TECHNISCHE
UNIVERSITÄT
DRESDEN

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

Hardware and Device Drivers

Björn Döbel

Dresden, 2010-11-16

Outline

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

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

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

Idea: User-level Drivers

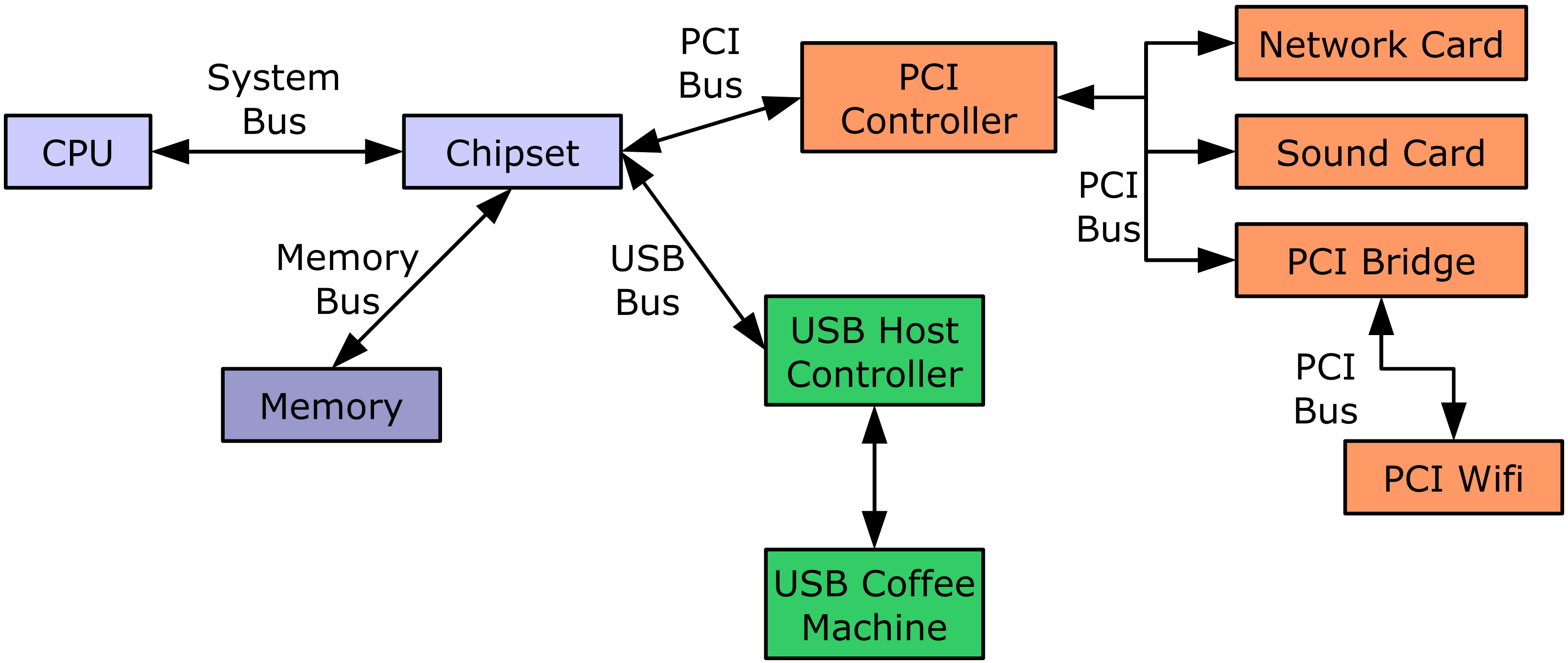
- Isolate components
  - device drivers (disk, network, graphic, USB cruise missiles, ...)
  - stacks (TCP/IP, file systems, …)
- Separate address spaces each
  - More robust components
- Problems
  - Overhead
    - HW multiplexing
    - Context switches
  - Need to handle I/O privileges

Break

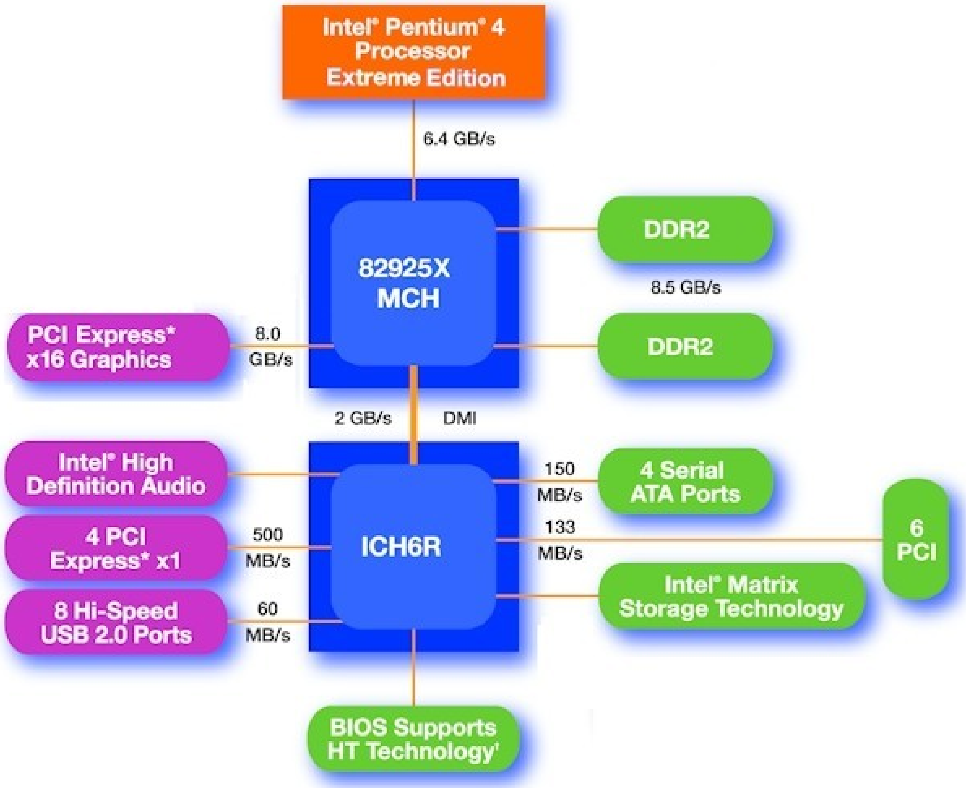
- Need special care for device drivers.
- Next: A closer look on how hardware works.

System Layout

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



Real World Example



Intel 925x Chipset
(source: http://www.intel.com)

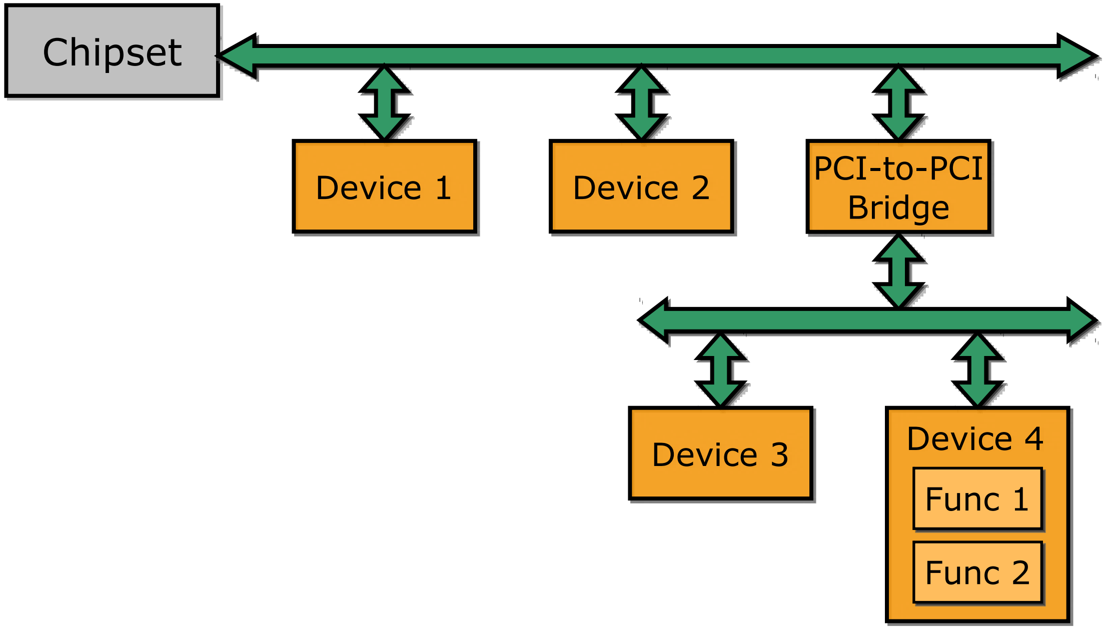
† Hyper-Threading (HT) Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See www.intel.com/info/hyperthreading for more information including details on which processors support HT Technology.

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

Buses: USB

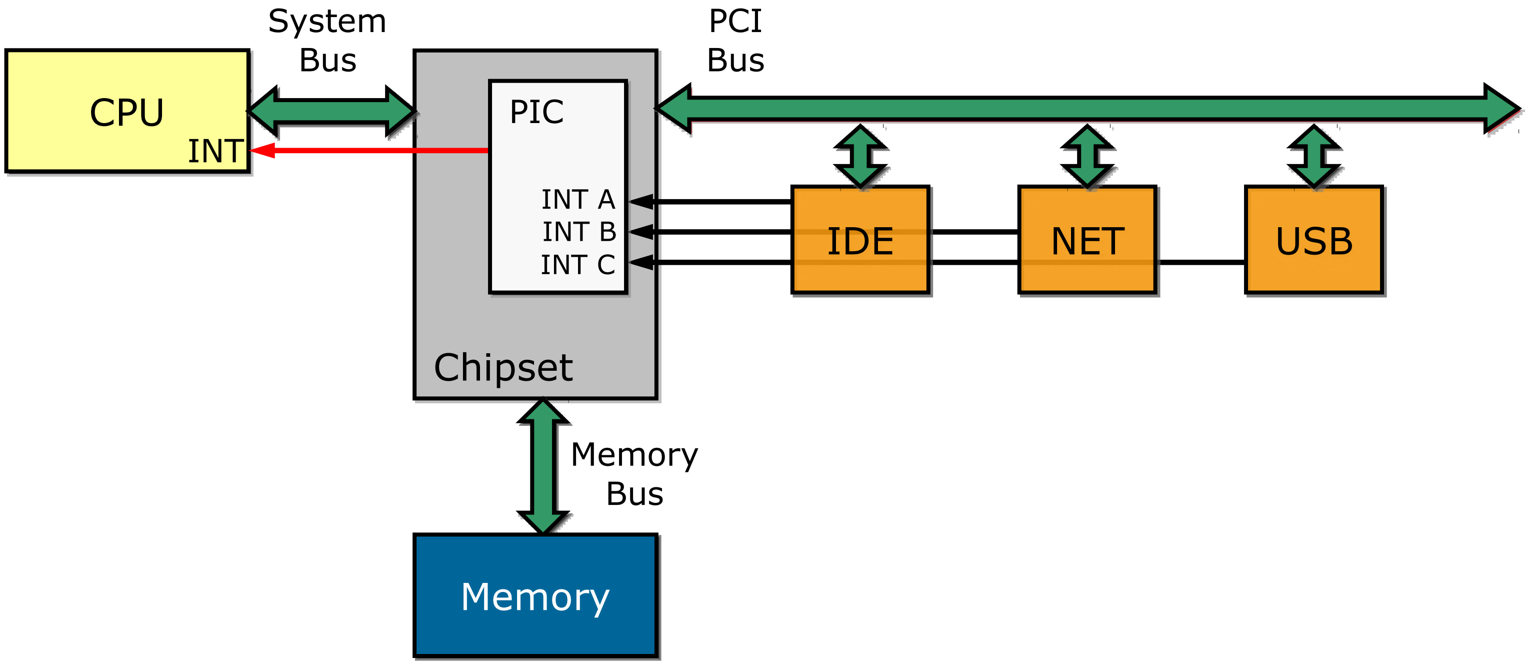
- Intel, 1996
- One bus to rule them all?
  - Firewire has always been faster
- Tree of devices
  - root = Host Controller (UHCI, OHCI, EHCI)
  - Device drivers use HC to communicate with their device via USB Request Blocks (URBs)
  - USB is a serial bus
    - HC serializes URBs
- Wide range of device classes (input, storage, peripherals, ...)
  - classes allow generic drivers

Break

- Someone (BIOS) organizes physical hierarchy of devices → buses.
- Devices need to interact with the rest of the system
  - Interrupts
  - I/O ports
  - Memory-mapped I/O registers

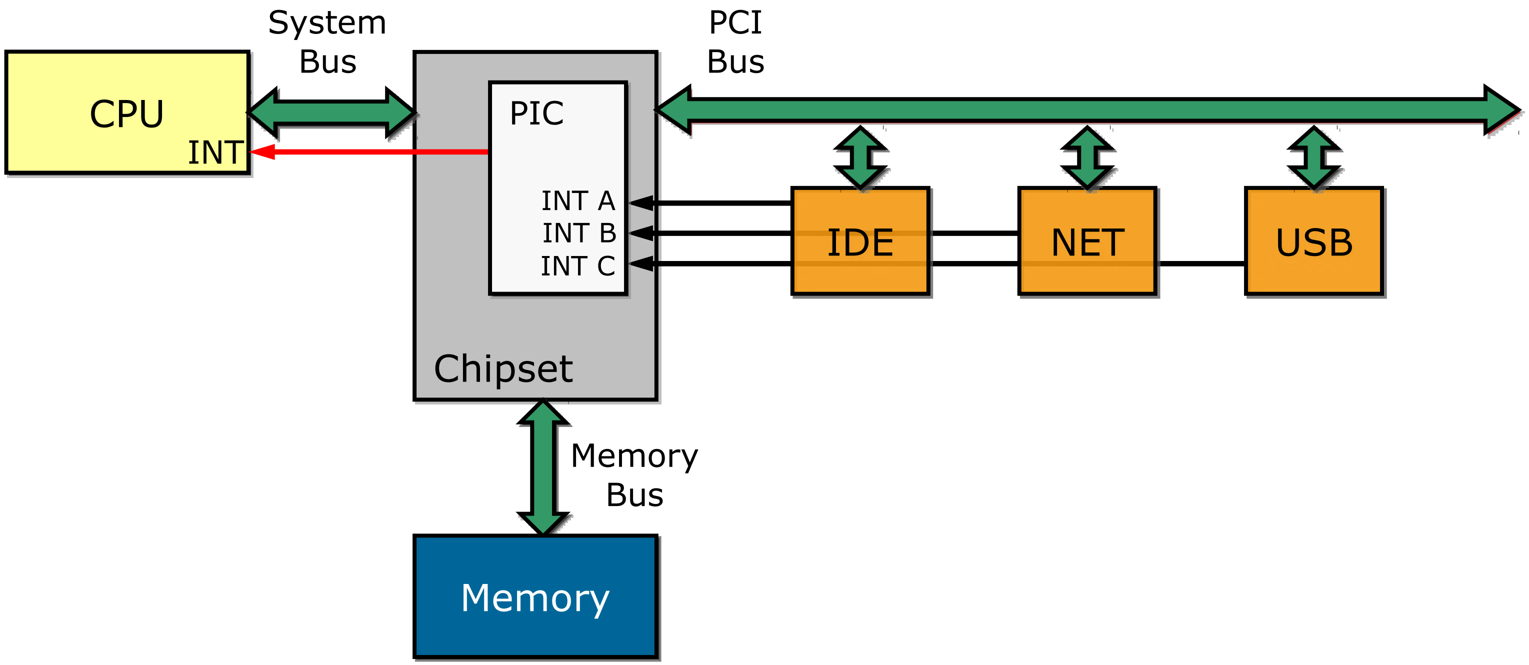
Interrupts

- 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
  - unstrusted 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

I/O ports

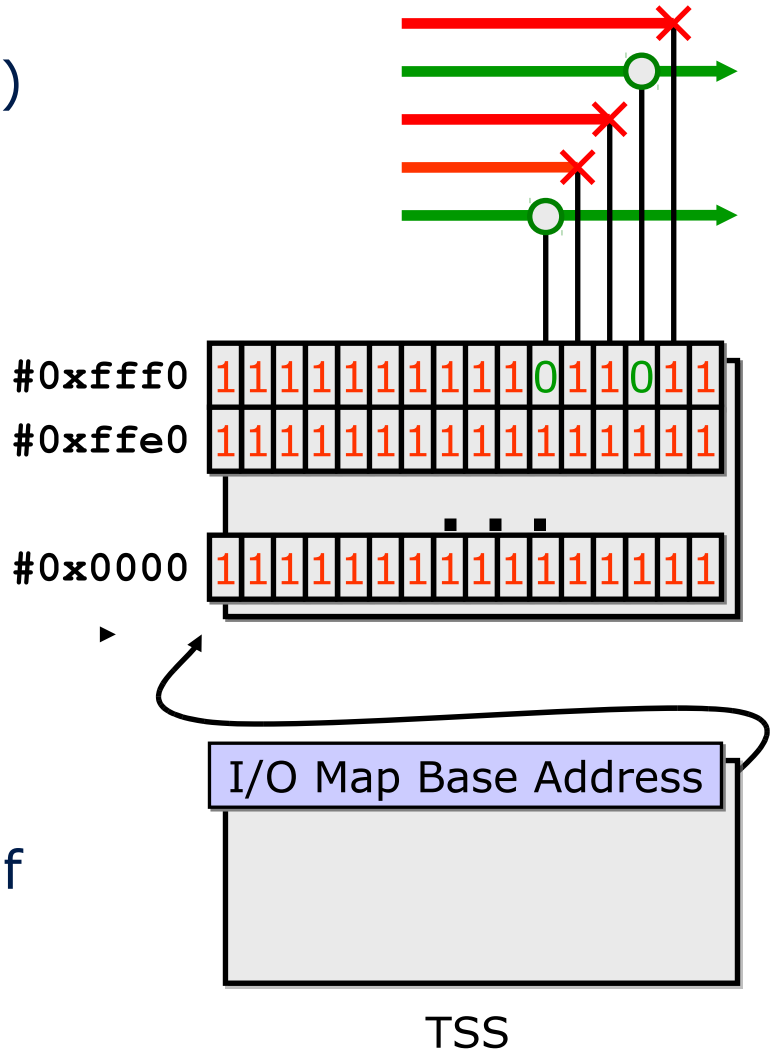
- x86-specific feature
- I/O ports define own I/O address space
  - Each device uses its own area within this address space
- Special instruction to access I/O ports
  - in / out: I/O read / write
  - Example: read byte from serial port

```
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

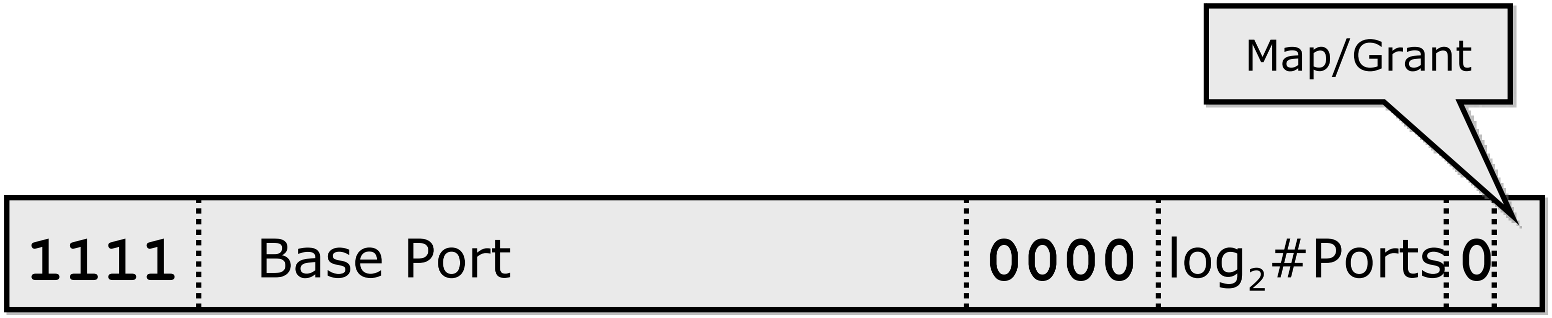
I/O Bitmap

- 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 Flexpages

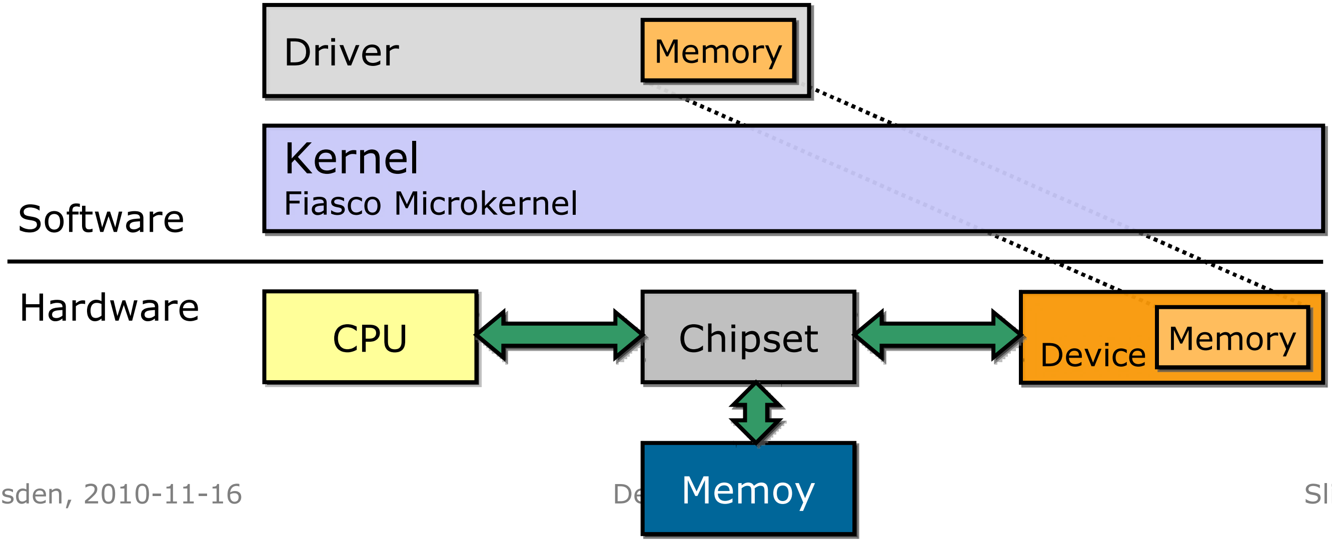
- Reuse kernel's map/grant mechanism for mapping I/O port rights -> I/O flexpages
- Kernel detects type of flexpage and acts accordingly
- Task with all I/O ports mapped is raised to IOPL 3



L4.Fiasco I/O flexpage format

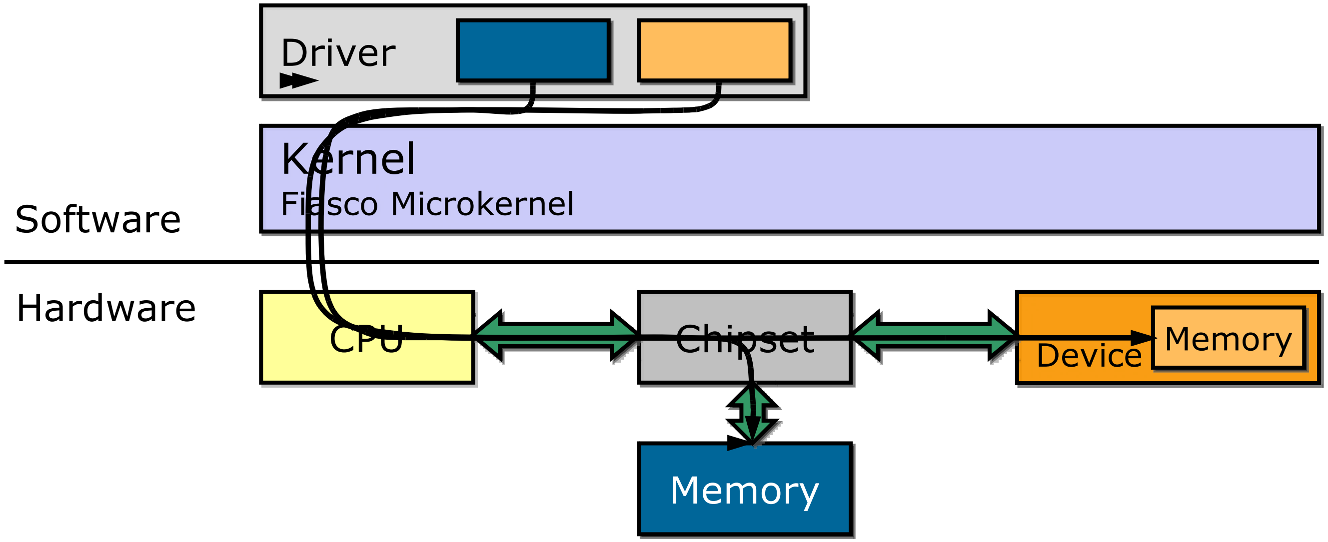
I/O Memory

- Devices often contain on-chip memory (NICs, graphcis cards, ...)
- Instead of accessing through I/O ports, 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



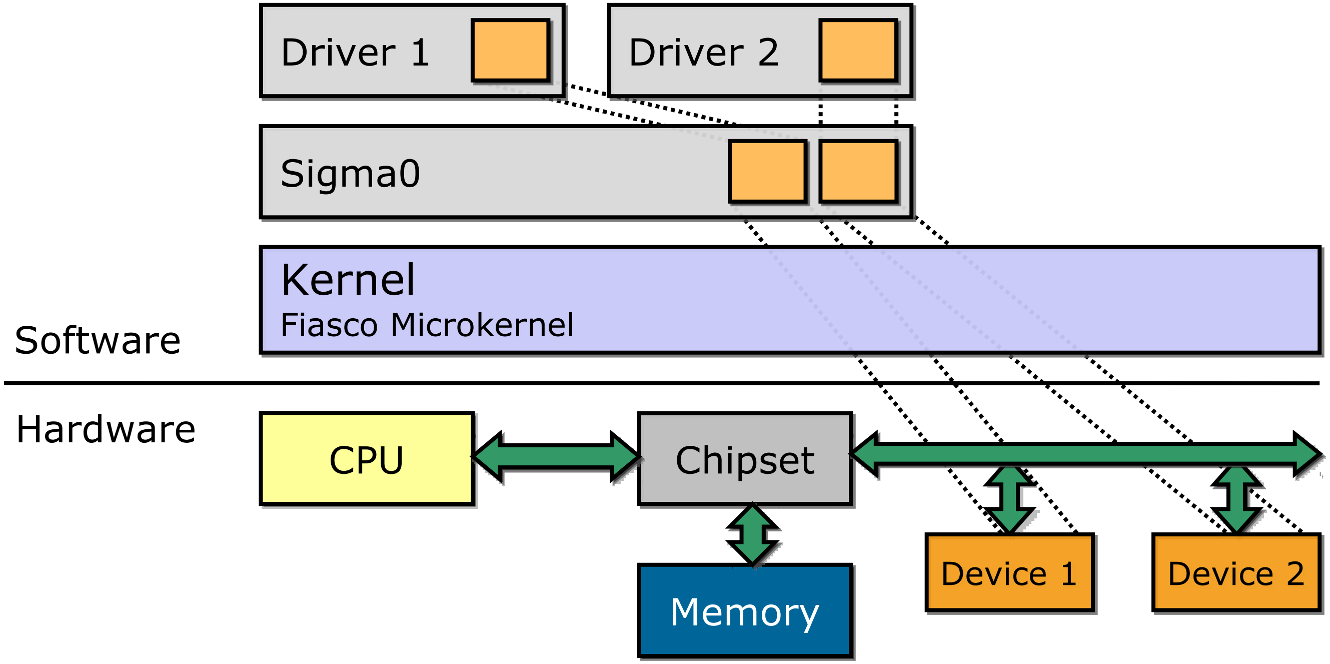
I/O memory (2)

- 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



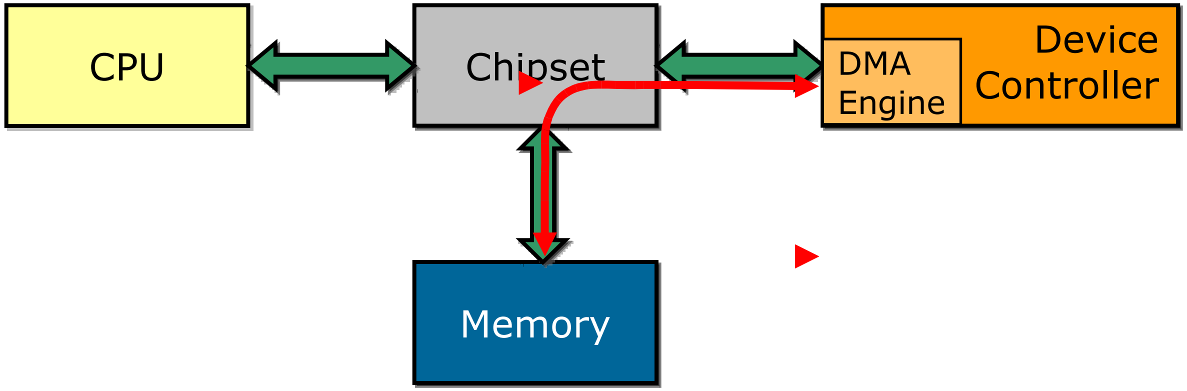
I/O memory in L4

- 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)



Problems with DMA

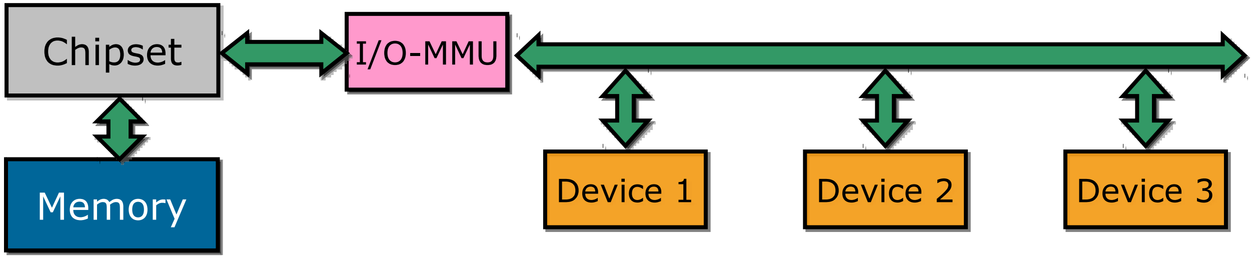
- DMA mostly uses physical addresses.
- Drivers need to know these addresses.
- Buffers must not be paged out (pinned pages) during DMA
  - close interaction with memory management
- DMA with phys. addresses bypasses VM management
  - Drivers can overwrite any phys. Address
- DMA is a security risk.

Idea: I/O MMU

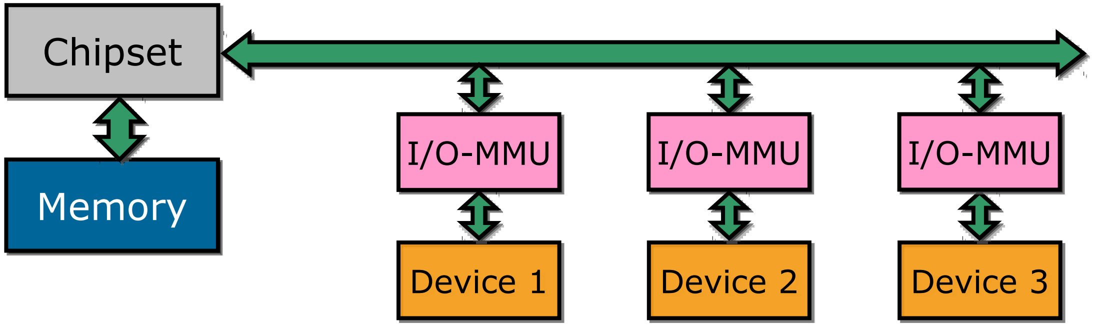
- Like traditional MMU maps virtual to physical addresses
  - implemented in PCI bridge
  - manages a page table
  - I/O-TLB
- Drivers access buffers through virtual addresses
  - I/O MMU translates accesses
  - restrict access to phys. memory by only mapping certain DMA addresses into driver's address space

I/O MMUs

- Per-Bus IOMMU already in 64bit architectures

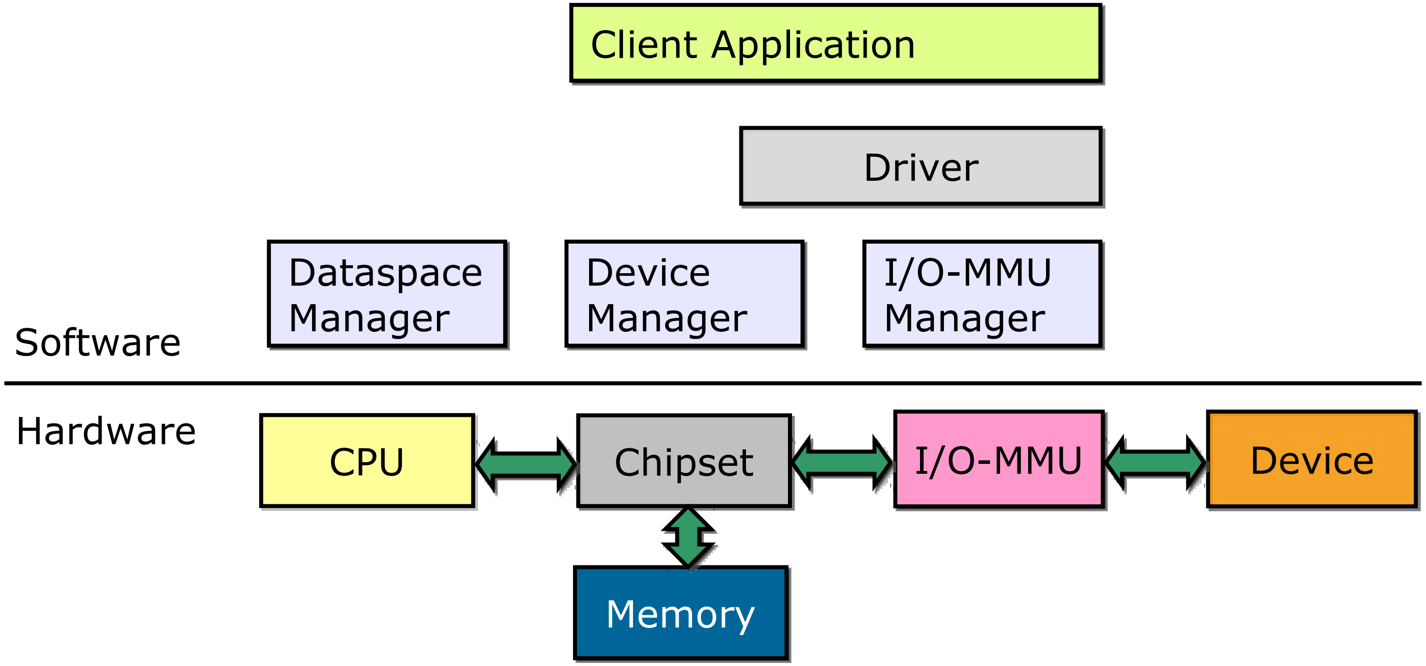


- Per-device IOMMU not available yet



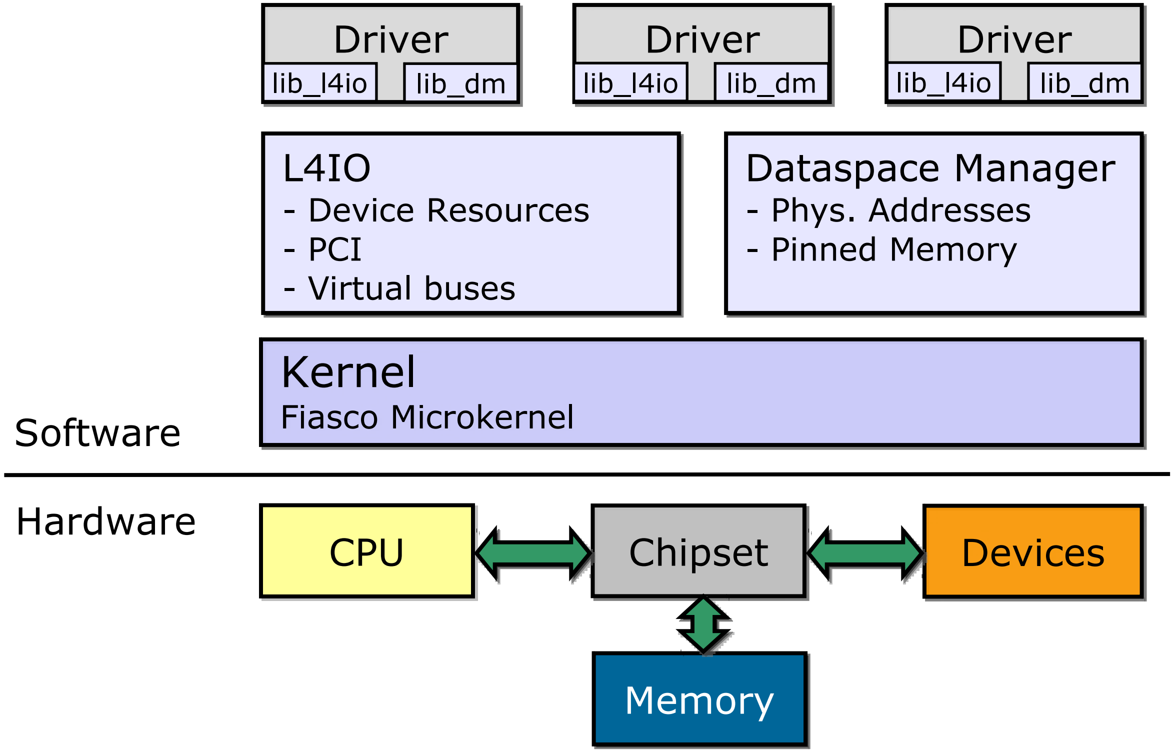
I/O MMU Architecture

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



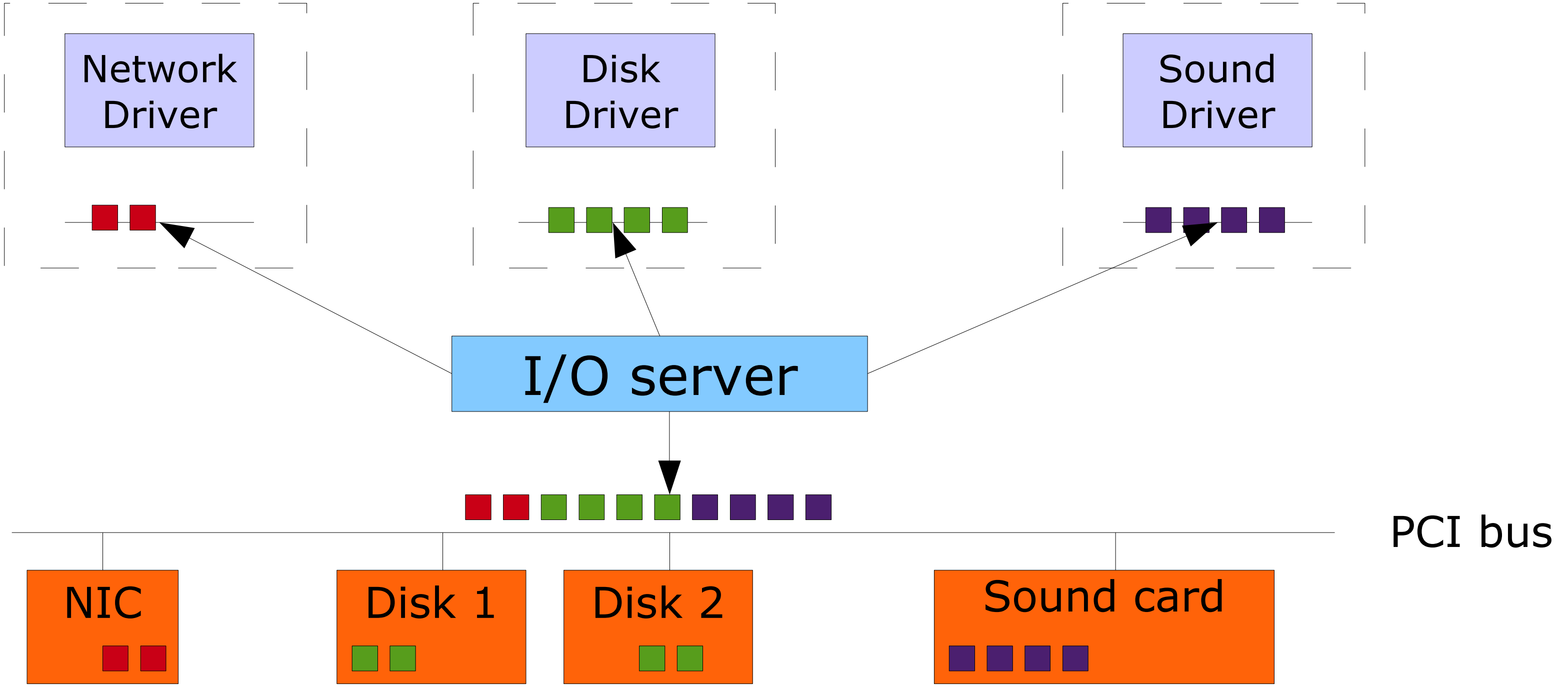
Summary: Driver support in L4

- Interrupts -> Kernel object + IPC
- I/O ports and memory -> flexpage mappings
- User-level resource manager -> L4IO



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.
- L4 hardware support
- Virtual buses for isolating device resources

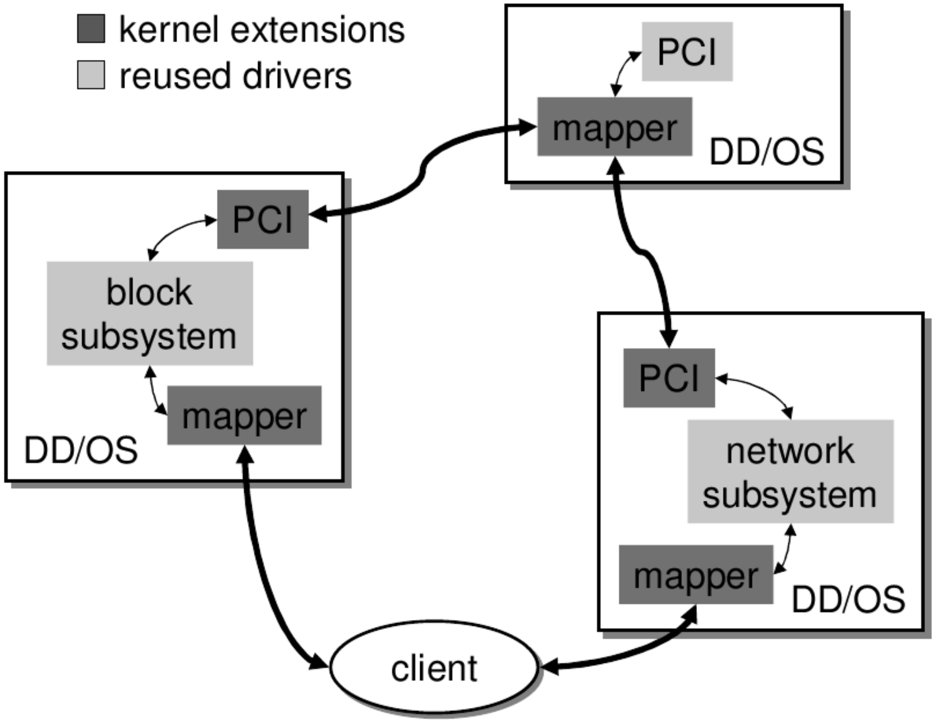
- Next: Implementing device drivers on L4 without doing too much work

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 there are already versions available?
  - Linux, BSD – Open Source
  - Windows – Binary drivers

Reusing legacy device drivers

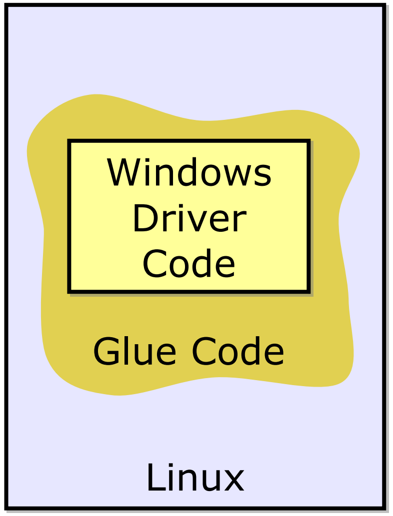
- Exploit virtualization: Device Driver OS



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

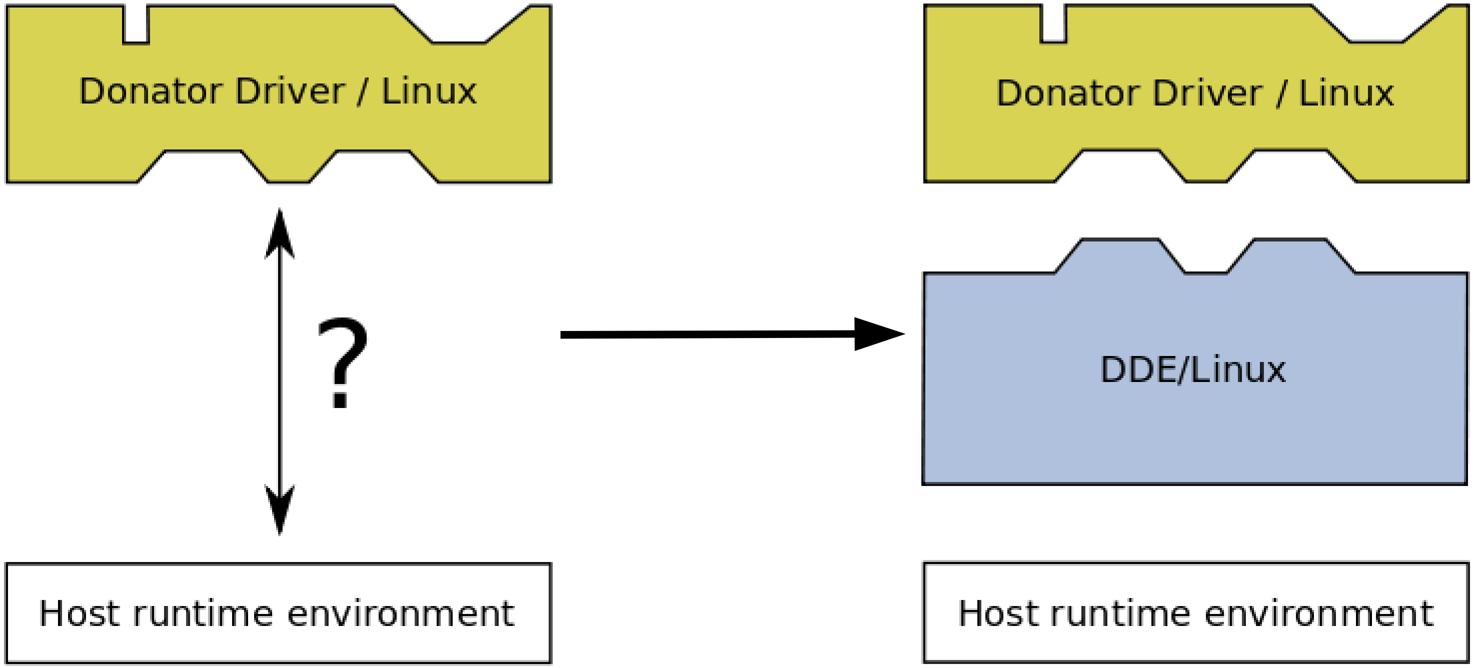
Reusing Legacy Device Drivers

- 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)

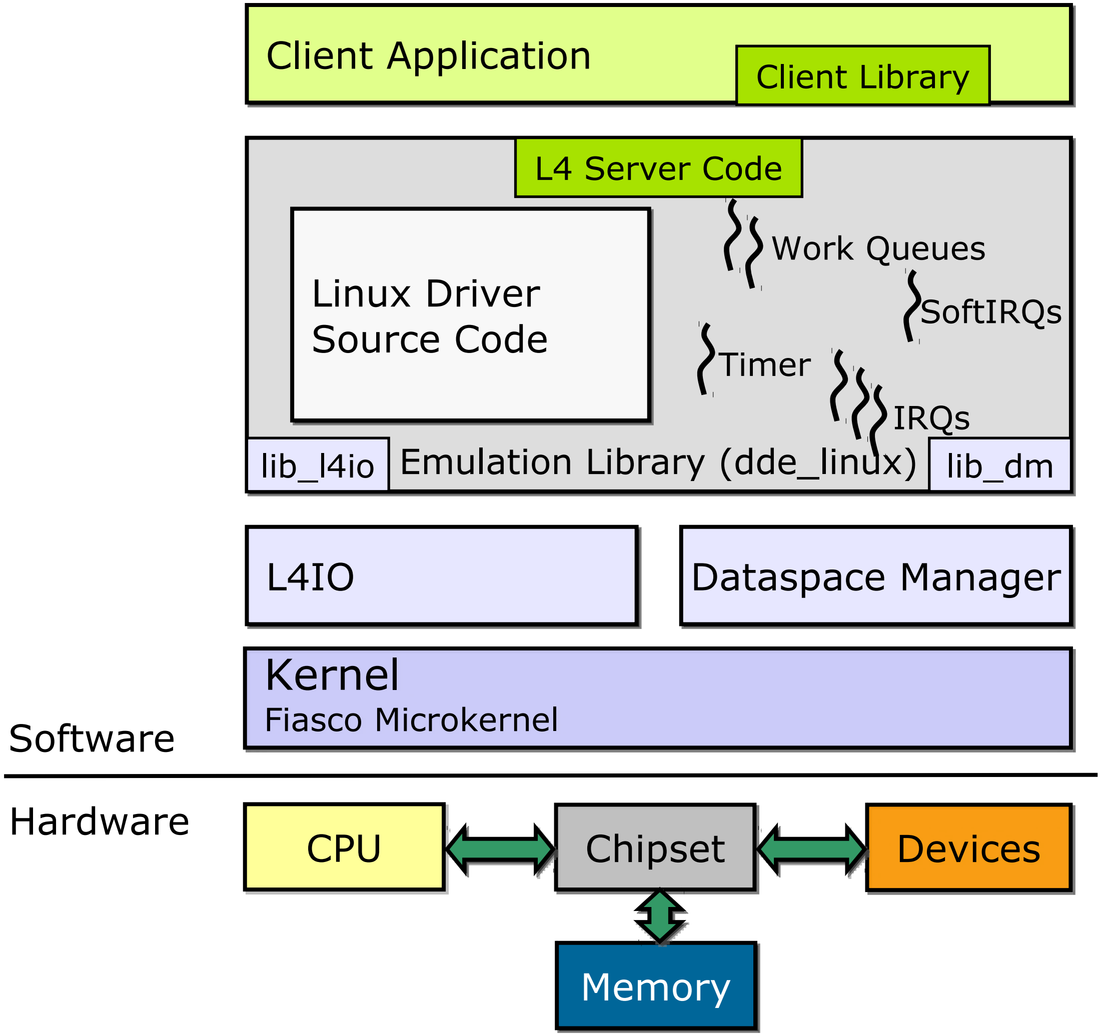
- Generalize the idea: provide a Linux environment to run drivers on L4
  → Device Driver Environment (DDE)



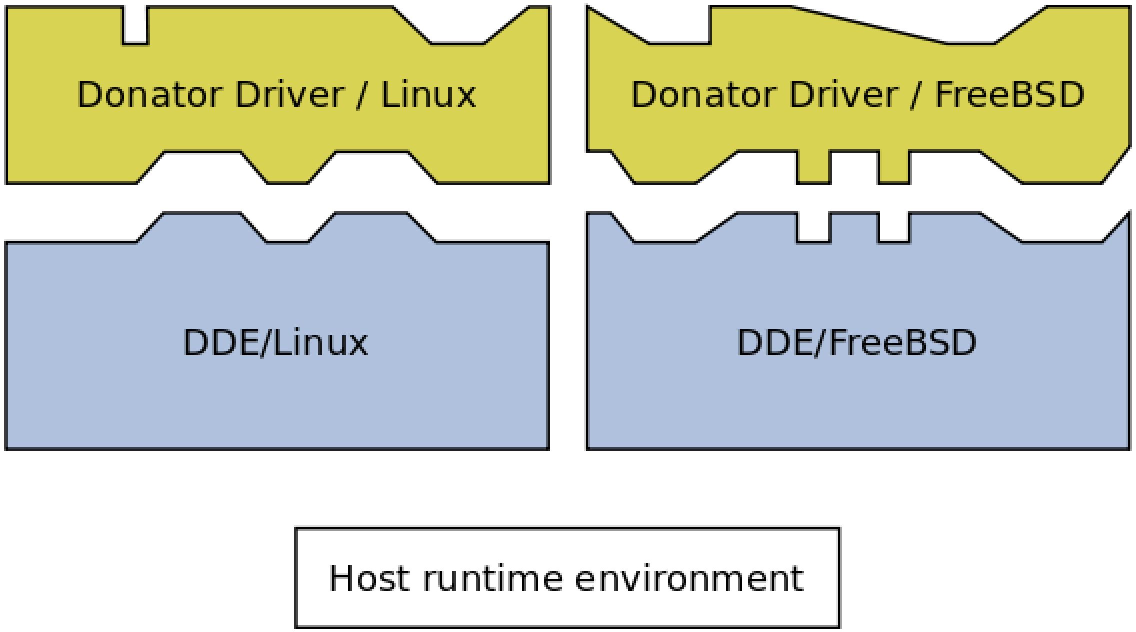
Emulating Linux: DDE/Linux

- Multiple L4 threads provide a Linux environment
  - Workqueues
  - SoftIRQs / Bottom Halves
  - 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



Multiple Donator OSes


Donator Driver / Linux

Donator Driver / FreeBSD

DDE/Linux

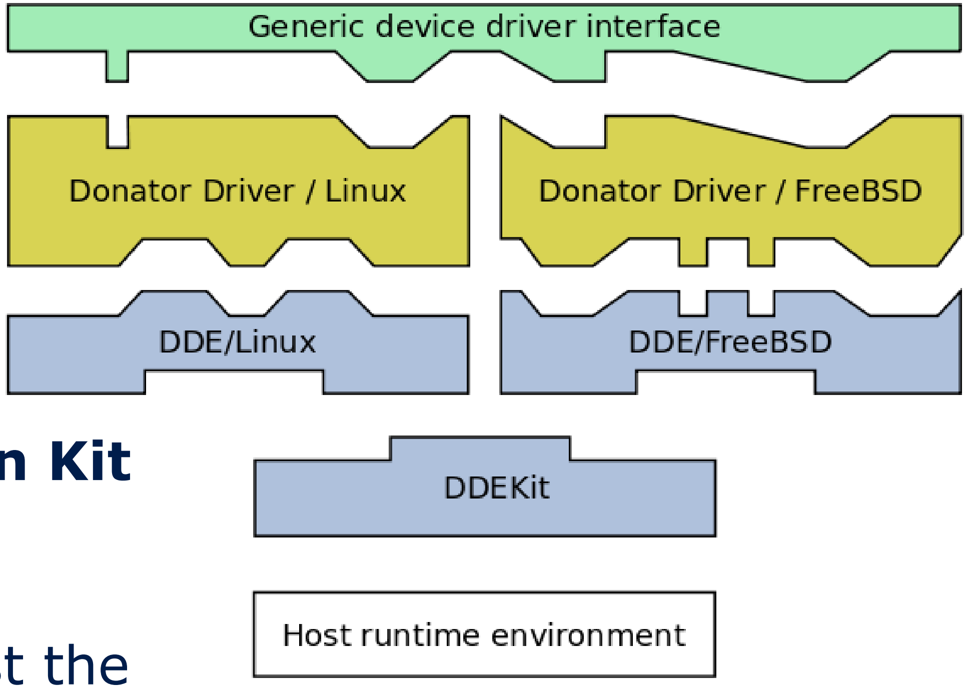
DDE/FreeBSD

Host runtime environment

DDEKit – another abstraction

- Pull common abstractions into dedicated library
  - Threads
  - Synchronization
  - Memory
  - IRQ handling
  - I/O port access

  → **DDE Construction Kit (DDEKit)**

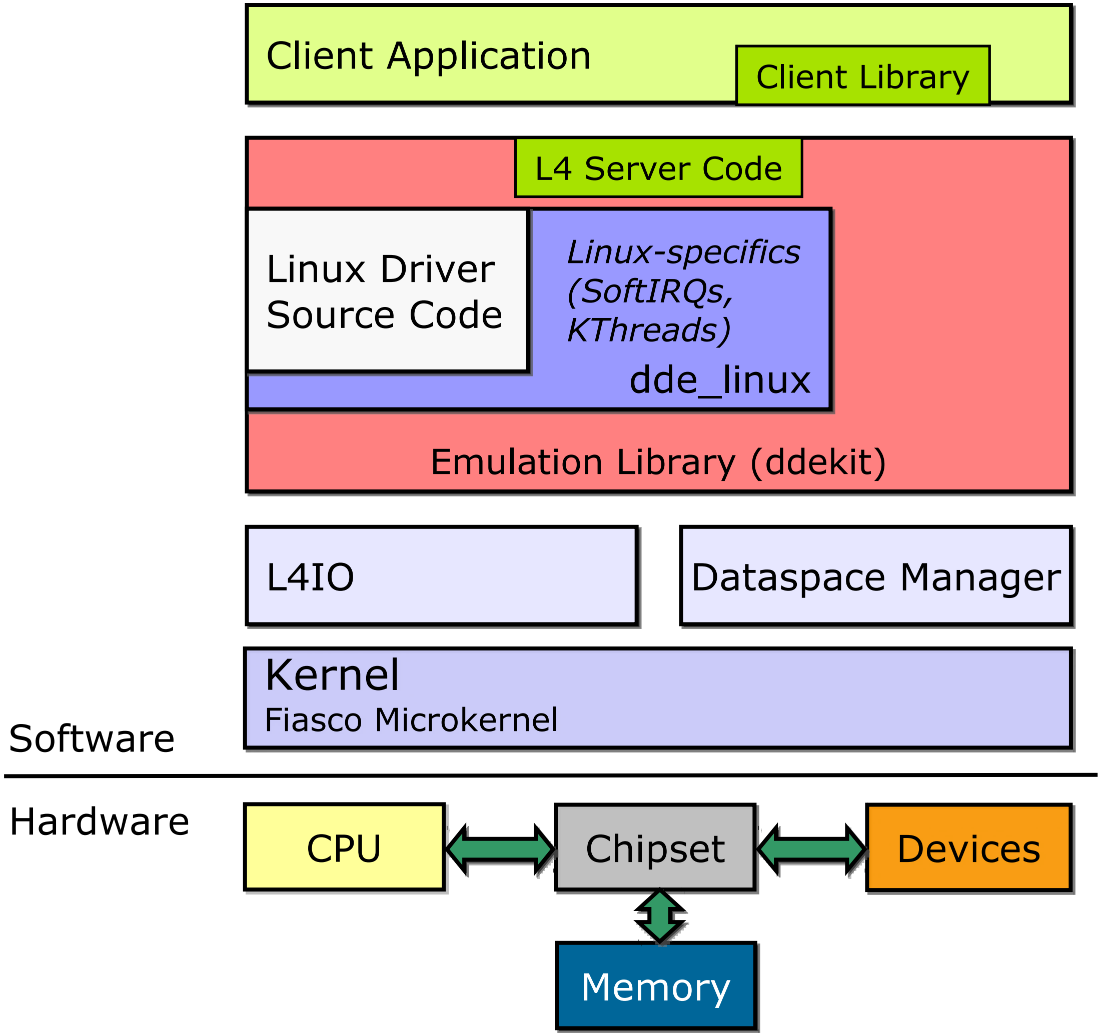


- Implement DDEs against the DDEKit interface

DDEKit (2)

- Implementation overhead for single DDEs gets much smaller
- Performance overhead still reasonable
  - e.g., no visible increase of network latency in user-level ethernet driver
- L4-specific parts (sloccount):
  - standalone DDE Linux 2.4: **~ 3.000 LoC**
  - DDEKit **~ 2.000 LoC**
  - DDEKit-based DDE Linux 2.6: **~ 1.000 LoC**
  - Standalone Linux VM: **> 500.000 LoC**
- Highly customizable: implement DDE base library and support libs (net, disk, sound, ...)

DDEKit (3)



Client Application

Client Library

L4 Server Code

Linux Driver
Source Code

*Linux-specifics
(SoftIRQs,
KThreads)*

dde_linux

Emulation Library (ddekit)

L4IO

Dataspace Manager

Kernel
Fiasco Microkernel

Software

Hardware

CPU

Chipset

Devices

Memory

DDEKit: portability

- Reversing the DDE idea: port DDEKit to host environment → reuse whole Linux support lib
- Has been done for:
  - L4Env, L4Re
  - Genode OS Framework
  - Minix 3
  - GNU/Hurd
- Currently: Linux/UIO on the way

DDE(Kit): Use Cases

- DDELinux2.4
  - IDE Disk Driver
  - Virtual Ethernet Interface
  - USB Webcam
  - TCP/IP Network Stack
  - OSS sound server
- DDELinux 2.6
  - Virtual Ethernet Interface
  - ATA disk driver
  - ALSA sound server
  - USB host controller, web cams, disks, ...
- DDEFreeBSD
  - ATA disk driver

Break

- Device driver support library
  - Reuse donator drivers
  - Split into generic and donator-specific parts
  - Portable on both directions
- Next: Securing device drivers

Securing Drivers: Nooks

- Failure model: transient failure of driver

- Run drivers in lightweight protection domain
  - still ring0
  - switch page table before executing driver code (make kernel data read-only)

- Need to wrap all driver-kernel function calls
  - Track and update duplicate objects

- 22,000 LoC, performance near native

Nooks Shadow Drivers

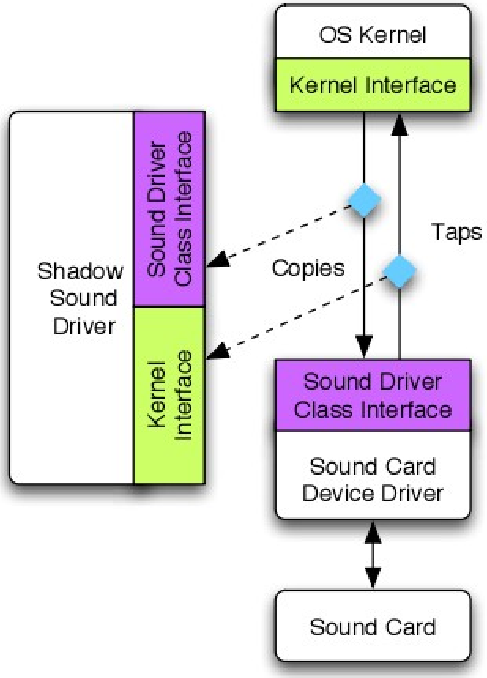


Figure 2: **A sample shadow driver operating in passive mode. Taps inserted between the kernel and sound driver ensure that all communication between the two is passively monitored by the shadow driver.**

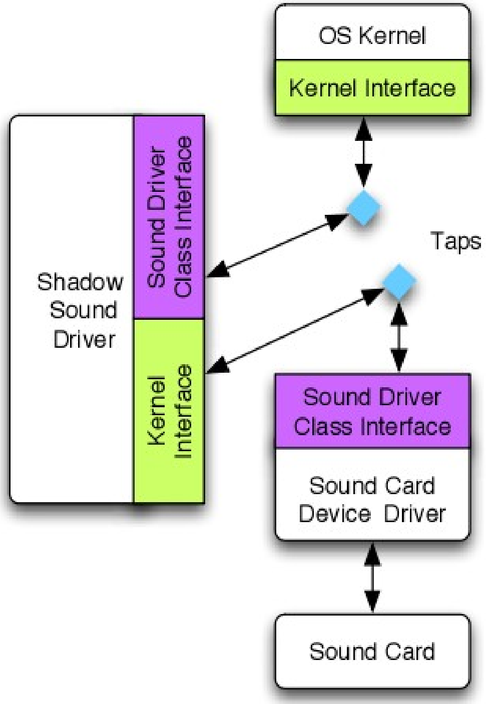


Figure 3: **A sample shadow driver operating in active mode. The taps redirect communication between the kernel and the failed driver directly to the shadow driver.**

Securing Drivers: Dingo

- 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

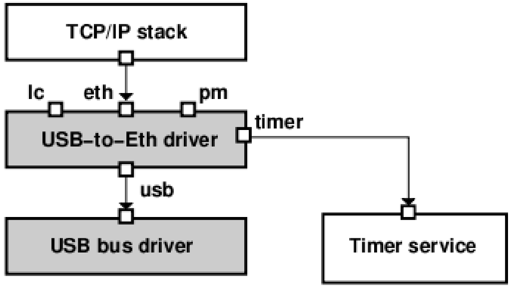
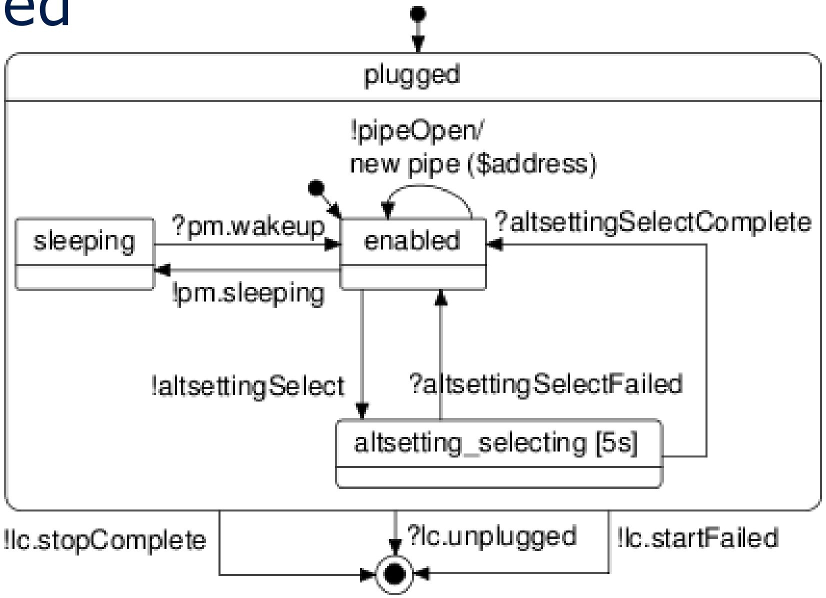


Figure 3. Ports of the USB-to-Ethernet adapter driver.



Dingo (2)

- 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

Various Cool Things ™

- DevIL (OSDI 2000): generate driver from an IDL spec of the device interface

  "...our vision is that Devil specifications either should be written by device vendors or should be widely available aspublic domain libraries..."

- Termite (SOSP 2009): use device driver spec (VHDL) to generate
  - Lets vendors generate drivers on their own

- RevNIC (EuroSys 2010):
  - Obtain I/O trace from existing driver (Windows)
  - Analyse driver binary
  - Generate Linux driver

TECHNISCHE UNIVERSITÄT DRESDEN

Literature

- **Reading on 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
  - Leonid Ryzhyk, Peter Chubb, Ihor Kuz and Gernot Heiser: *"Dingo: Taming device drivers"*, EuroSys 2009
  - 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
- **DDE-related**
  - http://os.inf.tu-dresden.de/papers_ps/helmuth-diplom.pdf
  - http://os.inf.tu-dresden.de/papers_ps/friebel-diplom.pdf
  - http://os.inf.tu-dresden.de/papers_ps/vogt-beleg.pdf

Coming soon

- Today:
  - Paper Reading Exercise:
    Singularity – Rethinking the Software Stack
- **NO LECTURE OR EXERCISE NEXT WEEK**
- Nov 30th:
  - Lecture: Resource Management
  - Practical exercise