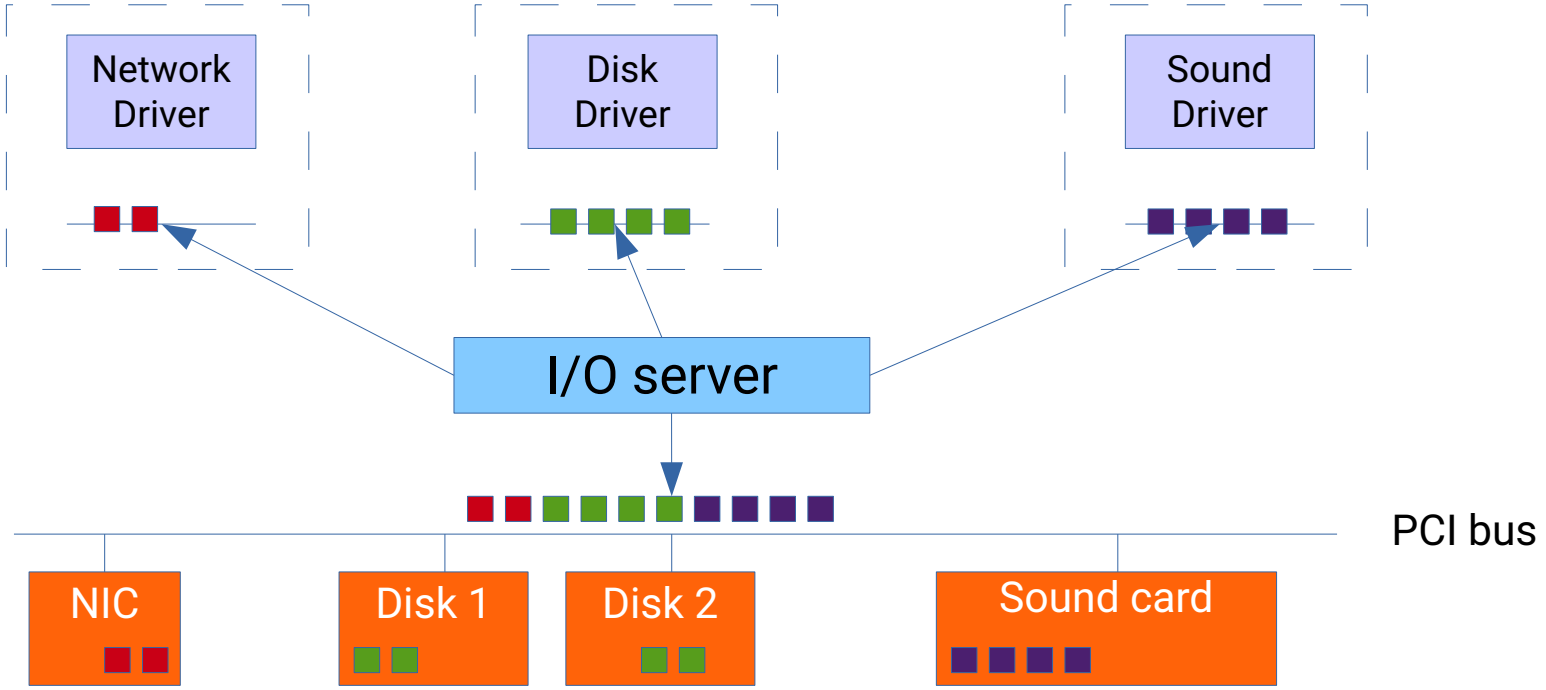
Restricting Untrusted Drivers

- How to enforce device access policies on untrusted drivers?



Restricting Untrusted Drivers

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



Reflections...

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

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

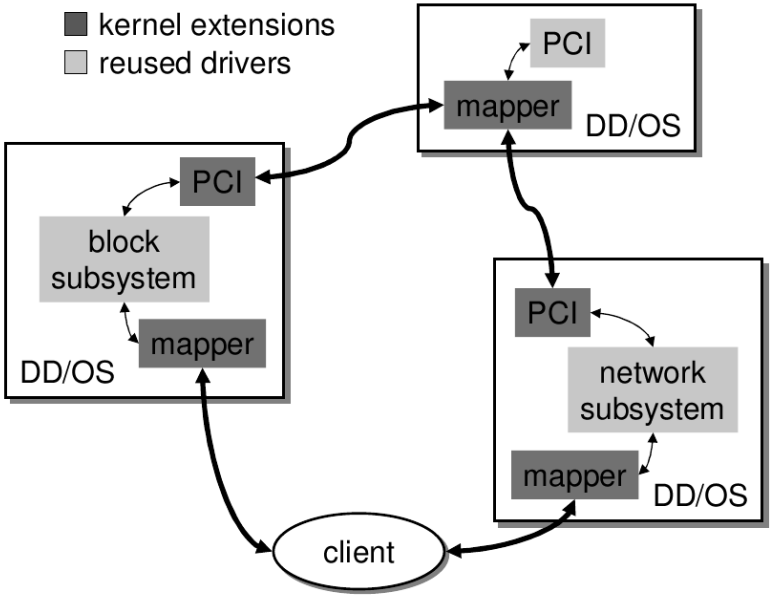
Reusing Device Drivers

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

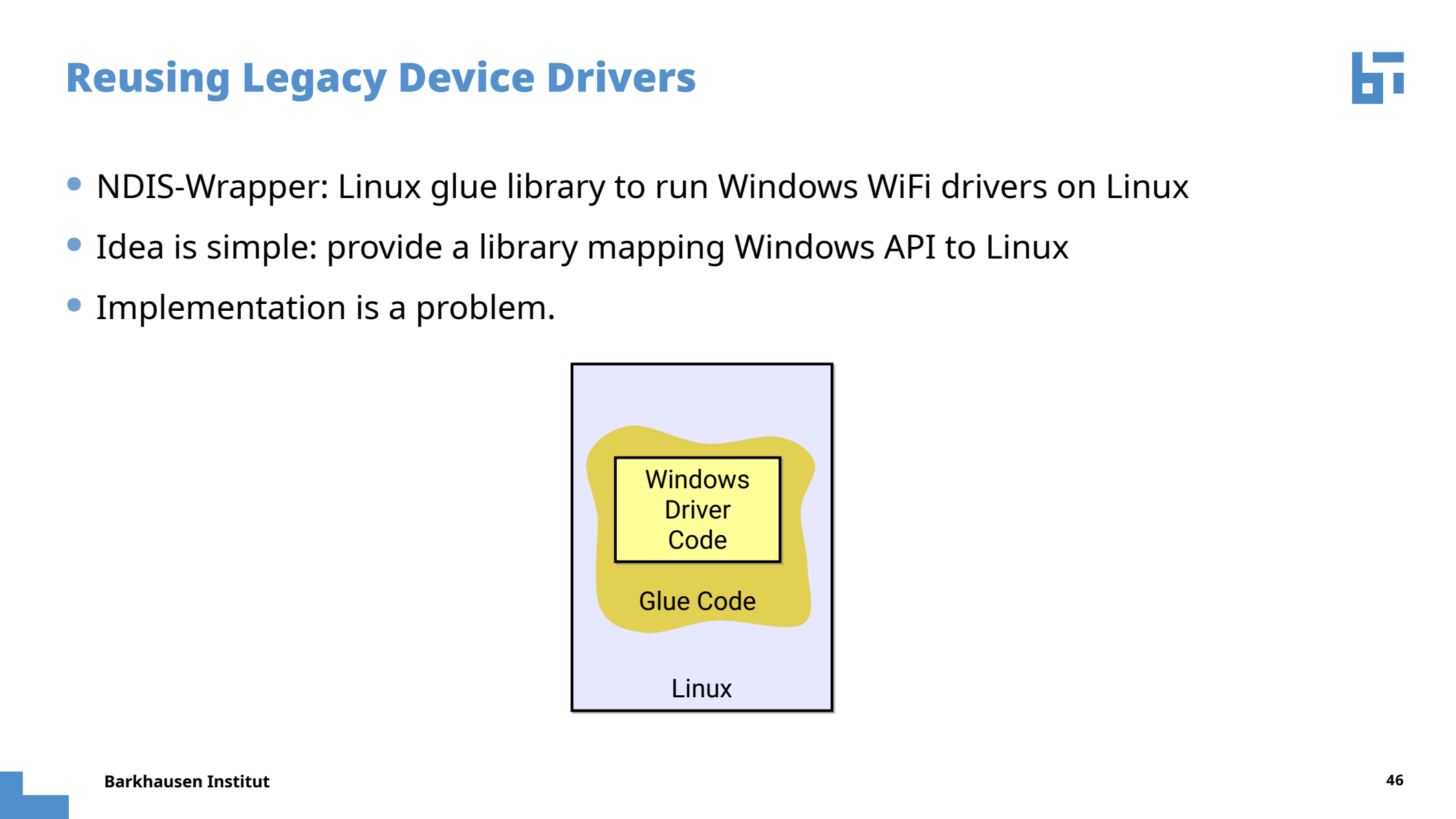
Reusing Legacy Device Drivers

- Exploit virtualization: Device Driver OS



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

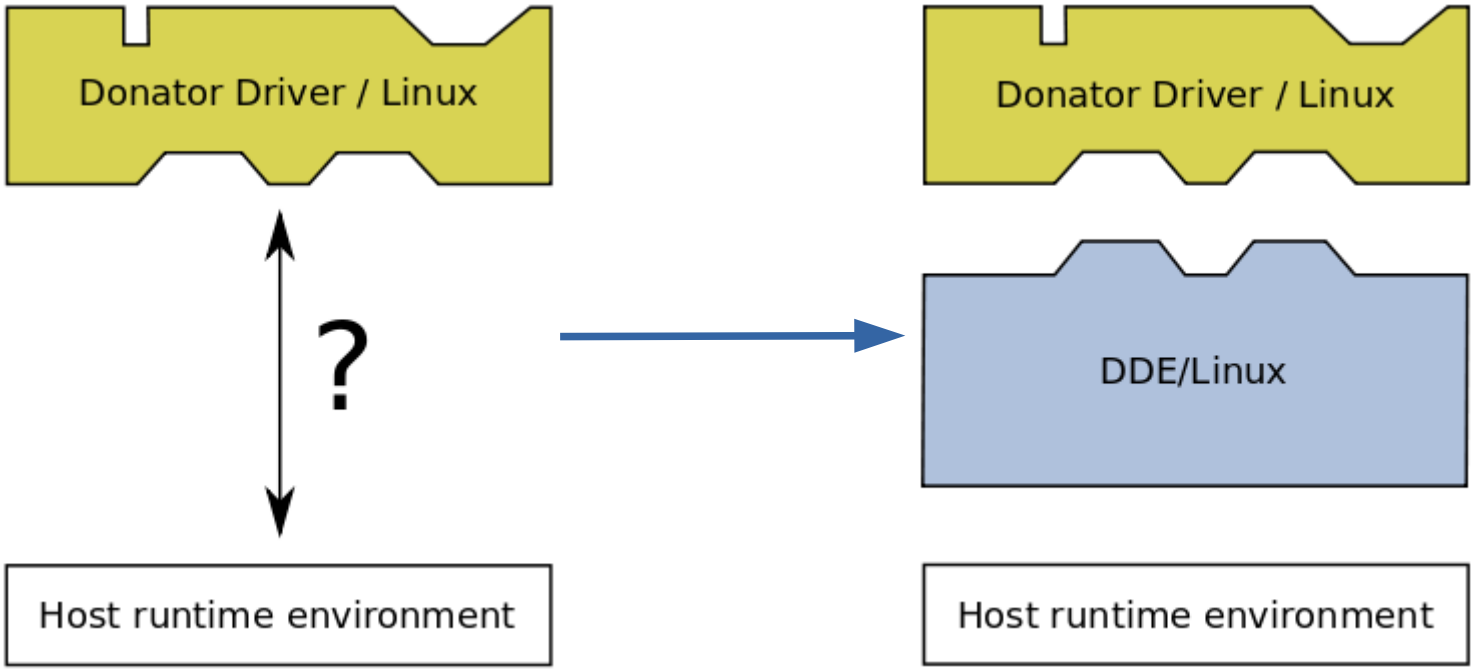
Reusing Legacy Device Drivers



- 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 II

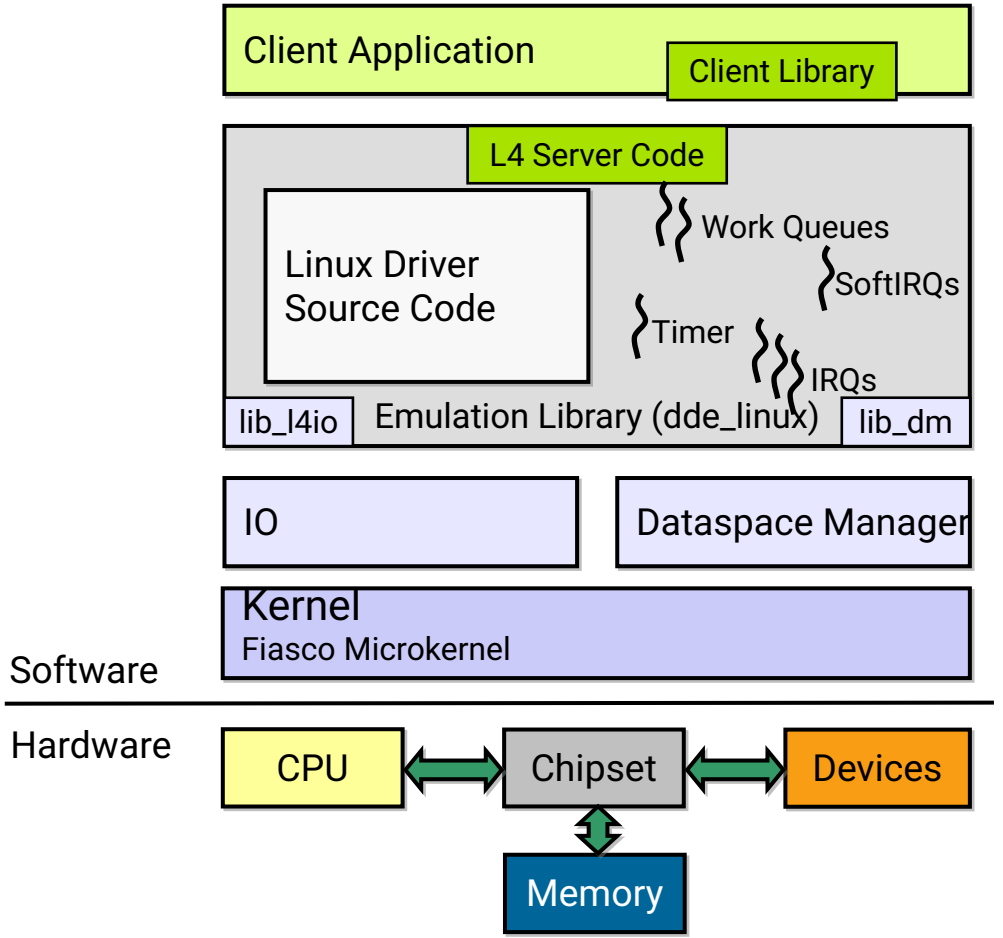
- Generalize the idea: provide a Linux environment to run drivers on L4
  - Device Driver Environment (DDE) [WB+11]



Emulating Linux: DDE/Linux

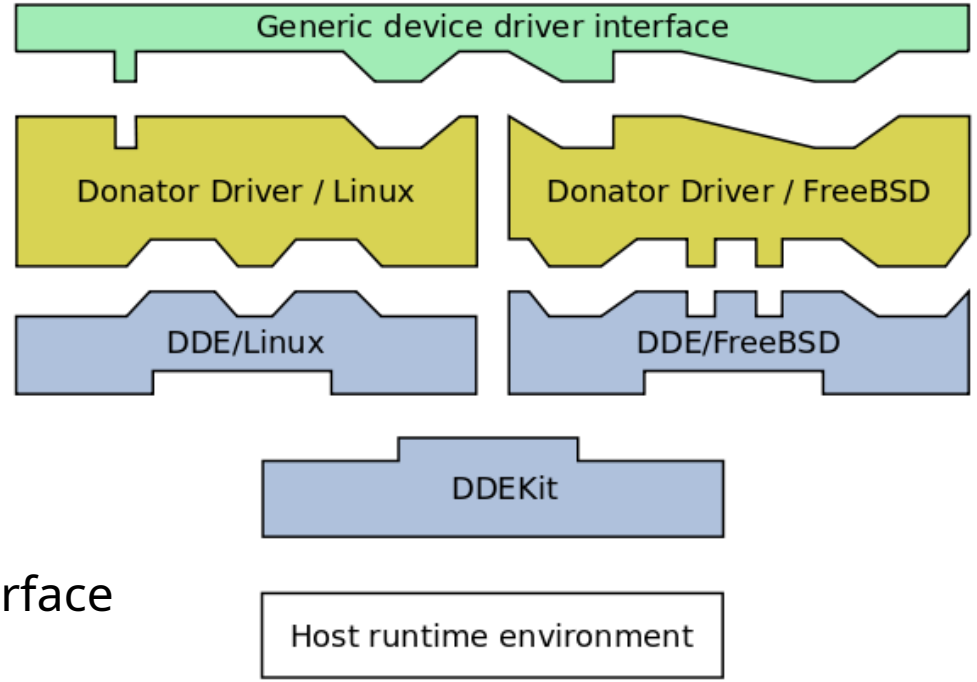
- Multiple L4 threads provide a Linux environment
  - Workqueues
  - SoftIRQs
  - Timers
  - Jiffies
- Emulate SMP-like system (each L4 thread assumed to be one processor)
- Wrap Linux functionality
  - kmalloc() → L4 Slab allocator library
  - Linux spinlock → pthread mutex
- Handle in-kernel accesses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

DDE Structure



DDE Kit: Another Abstraction

- Pull common abstractions into dedicated library
  - Threads
  - Synchronization
  - Memory
  - IRQ handling
  - I/O port access
- → DDE Construction Kit (DDEKit)



- Implement DDEs against the DDEKit interface

Applying Formal Methods To Device Drivers

Formalizing Drivers: Dingo [RC+09]

- Observations:
  - drivers fail to obey device spec
  - developers misunderstand OS interface
  - multithreading is bad
- Tingu: state-chart-based
  - specification of device
  - protocols
- Event-based state transition
  - Timeouts
  - Variables

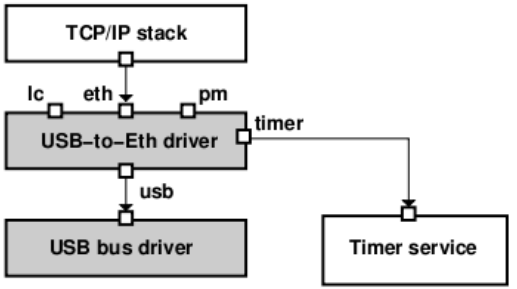
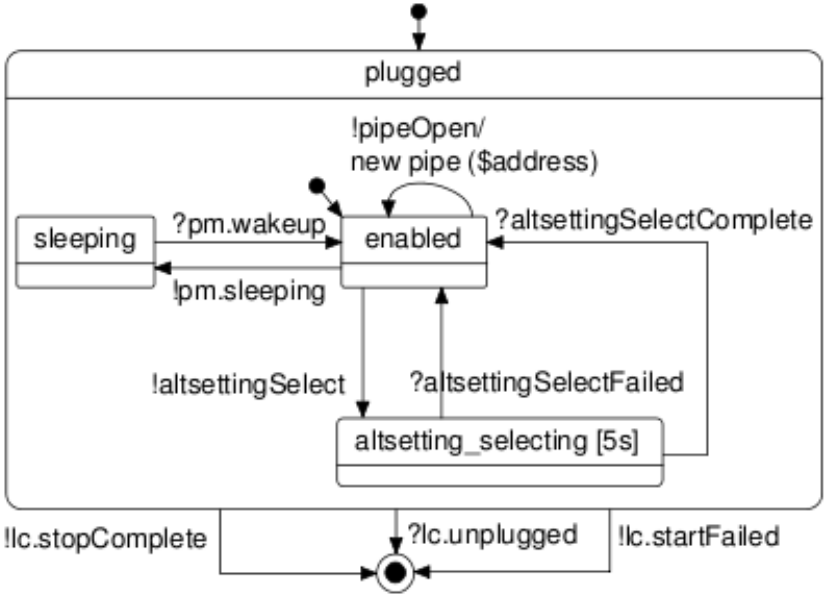


Figure 3. Ports of the USB-to-Ethernet adapter driver.



Dingo

- Dingo: device driver architecture
- Single-threaded
  - Builtin atomicity
  - Not a performance problem for most drivers
- Event-based
  - Developers implement a Tingu specification
- Can use Tingu specs to generate runtime driver monitors

References for Further Reading

[T09] Andrew S. Tanenbaum: “Modern operating systems”, 3rd Edition. Pearson Prentice-Hall 2009, ISBN 0138134596.

[SB+03] Michael M. Swift, Brian N. Bershad, Henry M. Levy: “Improving the Reliability of Commodity Operating Systems”, SOSP 2003

[CY+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson R. Engler: “An Empirical Study of Operating System Errors”, SOSP 2001

[R+09] Leonid Ryzhyk et al.: “Automatic Device Driver Synthesis with Termite”, SOSP 2009

[RC+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Gernot Heiser: “Dingo: taming device drivers”, EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems, 2009

[XZ+19] Guanping Xiao, Zheng Zheng, Bo Jiang, Yulei Sui: “An Empirical Study of Regression Bug Chains in Linux”, IEEE Transactions on Reliability, 2019

[WB+11] Hannes Weisbach, Björn Döbel, Adam Lackorzynski: “Generic User-Level PCI Drivers”, Real-Time Linux Workshop 2011