



TECHNISCHE
UNIVERSITÄT
DRESDEN

HARDWARE UND GERÄTETREIBER

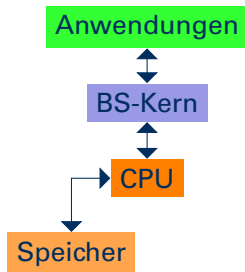
Björn Döbel
Michael Raitza

Dresden, 05.01.2016

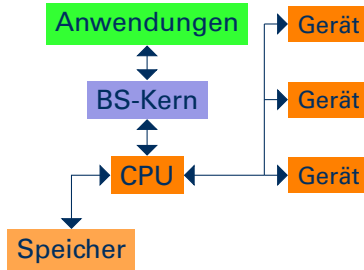
Übersicht

- Kommunikation zwischen Hardware und CPU
 - Interrupts
 - I/O-Ports
 - I/O-Speicher
 - Busse
- Verwaltung von Geräten
 - Dynamisches Hinzufügen/Entfernen
 - Anbindung an das Betriebssystem
- Warum ist das alles so schwer?

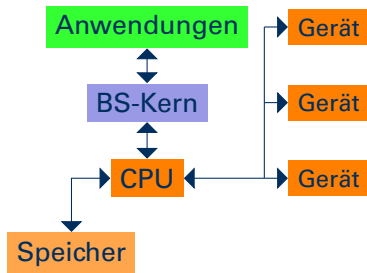
Bausteine



Bausteine



Bausteine

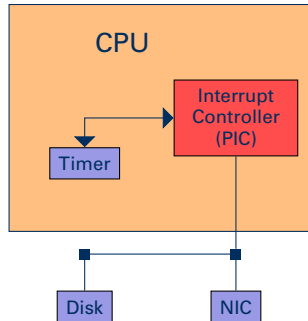


Beispiele für Geräte

- Eingabegeräte (*Human Interface Devices (HID)*)
- Netzwerkkarten (*Network Interface Cards (NIC)*)
- Grafik-Karten
- Speichermedien
- CPU-nahe Geräte (Timer, Interrupt-Controller)

Kommunikation zwischen Gerät und CPU

- Interrupt signalisiert Zustandsänderung des Geräts
- Interrupt-Controller (PIC):
 - Umwandlung von HW-Interrupts in CPU Exceptions
 - Priorisierung, Kombination von Interrupts
 - Maskieren von Interrupts
 - SMP: senden/empfangen von Inter-Prozessor-Interrupts (IPI)



x86: Exceptions

- 256 CPU exceptions
- 0-31 sind reserviert
 - Vordefinierte CPU-Fehler und -Signale
- 32-255 frei verfügbar
- Programmable Interrupt Controller – (A)PIC:
 - Local Vector Table (LVT) bildet HW-Interrupts auf CPU Exceptions (> 31) ab
 - Mehrere Interrupts können die selbe Exception auslösen (Interrupt Sharing)

Exception	Signal
0	Division durch 0
2	Nicht-maskierbarer Interrupt (NMI)
3	Debug-Interrupt
6	Undefinierte Instruktion
13	Schutzfehler (Protection Fault)
14	Seitenfehler (Page Fault)

Behandeln von Exceptions

- CPU: Interrupt Descriptor Table (IDT)
 - Einsprung-Adresse für Handler-Funktion
 - Tabelle im Hauptspeicher
 - Adresse im IDT Base Register
- Interrupts / Exceptions unterbrechen die Ausführung der aktuellen CPU an beliebigen Zeitpunkten
 - Rettung des aktuellen CPU-Zustands nötig
 - Teil des Zustands wird auf den Stack gelegt
 - Anderer Zustand in Verantwortung des Entwicklers
 - Rückkehr und Wiederaufsetzen mit vorherigen Informationen: `iret`

Ein Blick ins x86-Manual

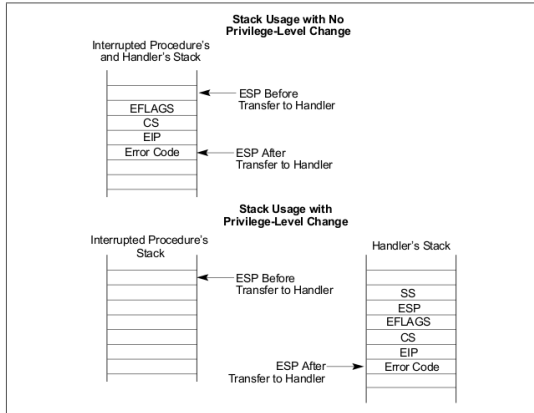


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Effizientes Interrupt-Handling

- Problem Teil 1: Interrupts können jederzeit auftreten
 - Insbesondere während einer zeitkritischen Operation
 - Z. B. Bildaufbau, Umrechnen von Audiodaten, ...
- Problem Teil 2: Behandlung selbst kann aufwendig sein
 - Z. B. Netzwerkpaket gegen aufwendige Filterregeln prüfen
- Lösung: Geteiltes Interrupt-Handling
 - System bleibt reaktionsschnell
 - System kann besser mit Interrupt Bursts umgehen
 - **Trotzdem:** Starke Zusicherung der Korrektheit von I/O-Abhängigkeiten

Top / Bottom Half Interrupt-Handling

a) Top Half:

- Direkter Einsprungpunkt aus der IDT als Interrupt Service Routine (ISR)
- Interrupts gesperrt \leadsto kann nicht unterbrochen werden
- Sichert nur Gerätezustand, damit Gerät weiterarbeiten kann
- Setzt *Deferred Function* (Soft-Interrupt-Handler) auf, die **Bottom Half**
- Rückkehr mit: `iret`

b) Bottom Half:

- Kann zu einem beliebigen, späteren Zeitpunkt vom Kernel aufgerufen werden
- Ist selbst unterbrechbar
- **Aber:** zwei verschiedene Aufrufe derselben Funktion laufen niemals parallel sondern stets nacheinander

Beispiel: Keyboard-Interrupt 0x1

- Setup während des Bootvorgangs
`request_irq(0x1, my_keyboard_handler);`
- Bei Tastendruck:
IRQ 1 →
APIC →
CPU Exception → BS Kern führt Handler-Funktion aus.
- Keyboard-Treiber weiß nun, dass eine Taste gedrückt wurde. Aber welche?

I/O-Ports

- Alternative Möglichkeit zum Zugriff auf x86-Gerätespeicher („Historisch gewachsen“ TM)
- Spezieller Adressbereich: einmalig 65536 I/O-Port-Bytes
- Nicht Teil des physischen Speichers
- Zugriff mittels spezieller Instruktionen
inb, inw, in
outb, outw, out
- Durchreichen an Anwendungen möglich (IO Privilege Level, IO Bitmap)

I/O-Speicher

- Komplexere Geräte:
eigener Speicher
- Von CPU als Teil des
physischen Speichers
zugegriffen
- Aufteilung des
physischen Speichers
plattformabhängig
- Aufgabe des BIOS
während des
Bootvorgangs

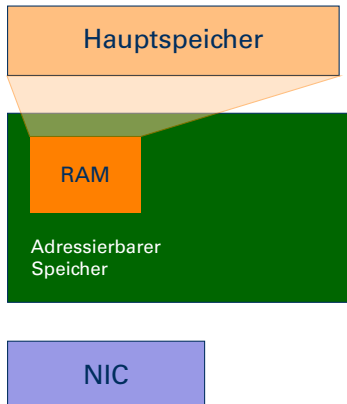
Hauptspeicher

Adressierbarer
Speicher

NIC

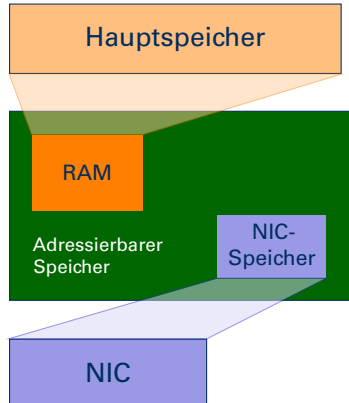
I/O-Speicher

- Komplexere Geräte: eigener Speicher
- Von CPU als Teil des physischen Speichers zugegriffen
- Aufteilung des physischen Speichers plattformabhängig
- Aufgabe des BIOS während des Bootvorgangs



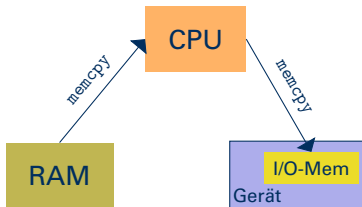
I/O-Speicher

- Komplexere Geräte: eigener Speicher
- Von CPU als Teil des physischen Speichers zugegriffen
- Aufteilung des physischen Speichers plattformabhängig
- Aufgabe des BIOS während des Bootvorgangs



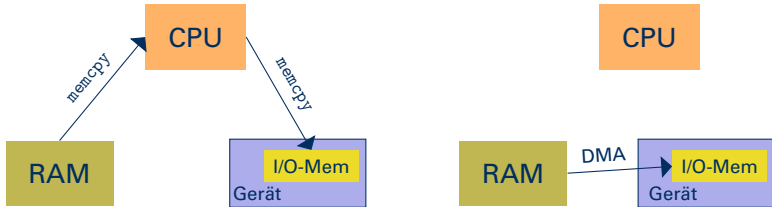
Direct Memory Access

- I/O-Ports: Zugriff auf maximal 4 Byte gleichzeitig
- I/O-Speicher: kann `mempcy()`-Instruktionen (z. B. `rep movs`, SSE-Erweiterungen) benutzen
- Weitere Optimierung: *DMA*



Direct Memory Access

- I/O-Ports: Zugriff auf maximal 4 Byte gleichzeitig
- I/O-Speicher: kann `memcpy()`-Instruktionen (z. B. `rep movs`, SSE-Erweiterungen) benutzen
- Weitere Optimierung: *DMA*



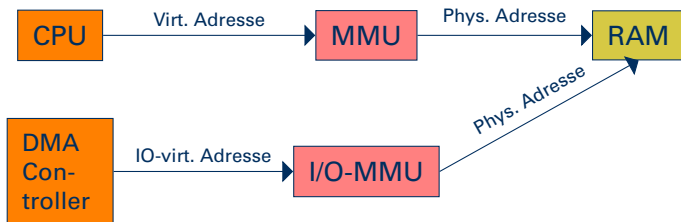
DMA: Sicherheitsproblem

- Physische Adressierung
- Überschreiben beliebiger Speicher-Regionen
- Also: Überschreiben von Kerndaten möglich



DMA: Sicherheitsproblem

- Physische Adressierung
- Überschreiben beliebiger Speicher-Regionen
- Also: Überschreiben von Kerndaten möglich
- Lösung: *I/O-MMU*



Geräte-Zustand

- BS liest und modifiziert Zustand via I/O-Speicher und I/O-Ports → Kernaufgabe eines *Gerätetreibers*
- Geräte-spezifische Aktionen
 - Welchen Zustand gibt es?
 - Wie wird er zugegriffen?
 - Wie führt man Geräte-Aktionen aus?
- Herkunft der Informationen variiert
 - Plattform-weite Definition (z. B. PC-Plattform)
 - Standard f. bestimmte Gerätekategorie (z. B. SATA-Festplatten)
 - Geräte-Spezifikation (vom Hersteller)

Beispiel: AT-Keyboard

- Definiert durch PC-Plattform
- Interrupt: 0x1
- I/O-Port 0x60: Daten-Register (Scancode)
- I/O-Port 0x64:
 - Lesen: Status-Register
 - Schreiben: Kommando-Register
- Wozu sendet man Kommandos ans Keyboard?
 - PC Speaker
 - Reboot

Beispiel: ATA-Festplatten

- ATA-Standard für Klasse von Geräten
- Kontrollregister via I/O-Ports (4 Bereiche für max. 4 parallel betriebene Geräte → [Primary|Secondary] [Master|Slave])
- Mindest-Anforderung: Byteweises Lesen/Schreiben von Daten auf Platte (PIO-Mode)
- Komplexes Protokoll: Liste von Kommandos und möglichen Antworten
- Erweiterungen:
 - Senden mehrerer Bytes
 - Scatter/Gather DMA

Beispiel: Netzwerkkarten

- Kein einheitlicher Standard, üblicherweise:
- Mindestens zwei I/O-Speicher-Bereiche (kb-MB)
 - Eingehende Pakete
 - Ausgehende Pakete
- Mindestens ein Interrupt (auch mehr möglich)
- I/O-Ports oder weitere I/O-Speicherbereiche für Gerätekonfiguration, Statusinformation etc.
- Moderne NICs: eigener Prozessor (TCP Offloading), parallele Paket-Queues

Typischer Ablauf: Netzwerk

- Interrupt trifft ein
- Treiber-IRQ-Handler wird aufgerufen
- Test: Send- oder Receive-Interrupt?
- SEND
 - Bestätigung, dass letztes Paket gesendet
 - Freigabe zugehörigen Speichers
- RECV
 - Test: Paket für mich bestimmt (MAC-Test, Broadcast, Promiscuous Mode)
 - Entfernen des Low-Level-Headers (Ethernet)
 - Weiterreichen an höher liegende Protokoll-Layer (TCP/IP)

Integration in den BS-Kern

Kern definiert Schnittstelle für Geräteklassen Linux
Netzwerk-Treiber-Interface:

```
struct net_device_ops
{
    int (*ndo_init)(struct netdev* dev);

    netdev_tx_t (*ndo_start_xmit)(sk_buff* skb,
                                   struct netdev* dev);

    [... weitere 30 Funktionen ...]
};
```

Treiber-Entwicklung

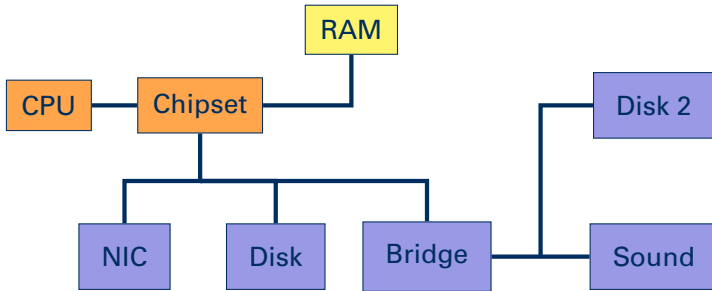
- Oft: Hersteller != BS-Hersteller
- Unterstützung zukünftiger Geräte nötig
- Lösung: nachladbare Treiber
 - Linux: Loadable Kernel Modules
 - Windows, MacOS analog
- Nutzung von Platzhalter-Symbolen
- Auflösung beim Laden durch Module Loader
- Resultat: Treiber muss für jede Kern-Version neu gebaut werden

Gerätetreiber

- Interrupt-Behandlung (Inspektion und Manipulation des Zustands)
- Kapselt geräte-spezifisches Wissen
- Integration in restliche BS-Infrastruktur

Wie findet man Geräte? Wie die zugehörigen Treiber?

Busse: Vernetzung von Geräten



Aufgaben: Identifikation, Adressierung, Weiterleiten von Kommandos/Interrupts

Peripheral Component Interconnect

- Standardisiertes Interface
- Adressierung über Tupel (Bus, Slot, Funktions-ID)
- PCI Configuration Space
 - Zugriff über I/O-Ports
 - Einheitliche Abfrage von Geräteinformationen
 - Liste der Geräte-Ressourcen

31		16 15		0	
Device ID		Vendor ID		00h	
Status		Command		04h	
Class Code			Revision ID		08h
BIST	Header Type	Lat. Timer	Cache Line S.		0Ch
Base Address Registers					10h
					14h
					18h
					1Ch
					20h
					24h
Cardbus CIS Pointer					28h
Subsystem ID		Subsystem Vendor ID			2Ch
Expansion ROM Base Address					30h
Reserved			Cap. Pointer		34h
Reserved					38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line		3Ch

Geräte-Erkennung

- Während des Bootvorgangs
- BS „scannt“ die PCI-Busse
 - Lesen der Geräte-ID
 - Rückgabewert != 0xFFFFFFFF → Gerät vorhanden
- Findend es zugehörigen Treibers
 - Treiber liefert Liste der unterstützten Geräte-IDs
 - BS ruft `init()`-Funktion auf, wenn entsprechendes Gerät gefunden

Universal Serial Bus

- Intel, Microsoft, Compaq, NatSemi seit 1994
- Einstiegspunkt: USB Host Controller
 - typischerweise PCI-Gerät
 - inkompatible Standards (OHCI, UHCI, EHCI, xHCI)
- Geräte-Klassen
 - ein Treiber für alle Geräte ähnlichen Typs
- Bandbreiten-Reservierungen (Echtzeit!)
- Boot-Scan ähnlich zu PCI notwendig

Advanced Configuration and Power Interface

- Microsoft, Intel, Toshiba seit 1997
- Standardisierte Beschreibung verfügbarer Hardware
 - Anzahl CPUs
 - physische Speicher-Bereiche
 - Geräte und Bus-Layout
- Standardisierte Energieverwaltung für CPU und Geräte
- Unabhängig von plattform-spezifischen Treibern oder BIOS
 - ACPI-Tabellen enthalten speziellen Bytecode (AML)
 - AML-Interpreter nötig, um Datenstrukturen zu parsen

Probleme bei Treiber-Entwicklung

- Entwicklung in „unsicherer“ Programmiersprache
 - C → Pointer-Arithmetik, Bit-Operationen etc.
- Kompliziertes Kern-Interface
- Komplizierte Hardware
 - Fehlende HW-Spezifikationen

Eine Anekdote

- 08.08.2008: Bug-Report auf LKML
 - Intel e1000 PCI-X NICs durch Linux 2.6.27 unbrauchbar
 - Überschriebenes NVRAM

Eine Anekdote

- **08.08.2008: Bug-Report auf LKML**
 - Intel e1000 PCI-X NICs durch Linux 2.6.27 unbrauchbar
 - Überschriebenes NVRAM
- **01.10.2008: Quickfix durch Intel**
 - Gerätespeicher an anderer Adresse eingeblendet

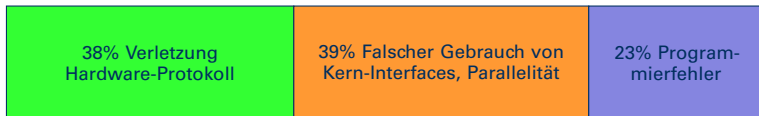
Eine Anekdote

- 08.08.2008: Bug-Report auf LKML
 - Intel e1000 PCI-X NICs durch Linux 2.6.27 unbrauchbar
 - Überschriebenes NVRAM
- 01.10.2008: Quickfix durch Intel
 - Gerätespeicher an anderer Adresse eingeblendet
- 15.10.2008: Grund für den Fehler: Linux Dynamic Function Tracing
 - Komplett unabhängiges Kernel-Feature
 - Fehlerhaftes Verhalten bei `cmpxchg` auf Gerätespeicher

Eine Anekdote

- 08.08.2008: Bug-Report auf LKML
 - Intel e1000 PCI-X NICs durch Linux 2.6.27 unbrauchbar
 - Überschriebenes NVRAM
- 01.10.2008: Quickfix durch Intel
 - Gerätespeicher an anderer Adresse eingeblendet
- 15.10.2008: Grund für den Fehler: Linux Dynamic Function Tracing
 - Komplett unabhängiges Kernel-Feature
 - Fehlerhaftes Verhalten bei `cmpxchg` auf Gerätespeicher
- 02.11.2008: Fehler korrigiert in Linux 2.6.28-rc3

Ein Bisschen Statistik



- L. Rhyzyk et al.: *Dingo – Taming Device Drivers*, 2009
- Analyse von Bug-Reports für eine Reihe von Linux-Treibern
- Andere Quellen: 85% aller Crashes von Windows XP sind auf fehlerhafte Treiber zurück zu führen.

Noch Mehr Statistik

- 2001: A. Chou et al.: *An Empirical Study of Operating System Errors*
- 2011: N. Palix et al.: *Faults in Linux: Ten Years Later*
- Automatisierte Analyse des Linux-Kerns mit statischen Code-Analyse-Tools

Linux: Lines of Code

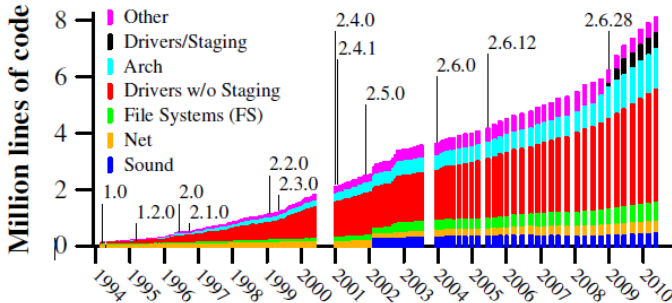
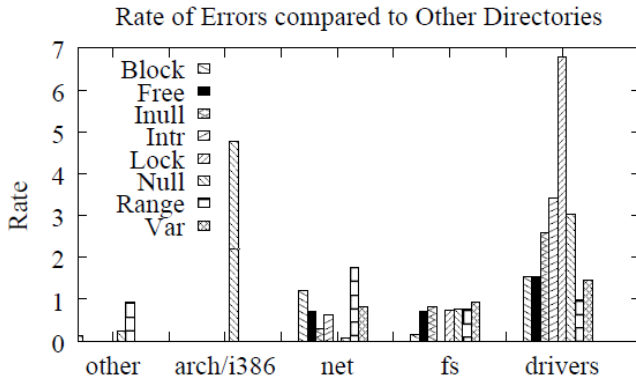
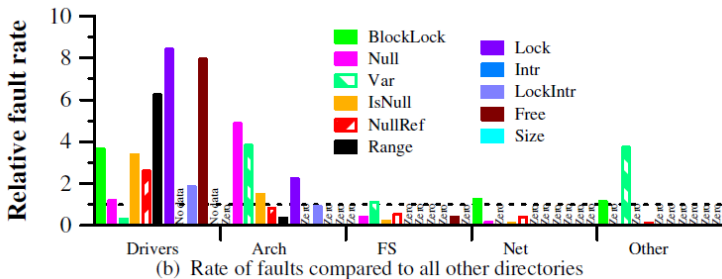


Figure 1. Linux directory sizes (in MLOC)

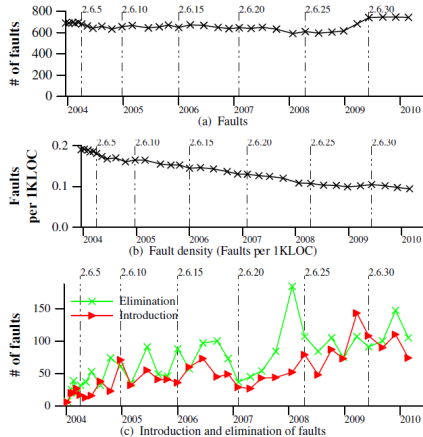
Fehlerrate (2001)



Fehlerrate (2011)



Fehlerentwicklung



Lebensdauer von Fehlern

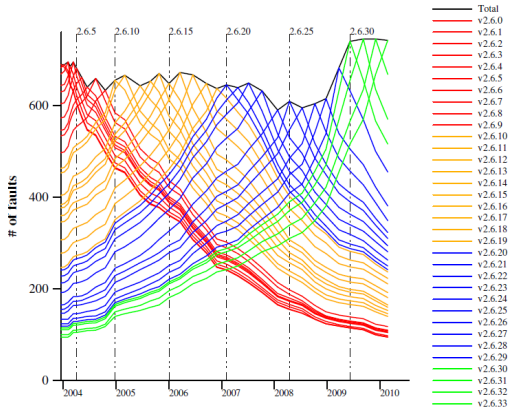


Figure 16. Lifetime of faults across versions

Was tun gegen Treiber-Fehler?

- Testen vs. Hardware-Verfügbarkeit
 - Simulatoren (bedingt geeignet)
- Statische Analyse / Formale Verifikation
 - Windows Certified Drivers
- Wiederverwendung von Treibern anderer Betriebssysteme
 - Linux: Windows WiFi-Treiber
 - Forschungs-BS: Nutzung von Linux-Treibern

Adressraum-Isolation

- Ziel: Verhindern, dass ein kaputter Treiber das ganze System zum Absturz bringt
- *Nooks*: komplizierte Modifikation des Linux-Kerns
- *Mikrokerne*: Treiber als Nutzer-Anwendungen
- *Virtuelle Maschinen*: ein BS pro Treiber