

# Betriebssysteme und Netzwerke

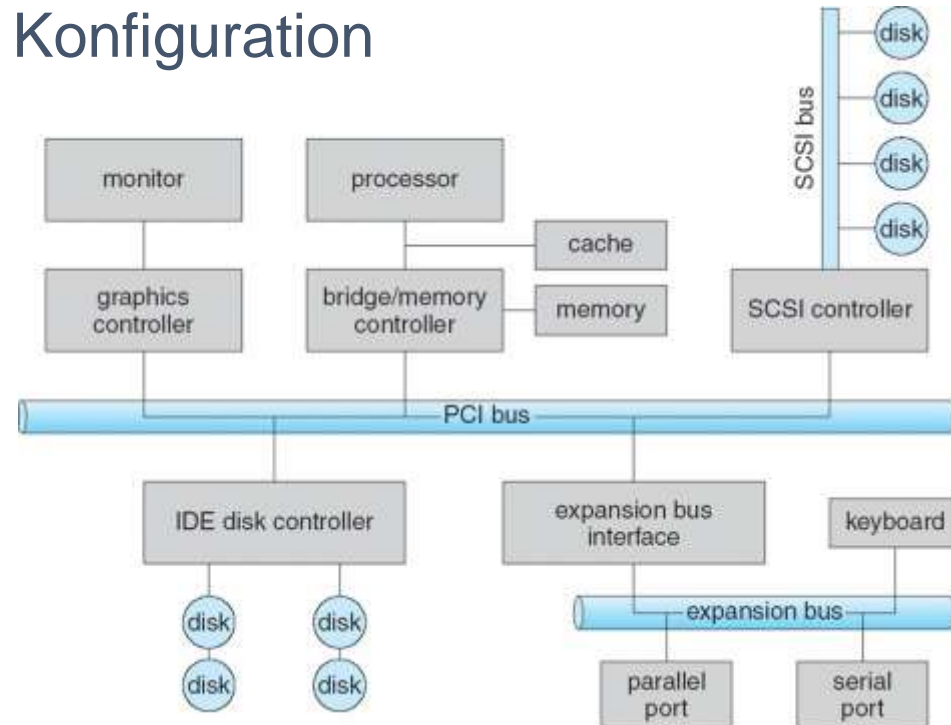
## Vorlesung 17

Artur Andrzejak

# I/O-Grundlagen: Hardware und Zugriff darauf

# I/O-Hardware

- ▶ Die Geräte kommunizieren mit dem Rechner und innerhalb des Rechners durch einen **Bus**
  - ▶ Ein Satz von Kabeln **und** ein Protokoll
  - ▶ Diese unterscheiden sich durch Geschwindigkeit, Typ der Datenübertragung, Art der Konfiguration
- ▶ **PCI Bus (Peripheral Component Inter-connect)** verbindet die schnellen Hauptkomponenten
- ▶ PCI-Nachfolger: PCI-X 2.0, HyperTransport, PCI-Express (**PCIe**) – 3.0 seit 18-11-2010

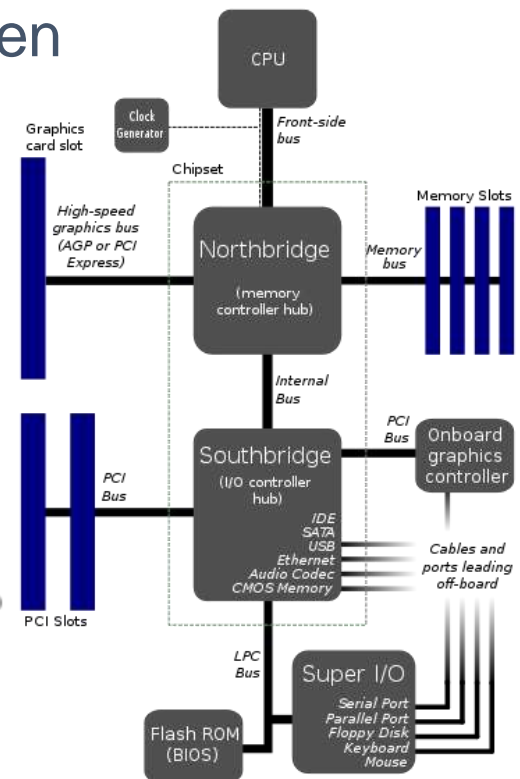
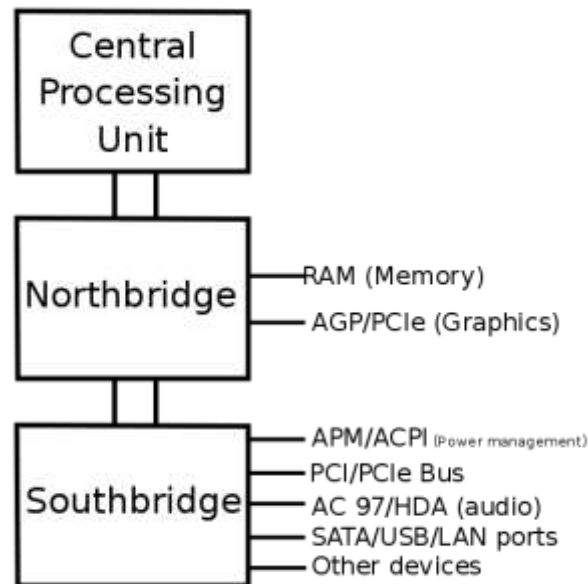


# Bus-Systeme in Modernen PCs

- ▶ Heutige PCs haben spezielle Chips für die Kommunikation zwischen Komponenten
- ▶ **Northbridge**
  - ▶ Für die Kommunikation zwischen CPU, RAM, BIOS ROM und PCI Express bzw. AGP Graphikkarten

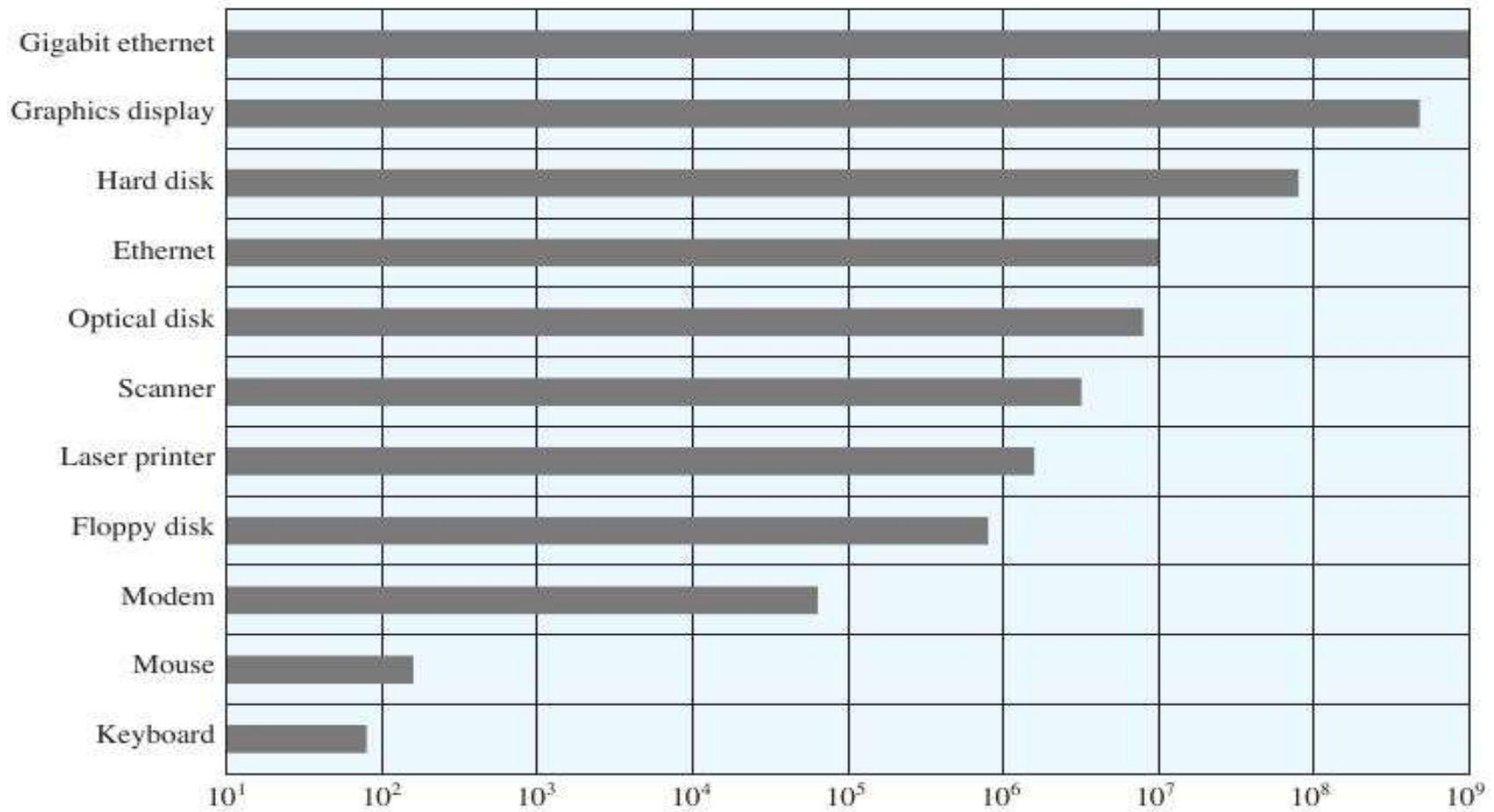
- ▶ **Southbridge**

- ▶ Für „langsamere“ Geräte, z.B. USB, SATA, Audio Chips, Netzwerk, PCI Bus



# Geschwindigkeit

- Die Geschwindigkeit der Kommunikation hängt vom Bus und dem I/O-Gerät ab



Beispiele (z.T. historisch)

Datenrate (Bit/s)

# Controller und Zugriff darauf

---

- ▶ Die elektronische Komponente eines I/O-Gerätes heißt **Controller** (**Gerätesteuereinheit**, **Adapter**)
- ▶ Wie kann die CPU Befehle an den Controller schicken und Daten senden / empfangen?
  - ▶ Controller hat einen oder mehrere Register, die über den Bus erreicht werden können
  - ▶ Gewisse Bits in den Kontrollregistern steuern das Gerät
    - ▶ Siehe Beispiel Floppy-Disk-Controller in VL 1
  - ▶ Andere sind für Daten
- ▶ **Zwei Arten des Zugriffs** durch CPU sind möglich
  - ▶ Spezielle **I/O-Prozessorbefehle**
  - ▶ **Memory-Mapped-I/O**

# I/O-Prozessorbefehle

---

- ▶ Jedem Kontrollregister (eines Geräts) wird eine **I/O-Ausgabeport-Nummer** (**I/O port number**) zugewiesen
- ▶ Ein Befehl wie **IN REG, PORTNR** liest den Wert aus dem Port PORTNR in den CPU-Register **REG**
  - ▶ **in** al,060h ;liest ein Byte vom Port 60h nach **AL**
  - ▶ **in** eax,0A0h ;ein DWord von Port 0A0h nach **EAX**
- ▶ Bei Ports über 255 muss man Register **DX** benutzen
  - ▶ **mov** dx,04F3Ch ;Portnummer (>255) nach **DX**
  - ▶ **in** ax,dx ;Vom Port [**DX**] ein Word nach **AX**

# I/O-Prozessorbefehle - OUT

---

- ▶ Analog schreibt **OUT PORTNR, REG** den Inhalt des Registers **REG** in den Port mit Adresse **PORTNR**
  - ▶ **out** 34h,ax ;Den Wert in **AX** an Port 34h senden
  - ▶ **mov** dx,04573h ;Portnummer 04573h in **DX**
  - ▶ **out** dx,al ;An Port [**DX**] den Inhalt von **AL** senden
- ▶ Achtung: Der Adressraum des Speichers und der I/O-Befehle können gleich sein, trotzdem wird woanders gelesen / geschrieben!
  - ▶ **IN R0,4** liest etwas anderes als **MOV R0,4**



# Memory-Mapped-I/O

- ▶ Hier werden die Kontrollregister (der Controller) in den Arbeitsspeicher „automagisch“ eingebledet
- ▶ Es wird auf diese Register mit normalen **MOV**-Befehlen der CPU zugegriffen
- ▶ Wie?: Wenn CPU die Speicheradresse in einem Bereich ausgibt, wird Mainboard den Zugriff auf I/O-Geräte umleiten

Adressen  
von Kontroll-  
Registern  
einiger Geräte  
bei alten PCs

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

# I/O-Grundlagen: Drei Verfahren der Ein-/Ausgabe

# Verfahren zu Ein-/Ausgabe

---

- ▶ Drei Möglichkeiten, die Ein-/Ausgabe durchzuführen
  - ▶ Unterschiedliche Belastung der CPU
- ▶ **1. Programmierte Ein-/Ausgabe** (prog'ed I/O -**PIO**)
  - ▶ Prozessor sendet Befehle an einen Controller und wartet blockierend auf die Ausführung
- ▶ **2. Interrupt-gesteuerte Ein-/Ausgabe**
  - ▶ Nach dem Senden eines Befehls an einen Controller arbeitet die CPU weiter, und erst wenn Controller Daten hat / fertig ist wird CPU von einem Interrupt unterbrochen
- ▶ **3. Direkter Speicherzugriff (Direct Memory Access)**
  - ▶ Ein HW-Baustein (DMA-Controller) steuert den Austausch von Daten zwischen Speicher und den I/O-Geräten

# Interrupt-gesteuerte Ein-/Ausgabe - Beispiel

---

- ▶ Beim **Systemaufruf** wird der Ausgabe-String in den Adressraum des BS kopiert
  - ▶ **Warten** des Aufrufers (Zustand **waiting**), bis Drucker frei ist
  - ▶ D.h. es wird Prozessorscheduler aufgerufen, der die CPU anderen Prozessen / Threads zuteilt – das „Blockieren“
- ▶ Wenn das I/O-Gerät (Drucker) frei ist, wird ein **Interrupt** erzeugt
  - ▶ Nun wird das nächste Zeichen ausgegeben – **vom BS Kern**
  - ▶ Falls es das letzte Zeichen war, wird der Benutzer-Aufruf „freigegeben“ (Zustand **ready**)
  - ▶ Rückkehr zur Routine, die den Systemaufruf ausgelöst hat (und ggf. blockiert war)

# Interrupts - Details

## ► Prozessoren haben sog.

### **Interrupt-Vektoren**

- Listen von Adressen der I.-Service-Routinen, an die gesprungen wird, abhängig von der Quelle des Interr.

- Spart Zeit, um die Ursache / das Gerät zu finden

Können  
nicht aus-  
geschaltet  
werden

## ► **Interrupt chaining**

- Falls mehr Geräte, als der Vektor decken kann?
- ... Jede Adresse führt zur Code, der eine Liste von I-Erzeuger abarbeitet

Können  
ausgeschaltet  
werden (**maskable**)

## Pentium Interrupt-Vektor

0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	<b>page fault</b>
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19-31	(Intel reserved, do not use)
32-255	maskable interrupts

# Direkter Speicherzugriff (DMA)

---

- ▶ Hatten wir schon in der VL 2, Hardware-Crashkurs
- ▶ Vorteile:
  - ▶ Entlastung der CPU
  - ▶ Geschwindigkeitszuwachs
- ▶ Adressierungsmodi
  - ▶ **Explicit Addressing (Two Cycle Transfer)**
    - ▶ DMA holt ein Datenwort ab und speichert dieses in einem internen Register (wie eine CPU)
    - ▶ Danach überträgt DMA die Daten an die Zielkomponente
  - ▶ Beim **Implicit Addressing (Single Bus Transfer)** entfällt die Zwischenspeicherung in einem Register
    - ▶ Ein einziger Buszyklus nötig, aber nicht für Speicher-zu-Speicher-Übertragungen geeignet

# DMA Chip Beispiel - Intel 8237 ([Link](#))

- ▶ Älterer Chip mit 4x 8-Bit und 4x 16-Bit Kanälen
- ▶ Programmierung erfolgt durch das Schreiben von Werten in die Kontrollregister

Channel	I/O port	Access	Description
Channel 0 (8-bit)	00H	Read/Write	Offset Register
	01H	Read/Write	Block Size Register
	87H	Write only	Page Register
Channel 2 (8-bit)	04H	Read/Write	Offset Register
	05H	Read/Write	Block Size Register
	81H	Write only	Page Register
Channel 4 (16-bit)	C0H	Read/Write	Offset Register
	C2H	Read/Write	Block Size Register
	8FH	Write only	Page Register

- ▶ Später Bestandteil von größeren Chips, wie der Intel [82371](#) (Southbridge), „PCI IDE ISA Xcelerator“

# IT-Sicherheit: Angriffe



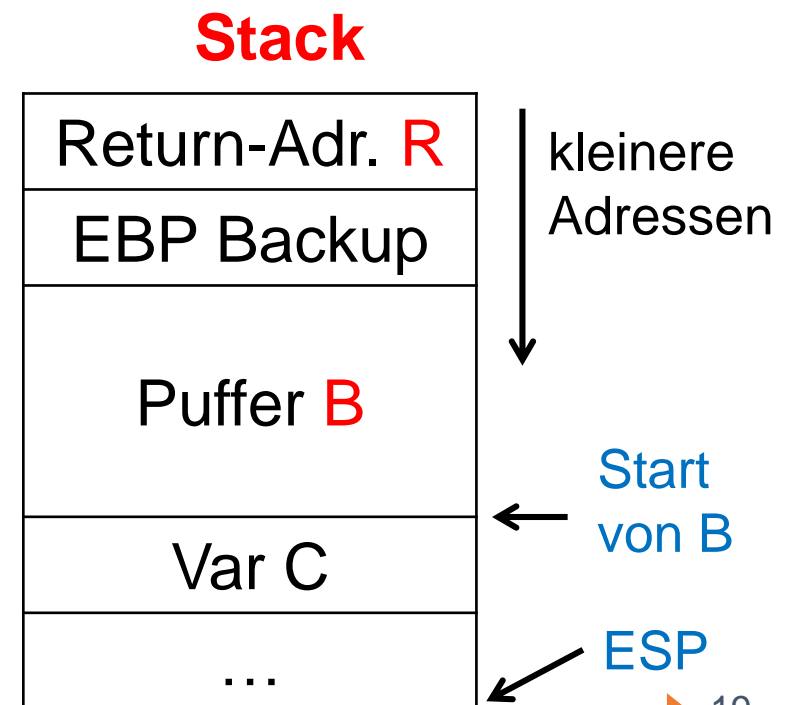
# Einbruch in ein System

---

- ▶ **Hacker/Cracker** sind Personen, die illegal in ein Computersystem einbrechen
  - ▶ Name verwendet auch für Personen, die Kopierschutzmechanismen entfernen (=> „**cracked** software“)
- ▶ Mögliche Arten des Einbruchs?
- ▶ Erraten von Passwörtern
- ▶ Pufferüberlaufangriff
  - ▶ Direkt
  - ▶ Ganzzahlüberlauf
  - ▶ Formatstring-Angriff
- ▶ Code-Injection / SQL-Injection

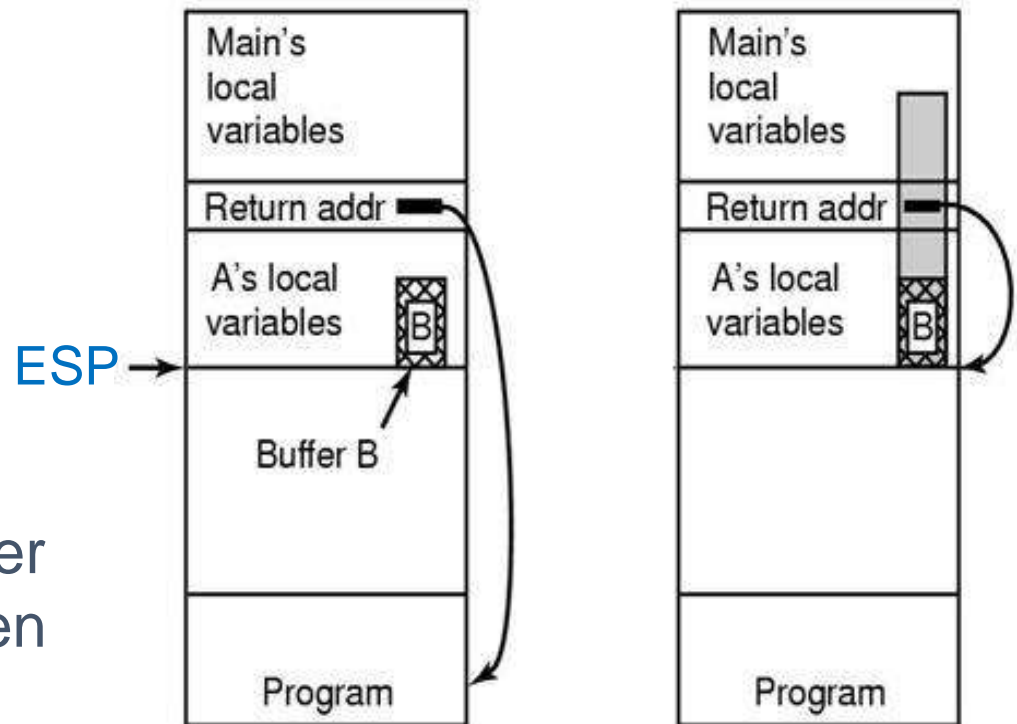
# Pufferüberlaufangriffe

- ▶ Wiederholung: Stack wird u.a. benutzt als...
  - ▶ Speicher für die Rücksprungadresse bei Subroutinen-Aufrufen
  - ▶ Speicher für lokale Daten der Subroutinen
- ▶ Situation innerhalb einer Subroutine A
  - ▶ A benutzt einen Puffer B (z.B. Array), feste Größe n
  - ▶ Daten werden in B von „unten nach oben“ abgelegt
- ▶ Wie könnte ein Einbruch funktionieren?



# Pufferüberlaufangriffe /2

- ▶ Ein **Pufferüberlauf-Angriff** (**buffer overflow attack**) läuft so ab:
  - ▶ Wenn die Eingabe länger als der Puffer B ist, wird die Rücksprungadresse R überschrieben
  - ▶ Eine geschickte Eingabe wird R so überschreiben, dass beim Rücksprung eigener Code ausgeführt wird
  - ▶ Dieser Code wird bevorzugt innerhalb der Eingabe (bzw. B) liegen



# Pufferüberlaufangriffe /3

---

- ▶ Das funktioniert mit vielen Eingaben
  - ▶ Lange Dateinamen, Umgebungsvariablen, ...
- ▶ Es ist nicht leicht, die korrekte Rücksprungadresse bei der Eingabe zu ermitteln
- ▶ Wie könnte ein Cracker vorgehen?
  - ▶ Man gibt ein: Dateinamen mit 10 kB, Gehälter mit 100 Stellen ein usw.
  - ▶ Beim evtl. Absturz sucht man im **Core Dump**, wo die Eingabe gespeichert wurde
  - ▶ Dann muss man noch „ausmessen“, welche Zeichen die Rücksprungadresse überschreiben

# Unterstützung durch **Ganzzahlüberlauf**

---

- ▶ Man füttert ein Programm mit zwei gültigen aber großen Parametern
- ▶ Durch Codeanalyse weiß man, dass diese addiert bzw. multipliziert werden, was zu einem Überlauf führen wird
  - ▶ Das Programm arbeitet dann mit einer negativen oder zu kleinen Zahl, und reserviert einen zu kleinen Speicher
  - ▶ Vorbereitung für Pufferüberlauf
- ▶ Z.B. Angabe der Höhe und Breite einer Bilddatei, die zum Ganzzahlüberlauf führt, bewirkt, dass für die Bilddatei zu wenig Speicher reserviert wird
  - ▶ Die geladene Bilddatei enthält Code, welcher bei Pufferüberlauf ausgeführt wird

# Angriffe durch **Code-Injektion**

- ▶ Hier wird die Situation ausgenutzt, dass ein Eingabestring ein Teil eines ausführbaren Shell / SQL-Codes wird
- ▶ Z.B. ein Kopierprogramm:

```
int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";
    printf ("Please enter name of source file: ?");
    gets (src);          // Eingabe einlesen
    strcat (cmd, src); // Strings aneinanderhängen
    strcat (cmd, " ");
    printf ("Please enter name of destination file: ");
    gets (dst);
    strcat (cmd, dst);
    system (cmd);        // Inhalt von cmd an Shell übergeben
}
```

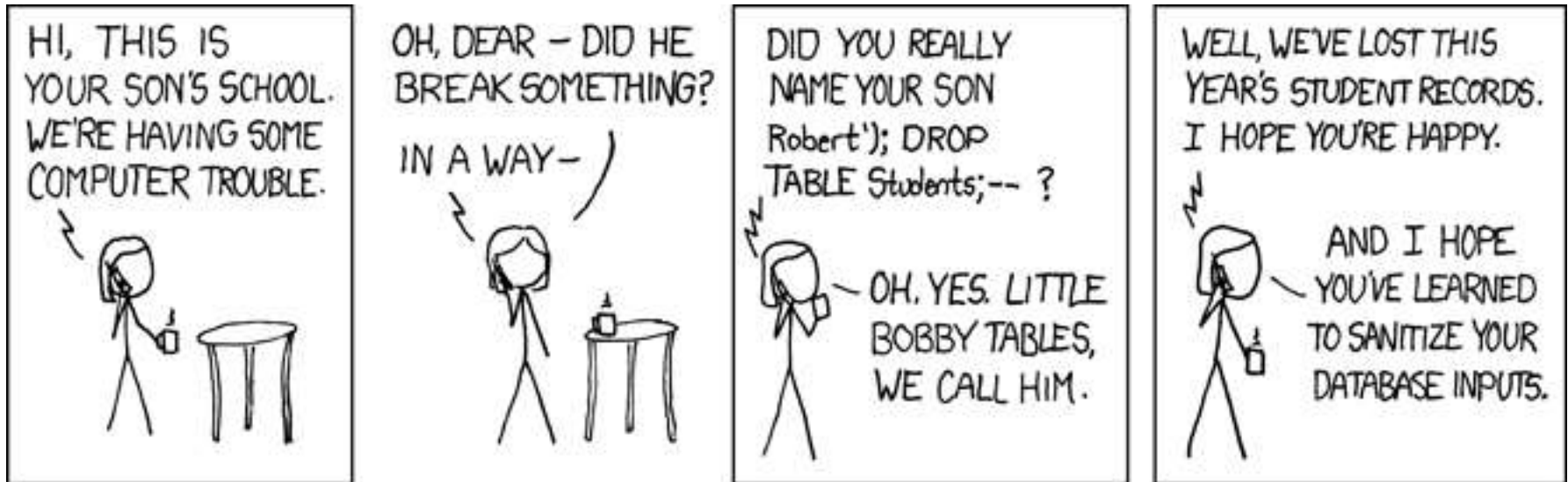
- ▶ Ist alles OK bei der Eingabe „**abc def**“?
- ▶ Was passiert bei der Eingabe „**abc def; rm -rf /**“ ?

# Spezialfall: SQL-Code Injektion

---

- ▶ SQL-Code Injektion ist besonders häufig, siehe [hier](#)
  - ▶ On July 2010, a South American security researcher (...) obtained sensitive user information from popular BitTorrent site “The Pirate Bay”. He gained access to the site's administrative control panel and exploited a SQL injection vulnerability ...
  - ▶ On July 24-26, 2010, attackers from within Japan and China used an SQL injection to gain access to customers' credit card data from Neo Beat (an Osaka-based company) that runs a large online supermarket site ...
  - ▶ On 8th November 2010 the British Royal Navy website was compromised using SQL injection ...
  - ▶ ...

# SQL-Code Injektion /2



- ▶ © xkcd - <http://xkcd.com/327/>
- ▶ A webcomic of romance, sarcasm, math, and language



# Exploits

---

- ▶ Was wir gesehen haben, sind **Exploits**
  - ▶ Techniken / Programme, die Sicherheitslücken bzw. Fehlfunktionen ausnutzen (**to exploit**), um die Kontrolle über ein System zu erlangen, oder es zu manipulieren
  - ▶ Begriff bezeichnet auch die Beschreibung einer Sicherheitslücke im BS oder Anwendung
- ▶ Mehr Techniken im Buch:
  - ▶ G. Hoglund und G. McGraw, *Exploiting Software*, Addison-Wesley, 2004

# IT-Sicherheit – Authentifizierung

# Methoden der Authentifizierung

---

- ▶ **Passwort-Basierte-Authentifizierung**
- ▶ **Public-Key-Authentifizierung** ([Link](#))
  - ▶ Bequem, wird zunehmend in Unix/Linux-Welt verwendet
- ▶ **Biometrische Authentifizierung**
  - ▶ Fingerabdrücke, Gesichtserkennung, Iriserkennung, Handgeometrie (z.B. Messung der Fingerlängen)
- ▶ **Einmalpasswörter**
  - ▶ Implementiert als ein „Passwort-Buch“ oder ein Verfahren von Leslie Lamport (von 1981)

# Methoden der Authentifizierung /2

---

## ▶ **Authentifizierung über Besitz**

- ▶ Speicherchipkarten und Smart Card

## ▶ **2-Schritt-Authentifizierung**

- ▶ Eine Mischung von „... über Besitz“ und von „Einmalpasswörter“
- ▶ Z.B. Google-Konto „Die Bestätigung in zwei Schritten“
- ▶ Nach einem Login mit einem (permanenten) Passwort bekommt man
  - ▶ eine SMS mit einem Code oder
  - ▶ eine Smartphone-Applikation, die einen Code generiert

# Passwort-Basierte Authentifizierung

---

- ▶ In Unix/Linux Systemen werden die Passwörter verschlüsselt gespeichert
  - ▶ Verschlüsselt durch eine Einwegfunktion **f**: leicht zu berechnen, aber schwer umzukehren, z.B. MD5, SHA1
  - ▶ Die Paare (Loginname, **f**(Passwort)) werden in der Datei **/etc/passwd** gespeichert
- ▶ *Wie kann man dann überhaupt authentifizieren?*
- ▶ Bei der Eingabe X eines (möglichen) Passworts wird auch X mit der Funktion **f** verschlüsselt
  - ▶ Dann wird in /etc/passwd bei der Zeile mit dem Loginnamen verglichen: **ist  $f(X) == f(\text{Passwort})$ ?**
  - ▶ Damit kann man die Korrektheit des Passworts prüfen, ohne dieses entschlüsseln zu müssen

# Passwort-Basierte Authentifizierung /2

---

- ▶ Datei /etc/passwd enthält mehr Informationen
  - ▶ user:pw:UserID:GroupID:UserInfo:homeDir:shellCmd  
oracle:x:1021:1020:Oracle user:/data/network/oracle:/bin/bash
- ▶ Deswegen ist diese Datei von allen lesbar
  - ▶ Welche Folgen hat das?
- ▶ Jeder Benutzer des Systems kann die Datei kopieren und sehr schnell mit Listen von potentiellen (verschlüsselten) Passwörtern vergleichen
  - ▶ Insider oder ein Cracker, der als „einfacher“ User eingebrochen ist, konnte somit Passwörter anderer Benutzer schneller erraten

# Passwort-Basierte Authentifizierung /3

- ▶ In neueren BS wird das (verschlüsselte) Passwort in der Datei **/etc/shadow** (oder ähnliche) gespeichert
  - ▶ /etc/passwd hat als Passwort nur ein „x“ oder „\*“ usw.
- ▶ /etc/shadow ist nur vom root (Superuser) lesbar

vivek:\$1\$fnfffc\$pGteyHdicpGOfffXX4ow#5:13064:0:99999:7:::

↓  
Loginname

↓  
f(Passwort)

↓  
3

↓  
4

↓  
5

↓  
6

- ▶ 3: Datum der letzten Passwort-Änderung
- ▶ 4: Min. Anzahl der Tage zwischen Änderungen des Passworts
- ▶ 5: Max. Anzahl der Tage zwischen Änderungen des Passworts
- ▶ 6: # Tage vor forcierter Änderung soll ermahnt werden
- ▶ 7: # Tage nach dem Erlöschen des Passworts bis Konto deaktiviert
- ▶ 8: # Tage nach 01.01.1970 bis das Login ungültig

# Wie sicher sind die Passwörter?

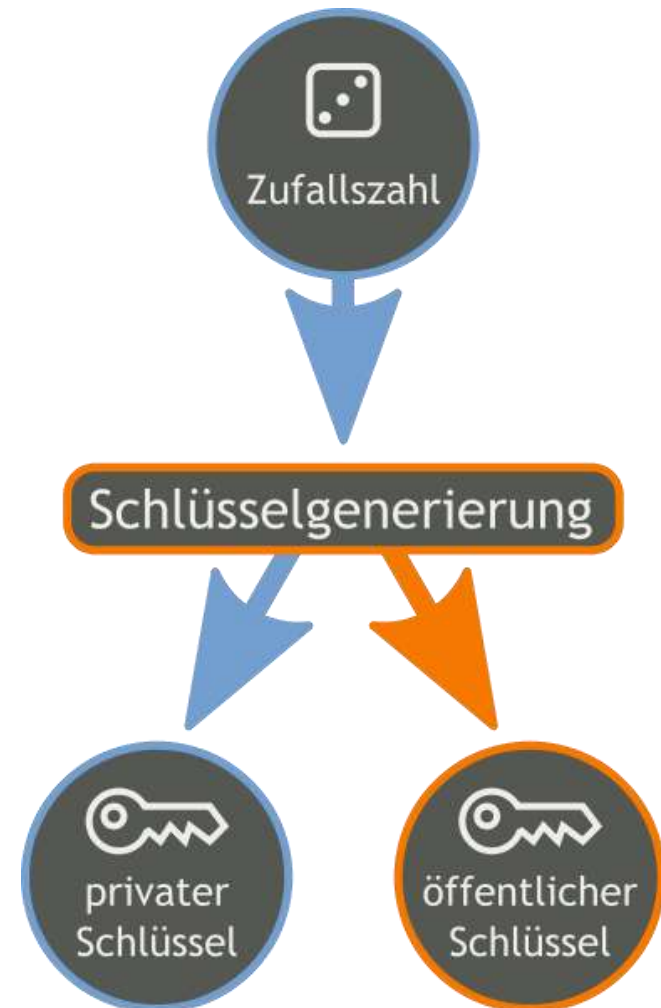
---

- ▶ Studie von 1979: 86% der Passwörter eines Systems waren in einer einfachen Liste zu finden
  - ▶ Aus Nachnamen, Straßennamen, Wörter eines mittelgroßen Wörterbuchs, Autokennzeichen, kurze zufällige Buchstabenketten,...
- ▶ Studie von 1997 (Finanzinstitutionen in London): 82% der Passwörter waren leicht zu erraten
- ▶ Noch ein Problem mit Passwörtern: Sie müssen dem BS vertrauen (insbesondere, wenn Sie gleiches Passwort auf mehreren Systemen nutzen)
  - ▶ Abhilfe: Public-Key-Authentifizierung



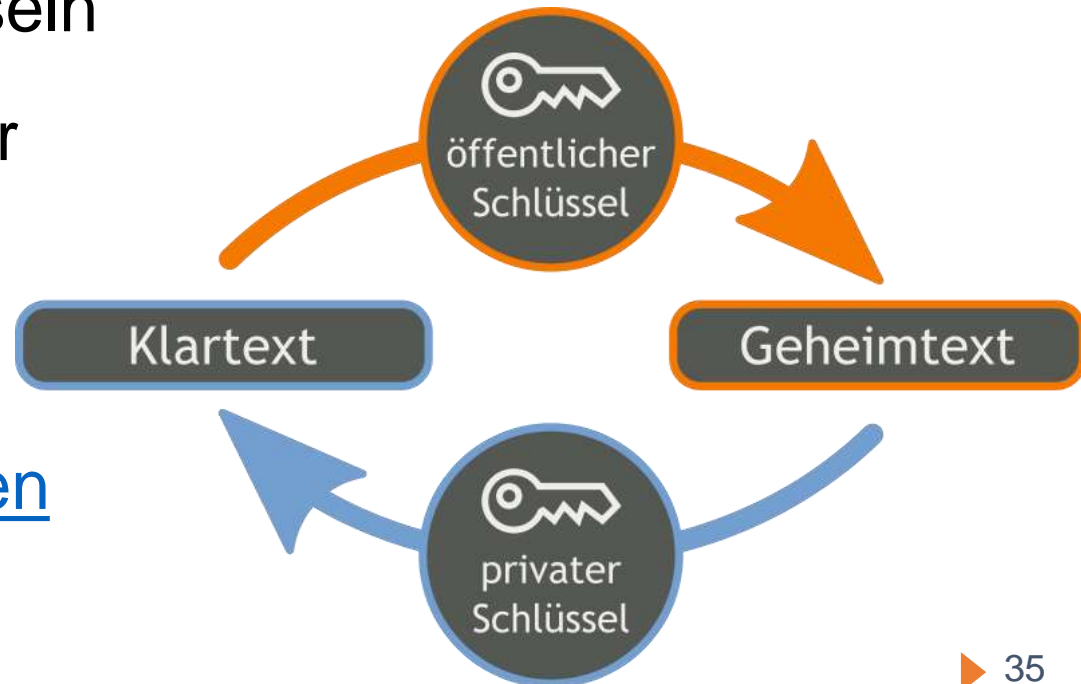
# Public-Key-Authentifizierung ([Link](#))

- ▶ Zunächst wird ein Schlüssel-Paar generiert
- ▶ **Privater Schlüssel (private key)**: wird auf dem eigenen Rechner gespeichert und NIE herausgegeben
  - ▶ Kann optional mit einer beliebigen Kennung (**passphrase**) gesichert werden
- ▶ **Öffentlicher Schlüssel (public key)**: wird jedem zugänglich gemacht
- ▶ Der Ablauf der Authentifizierung ist etwas komplizierter ([Link](#))



# Asymmetrische Kryptosysteme (AK)

- ▶ Die theoretische Grundlage für solche asymmetrische Kryptosysteme sind Falltürfunktionen
  - ▶ Funktionen, die leicht zu berechnen sind, aber ohne ein geheimes String praktisch unmöglich zu invertieren sind
- ▶ Man kann mit Hilfe des öff. Schlüssels leicht eine Nachricht verschlüsseln
- ▶ Aber sie läßt sich nur mit einem privaten Schlüssel entschlüsseln
- ▶ Siehe: RSA-Verfahren



# Öffentliche / private Schlüssel

---

- ▶ Der öffentliche Schlüssel ermöglicht jedem Server:
  - ▶ .. Den Inhaber des privaten Schlüssels zu authentifizieren
  - ▶ .. Die Daten für ihn zu verschlüsseln
  - ▶ .. Die **digitalen Signaturen** des Inhabers des privaten Schlüssels zu prüfen
- ▶ Der private Schlüssel ermöglicht es seinem Inhaber
  - ▶ Sich zu authentisieren
  - ▶ Mit dem öffentlichen Schlüssel verschlüsselte Daten zu entschlüsseln
  - ▶ Digitale Signaturen zu erzeugen

# Vorteile und Nachteile

---

- ▶ Asymmetrische V. schützen das „Geheimnis“ besser, da nur der Benutzer seinen privaten S. kennen muss
  - ▶ Die symmetrischen Verfahren (d.h. beide Seiten kennen den gleichen Schlüssel) erfordern, dass vorher der Schlüssel über einen sicheren Kanal ausgetauscht wird
- ▶ Nachteil von asymmetrischen Verfahren?
- ▶ Sie sind langsam (gegenüber symmetrischen V.)
- ▶ Man setzt **hybride Verfahren** ein
  - ▶ Am Anfang wird ein symmetrischer Schlüssel erzeugt und mit dem asymmetrischen Verfahren ausgetauscht
  - ▶ Die eigentlichen Nachrichten werden mit diesem symmetrischen Schlüssel verschlüsselt

# Betriebssystem „Android“

# Android

---

- ▶ Ein BS und Plattform für mobile Geräte
- ▶ Android wird entwickelt vom [Open Handset Alliance](#) Konsortium unter Leitung von Google
  - ▶ 70+ Mitglieder, u.a. HTC, LG, Motorola, Samsung, T-Mobile
- ▶ Geräte gibt es ab dem 22.11.2008
- ▶ Heute in der Version Android Q ([link](#))
- ▶ Besonderheit: Große Bandbreite an Anwendungen („Apps“)
  - ▶ Feb 2012: ca. 645.000 ([Link](#))
  - ▶ ca. 15.1% davon sind Spiele
  - ▶ Via Google Play (vorher „Google Market“)

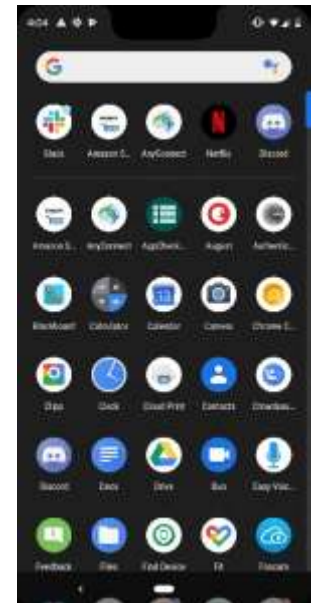


Bild: androidpolice.com,  
<http://bit.ly/2Ro749Q>

# Android – Marktanteile 2015 - 2016

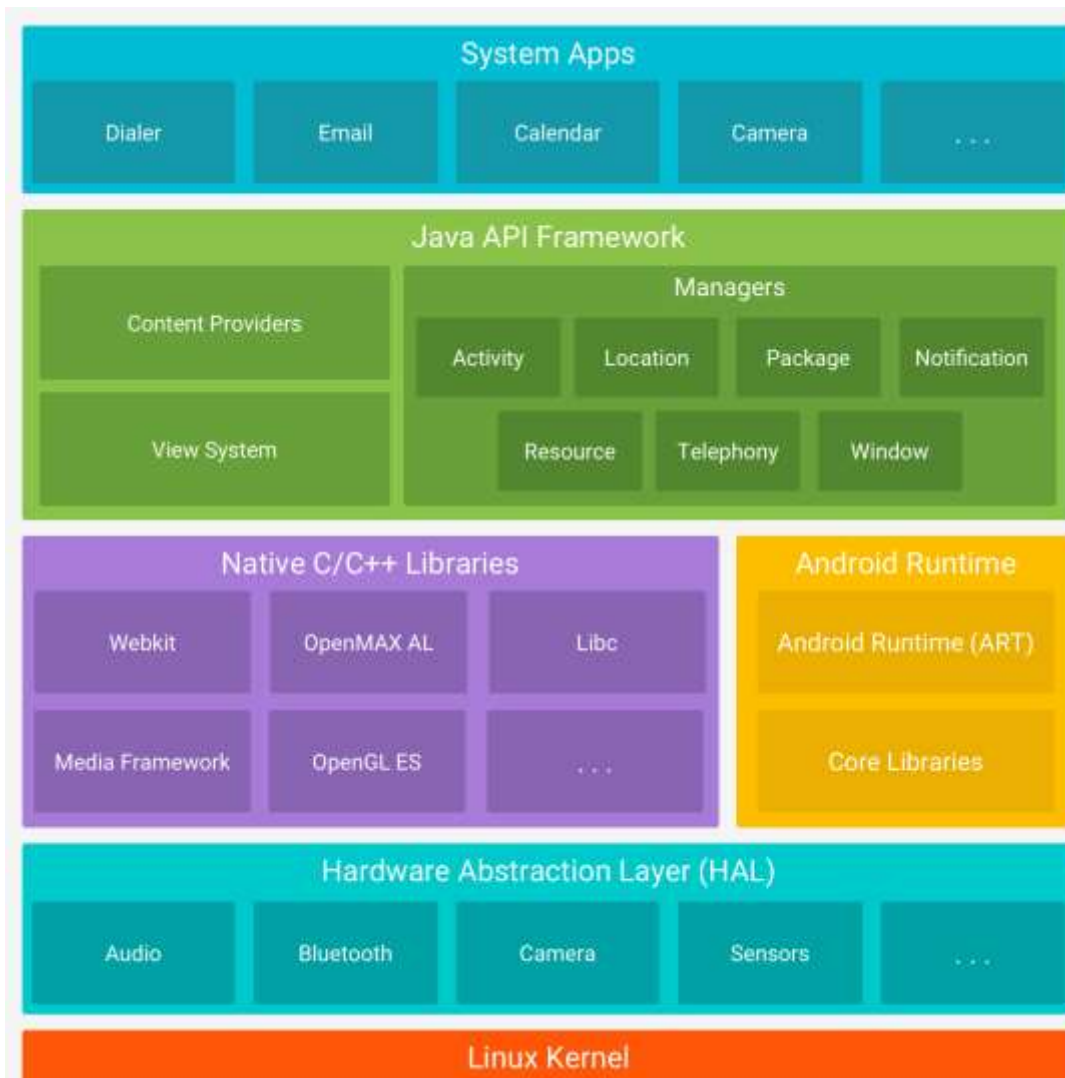
---

- ▶ Marktanteile von mobilen BS:
  - ▶ Android, Apple iOS, BlackBerry OS, Symbian, Windows Mobile/Phone

BS	Marktanteil (12/2015)	Marktanteil (10/2016) ( <a href="#">Quelle</a> )
Android	80,7%	87,5%
Apple iOS	17,7%	12,1%
Win Phone	1,1%	0,3%
Black Berry	0,2%	
Andere	0,2%	

# Android-Architektur

- ▶ Setzt auf einem Linux-Kern auf ([Link](#))





# Anwendungen unter Android

---

- ▶ Anwendungen müssen in Java oder [Kotlin](#) geschrieben sein
- ▶ Viele Bibliotheken sind nativ in C oder C++ erstellt
  - ▶ Z.B. Audio/Video Codecs, [WebKit](#)-Browser, OpenGL
- ▶ Für die Entwicklung benötigt man eine normale Java SDK plus **Android-SDK**
  - ▶ Quelltext wird zunächst in \*.class-Dateien übersetzt und dann für die Android Runtime (ART)-Format
  - ▶ Entwicklungsumgebungen: Android Studio, Eclipse, IntelliJ IDEA

# Architektur von Anwendungen

---

- ▶ Eine Anwendung (App) kann Teile einer anderen App benutzen
  - ▶ Z.B. Eine App hat ein „ImageViewer“ und dieser wird freigegeben, so können andere Apps diesen benutzen
- ▶ Dazu wird eine App in **Komponenten** unterteilt
  - ▶ Das BS kann eine Komponente erzeugen und ausführen
- ▶ Vier Typen von Komponenten ([Link](#)) - Einstiegspunkte einer App (es gibt keine „main()“)
  - ▶ **Activity** - GUI für einen spezifischen Zweck
  - ▶ **Service** - ein „Job“ im Hintergrund
  - ▶ **Broadcast receiver** - reagiert auf „globale“ Nachrichten
  - ▶ **Content provider** - veröffentlicht Daten an andere Apps

# Architektur von Anwendungen - Details

---

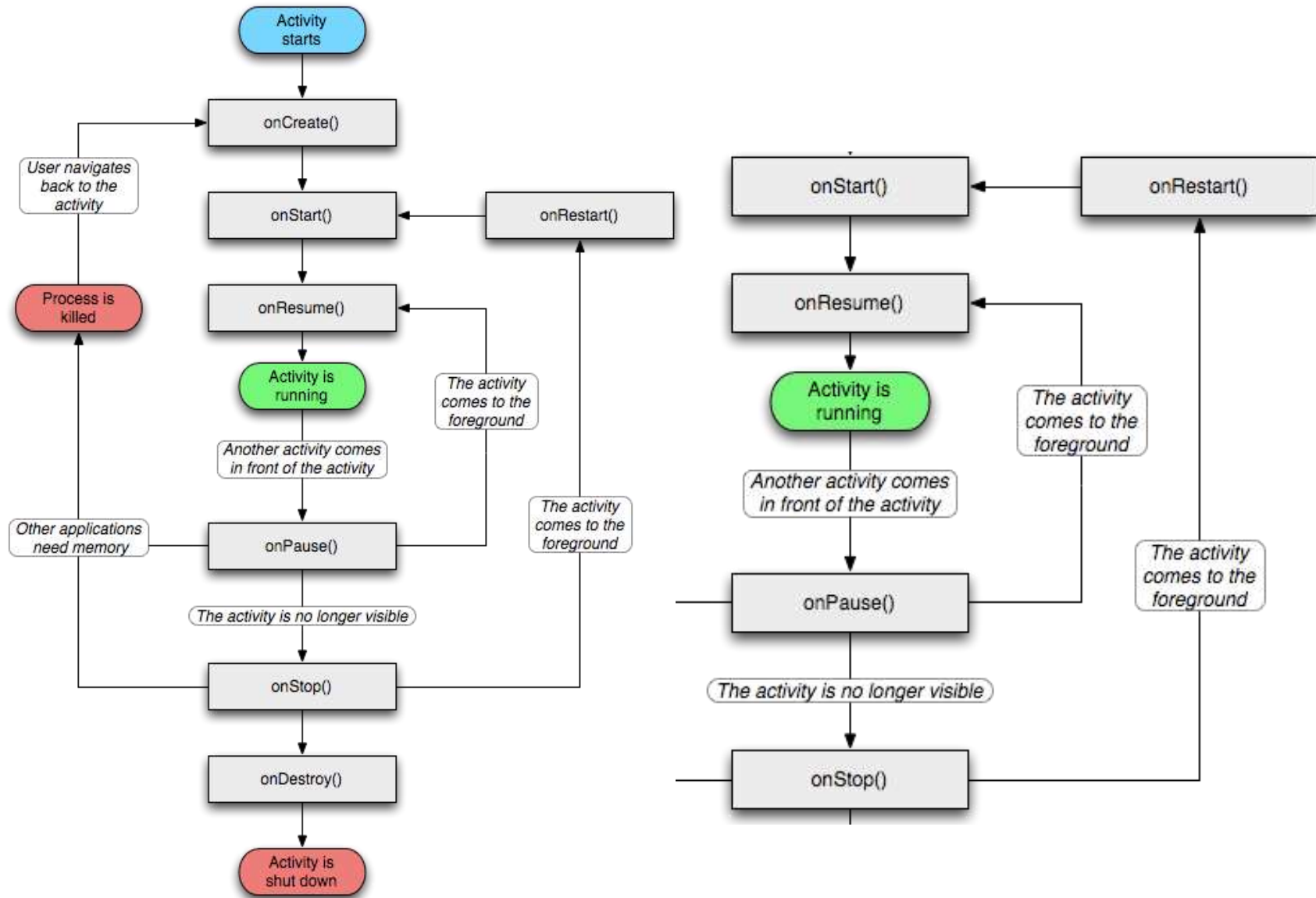
- ▶ **Activity** - meist eine GUI für den ganzen Bildschirm
  - ▶ Eine SMS-Anwendung würde einige Activities haben
    - ▶ Anzeigen einer Liste von Kontakten, an die die Nachricht geht
    - ▶ Ändern von alten Nachrichten; Ändern von Einstellungen, ...
- ▶ **Service** - arbeitet unbegrenzt lange im Hintergrund
  - ▶ Ein MP3/Video-Player würde ein Service benötigen
    - ▶ Zum Abspielen der Musik - auch dann, wenn Activities anderer App aktiv sind
- ▶ **Broadcast Receiver**
  - ▶ Ein „Lauscher“, der die globalen Nachrichten empfängt
  - ▶ Z.B. Batterie ist leer, Zeitzone wurde verändert, ...
- ▶ **Content Provider** - wie ein Datenlieferant
  - ▶ Veröffentlicht Daten aus dem Dateisystem oder SQLite DB

# Lebenszyklus von Komponenten

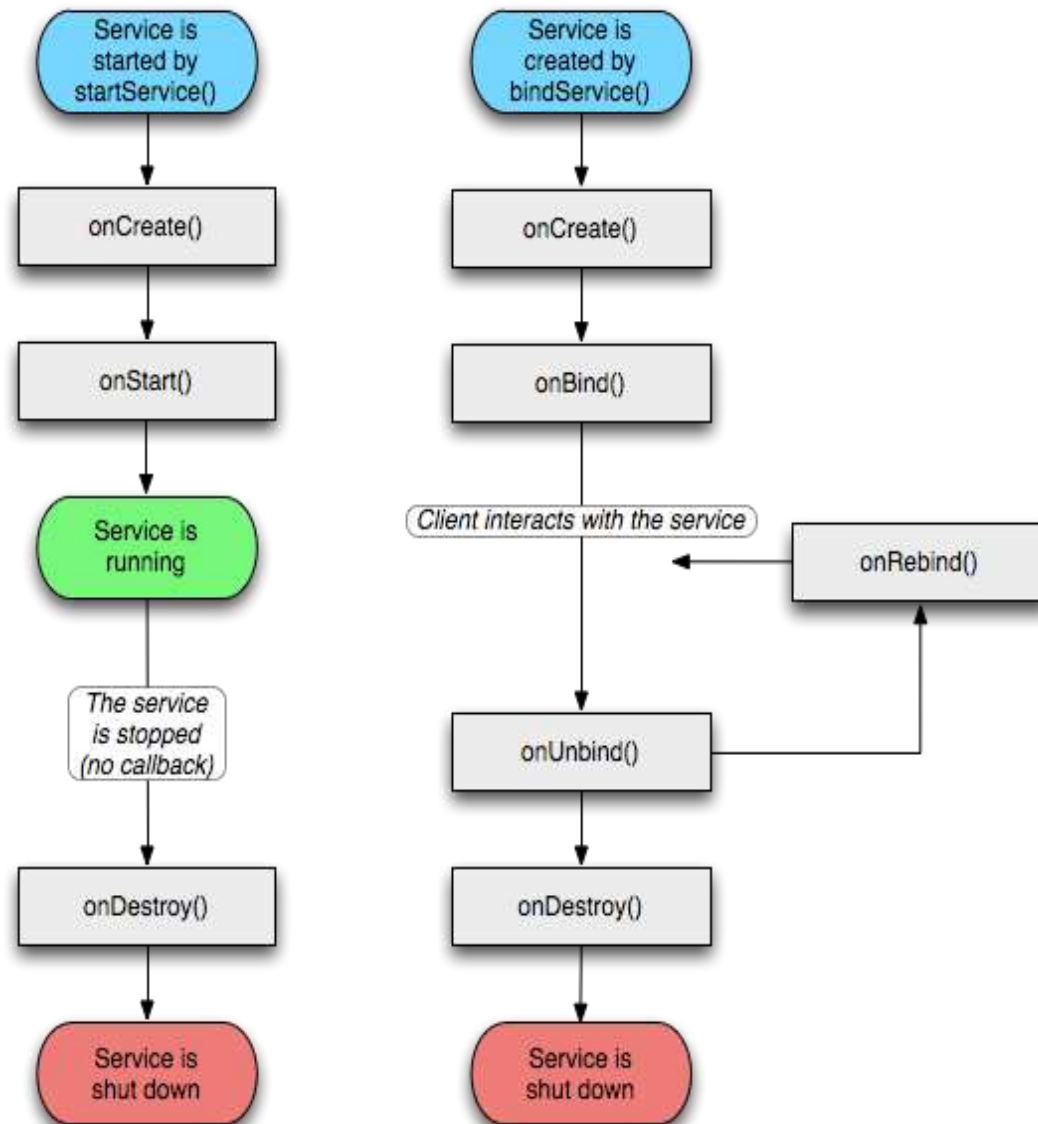
---

- ▶ Android bemüht sich, mit wenig Hauptspeicher auszukommen
  - ▶ Anwendungen, die gerade nicht aktiv sind (z.B. Activity nicht auf dem Bildschirm), werden auf einen Stack gelegt
  - ▶ Wenn der Speicher knapp wird, werden sie terminiert
- ▶ Die Komponent-API stellt Methoden zum Speichern und Wiederherstellen des App-Zustandes bereit
- ▶ Z.B. bei Komponenten vom Typ „Activity“
  - ▶ void onCreate (Bundle savedInstanceState)
  - void onStart()                      void onPause()
  - void onRestart()                    void onStop()
  - void onResume()                    void onDestroy()

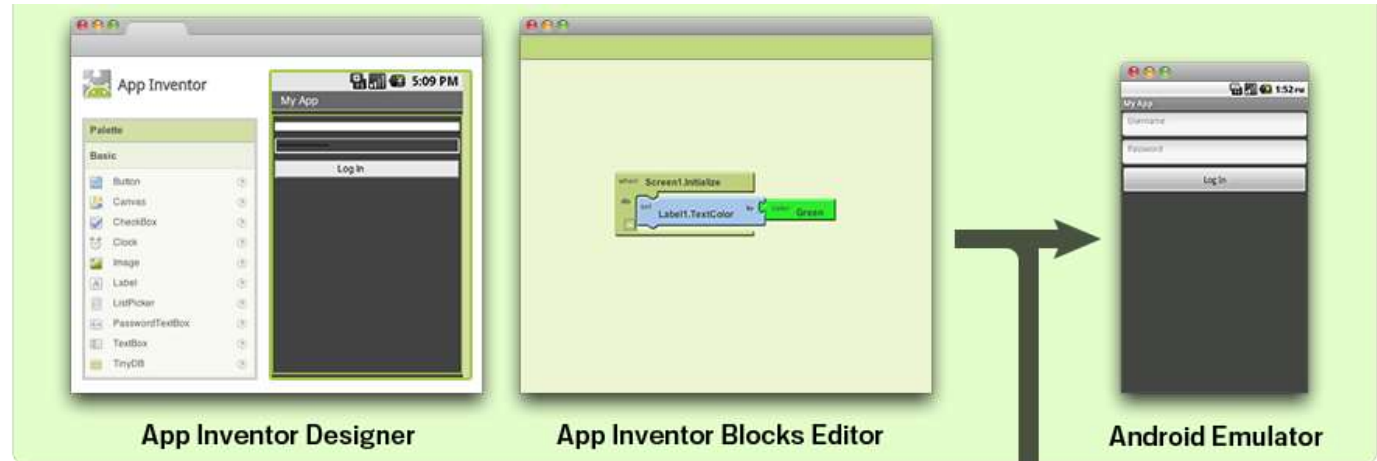
# Lebenszyklus von „Activity“



# Lebenszyklus von „Service“



# App Inventor for Android ([Link](#))



Android Phone

Ab 2012 bei  
MIT, nicht mehr  
bei Google

# Zusammenfassung I/O

---

- ▶ I/O-Grundlagen
- ▶ Controller, Busse, Zugriff auf die Controller
- ▶ Drei Verfahren der I/O-Behandlung
- ▶ Zusätzliche Folien I/O: Effizienz, Schnittstellen zu den I/O-Geräten
- ▶ Quellen (IO): Silberschatz et al. Kapitel 13; Tanenbaum Kapitel 5 (erster Teil davon)



# Zusammenfassung

---

- ▶ IT-Sicherheit: Angriffe
  - ▶ Pufferüberlauf und weitere Exploits
- ▶ IT-Sicherheit: Authentifizierung
  - ▶ Passwörter, Public-Key-Authentifizierung
- ▶ BS Android
- ▶ Zusatzfolien und IT-Sicherheit und Android
- ▶ Quellen (IT-Sicherheit):
  - ▶ Quellen: Tanenbaum Kapitel 9; Silberschatz et al. Kapitel 14 und 15; Wikipedia


Danke.

# I/O-Grundlagen: Drei Verfahren der Ein-/Ausgabe

# Programmierte Ein-/Ausgabe

- ▶ Controller hat ein **busy-Bit** in einem Kontrollregister
  - ▶ C. setzt es, wenn er arbeitet, und löscht es, wenn fertig
- ▶ CPU signalisiert einen neuen Controller-Befehl / neue Daten durch das Setzen des **command-ready-Bits** in einem Befehlsregister des Controllers
- ▶ Bevor CPU das machen kann, überprüft sie wiederholt das busy-Bit, bis es nicht mehr gesetzt ist
  - ▶ Das wiederholte Lesen führt zum Warten (siehe **Spinlocks**)

```
copy_from_user (buffer, p, count);  
for (i = 0; i < count; i++) {  
    while (*printer_status_reg != READY)  
        ;  
    *printer_data_register = p[i];  
}; return to user();
```



Man nennt das  
Verfahren  
**polling** oder  
**busy-waiting**

# PIO-Verbesserung: Asynchrones I/O

---

- ▶ Idee (am Beispiel Drucker-Ausgabe)
  - ▶ Routine **printstring**(s) (Aufruf vom Benutzer-Programm) speichert Daten im Kernel und kehrt zurück
  - ▶ Routine **putnextchar**() wird regelmäßig vom BS aufgerufen:
    - ▶ Wenn das Ausgabe-Gerät frei ist, gibt die Routine nächstes Zeichen aus, sonst kehrt sofort zurück
- ▶ Analog für die Eingabe
  - ▶ Routine **getachar**() wird von Benutzer-Programmen aufgerufen; wartet (blockierend), bis nächstes Zeichen im Puffer P verfügbar ist
  - ▶ Routine **getnextchar**() wird periodisch vom BS aufgerufen: wenn ein neues Zeichen am Gerät vorliegt, schreibt es in P, sonst kehrt sofort zurück

# Nur für Assembler "Freaks": Implementation von **putnextchar**

---

- ▶ **.data**
- ▶ **putq**        db        256
- ▶ tail\_ptr     dd        0
- ▶ head\_ptr    dd        0
- ▶ **.code**
- ▶ **putnextchar:**
- ▶ test DispStatus, 80h
- ▶ jz    putret
- ▶ cmp head\_ptr, tail\_ptr
- ▶ je    putret
- ▶ inc head\_ptr
- ▶ and head\_ptr, 000000ffh
- ▶ mov EAX, head\_ptr
- ▶ mov **BL**, byte **putq**[EAX]
- ▶ mov DisplayData, **EBX**
- ▶ **putret:**
- ▶ ret

Beispiel: Serielle Ausgabe  
auf ein Display

Wo wird es sichergestellt, dass  
der Puffer nicht überläuft?

# Vom Asynchronen I/O zu Interrupts

---

- ▶ Vorteile und Nachteile des Verfahrens?
- ▶ (+) Man wartet praktisch nicht auf das Gerät
- ▶ (-) I/O-Operation nicht sofort, wenn Gerät frei, sondern „erst“ beim periodischen Aufruf
- ▶ Verbesserung?
- ▶ putnextchar bzw. getnextchar wird nicht mehr periodisch, sondern **Interrupt-gesteuert** aufgerufen
  - ▶ Wenn das Gerät fertig ist / etwas empfangen hat, wird ein Interrupt erzeugt

# Zusätzliche Folien: Effizienz der I/O-Operationen



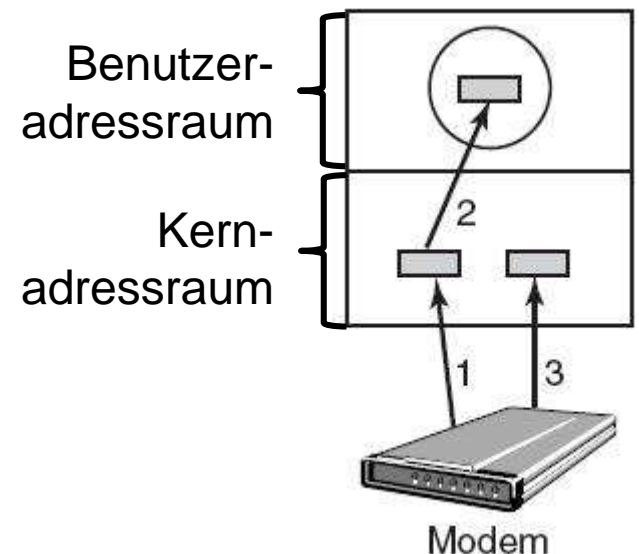
# Pufferung

---

- ▶ Die unterschiedliche Geschwindigkeiten der Komponenten machen die **Pufferung** notwendig
  - ▶ Daten werden durch das BS in einem Speicherblock gesammelt und erst, wenn dieser voll ist, wird der Pufferinhalt an den Benutzerprozess bzw. an ein anderes Gerät weitergegeben
- ▶ Weitere Gründe
  - ▶ **Anpassung** / Übersetzung **der Blockgrößen** zwischen Geräten
    - ▶ Z.B. Übertragung Netzwerk-an-Disk: kleinere Netzwerk-Pakete werden zu 4KB Blöcken zusammengesetzt
  - ▶ Die **Copy Semantics**: Durch das Kopieren in einen Puffer wird sichergestellt, dass nach dem Systemaufruf zum Senden die Daten unverändert bleiben

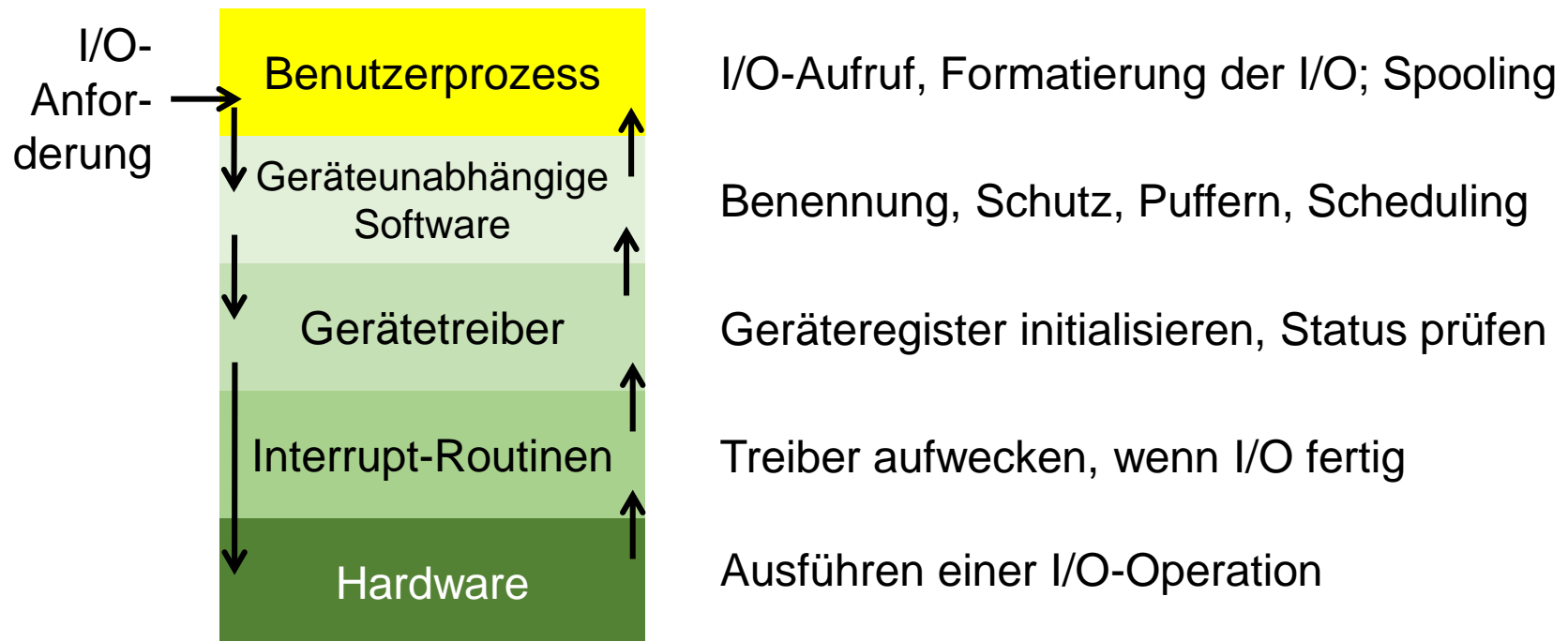
# Doppelpufferung

- ▶ Wenn der Puffer voll ist, wird er vom Kern in den Benutzerraum kopiert
  - ▶ Was passiert mit Daten, die während dieser Zeit ankommen?
- ▶ Lösung: ein zweiter Puffer, der (für die Eingabe) verwendet wird, wenn der 1. voll ist
  - ▶ Sie wechseln sich ab, sobald einer voll ist
- ▶ Eine weitere Form
  - ▶ **Zyklischer Puffer**
    - ▶ Zwei Zeiger: wo aktuell geschrieben, wo gelesen wird
  - ▶ Hatten wir schon bei dem Producer-Consumer Problem verwendet



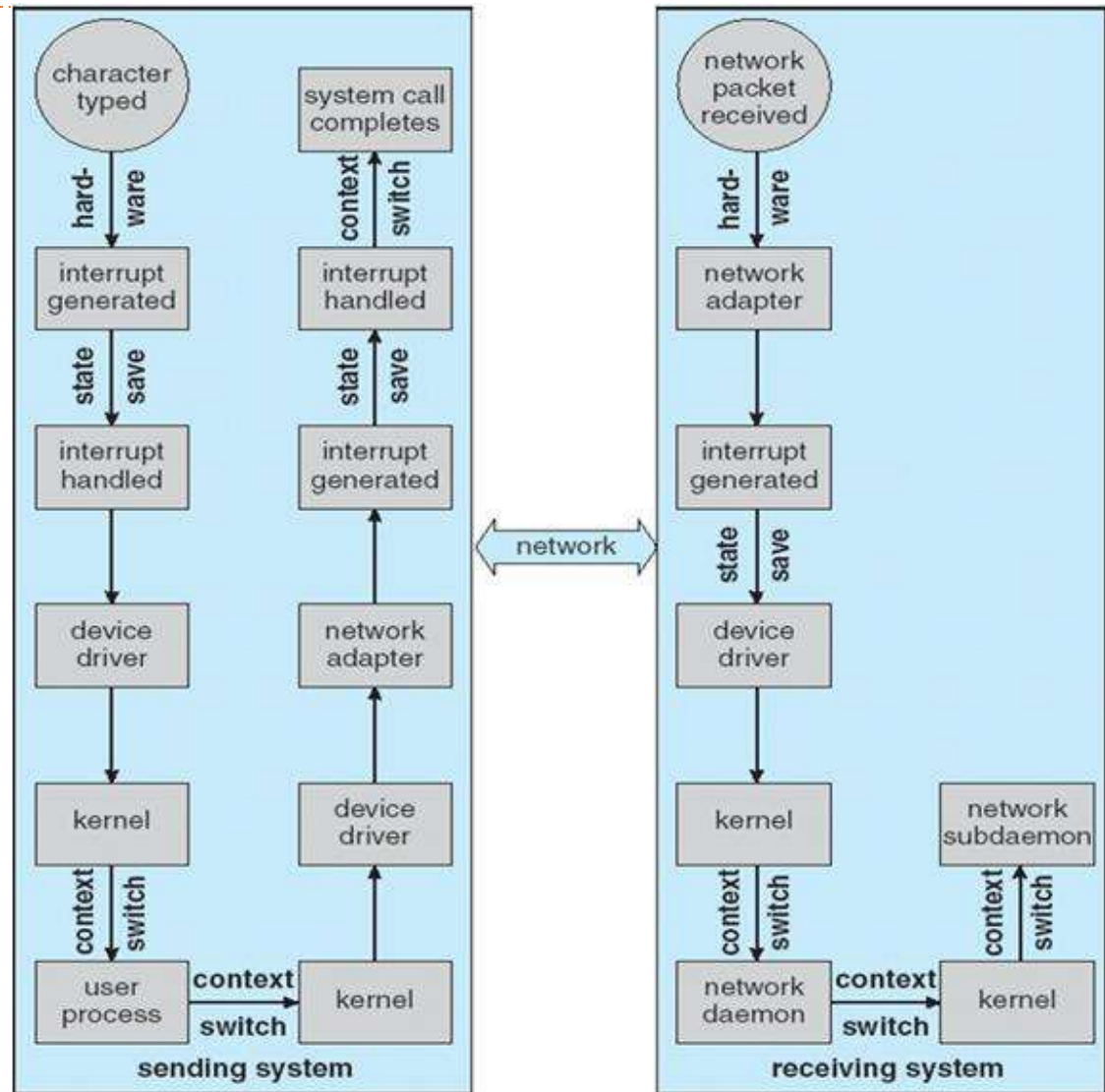
# Lebenszyklus einer I/O-Anforderung

- ▶ Der BS-Teil für I/O ist (wie fast alles andere) in Schichten aufgebaut
  - ▶ Jede Anfrage durchläuft alle diese Schichten
  - ▶ Das reduziert die Komplexität des BS, aber kostet Zeit



# Ein Beispiel: Telnet

- ▶ Die Eingabe eines einzigen Zeichens bewirkt mehrere **Kontextwechsel**
- ▶ Solaris-Entwickler haben den **telnet-Dämon** nur in Kernraum (neu) implementiert
  - ▶ Das hat die max. mögliche Anzahl der Logins auf einem großen Server von *Hundert* auf *Tausende* erhöht



# Wie kann man I/O effizienter machen?

---

- ▶ Reduziere die Anzahl der Kontextwechsel
- ▶ Reduziere das Kopieren der Daten im Speicher während der SW-Stack durchlaufen wird
- ▶ Reduziere die Frequenz der Interrupts durch große Transfers und intelligentere HW
- ▶ Delegiere das Kopieren von Daten von CPU auf DMA
- ▶ Verschiebe einfache Funktionen (primitives) in die HW
- ▶ Balanciere die Leistung von CPU, Speicher, Bussen, I/O-Geräten
  - ▶ Überlastung eines Bereichs lässt die anderen leerlaufen

# Zusatzfolien: Schnittstellen zu den I/O-Geräten

# I/O-Geräte und das Betriebssystem

## ▶ Aufgaben des BS bei den I/O-Geräten

### ▶ **Effizienz**

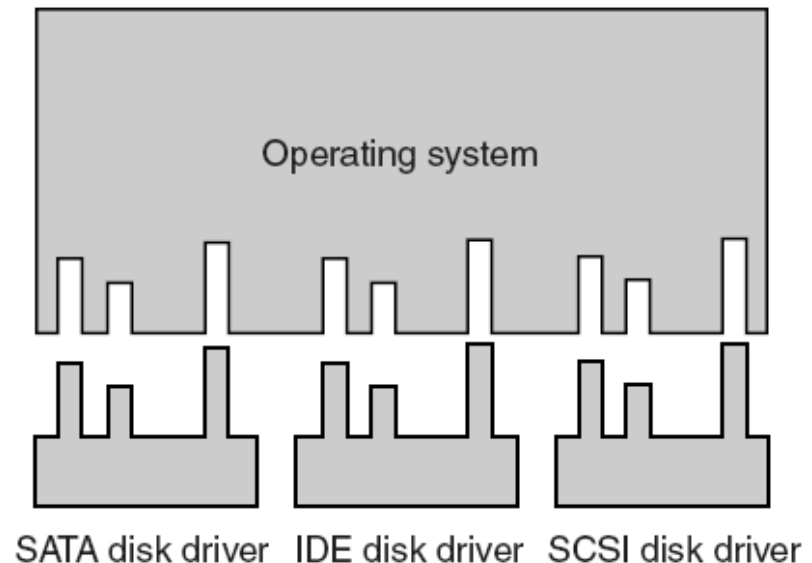
- ▶ Die I/O-Geräte sind meist langsamer als die CPU
- ▶ Durch Multiprogrammierung und Interrupts kann man vermeiden, dass die CPU wartet

### ▶ **Geräteunabhängigkeit** (bzw. **Allgemeingültigkeit**)

- ▶ Z.B. gleiche API beim Zugriff auf Festplatte, CD-ROM, DVD usw.

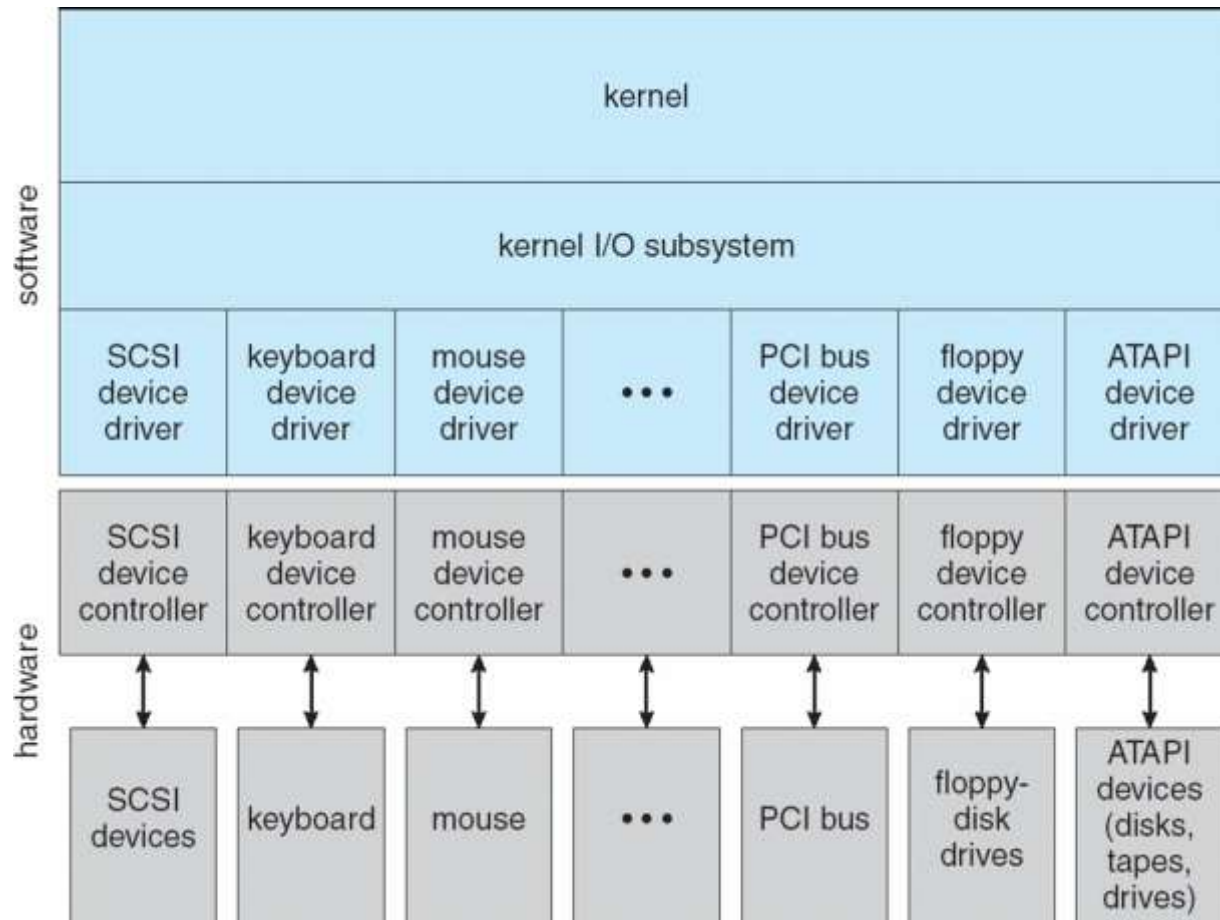
## ▶ Weitere Aufgaben

- ▶ Pufferung
- ▶ Fehlerbericht
- ▶ Anforderung und Freigabe von exklusiv zugewiesenen Geräten



# Geräteunabhängigkeit

- Wird erreicht, indem I/O-Geräte in mehrere generische Klassen unterteilt werden, die die HW-Differenzen verstecken





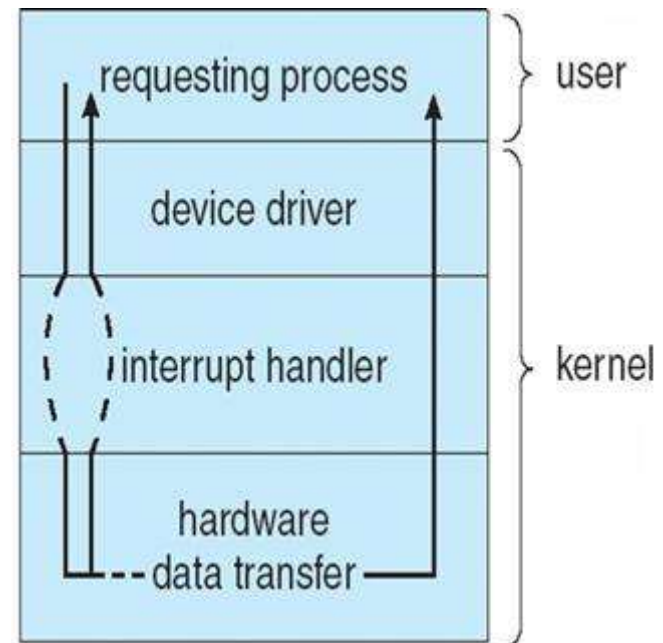
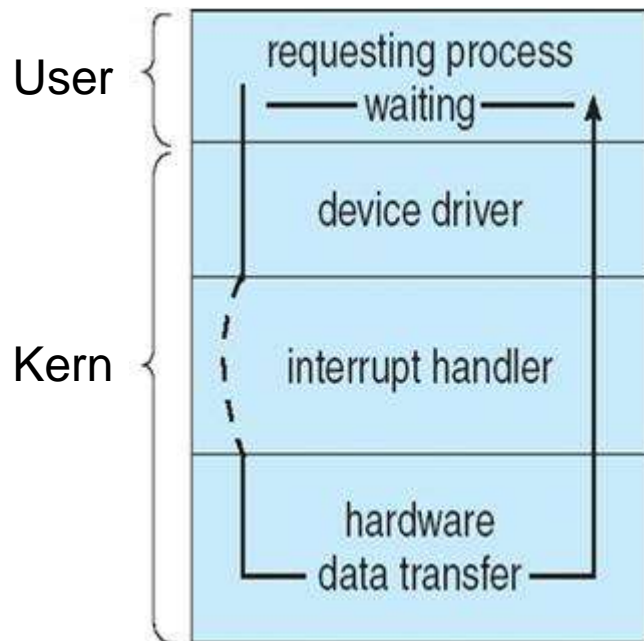
# Gruppieren von I/O-Geräten

## ► ... anhand einiger Aspekte

Aspekt	Varianten	Beispiele
Daten-Transfer-Modus	<b>Zeichen (character)</b> <b>Block (block)</b>	Terminal Festplatte
Zugriffsmethode	<b>sequentiell</b> <b>wahlfrei (random)</b>	Modem CD-ROM
Transferablauf (transfer schedule)	<b>synchron</b> <b>asynchron</b>	Magnetband Tastatur
Gemeinsame Nutzung (sharing)	<b>dediziert</b> <b>gemeinsam nutzbar</b>	Magnetband Audiokarte
Gerätegeschwindigkeit	<b>Latenz, Zugriffszeit, Transferrate</b>	
I/O-Richtung	<b>nur-lesen</b> <b>nur-schreiben</b> <b>lesen/schreiben</b>	CD-ROM Graphikkarte Festplatte

# Synchrone vs. Asynchrone Aufrufe

- ▶ Bei **blockierenden** bzw. **synchronen** Aufrufen wird die Anwendung angehalten, bis das I/O-Gerät fertig ist
- ▶ **Nicht-blockierende** bzw. asynchrone Aufrufe kehren sofort zurück – mit einem Handle für spätere Überprüfung



# Escape / Back Door

---

- ▶ Manchmal versteckt die „Unifizierung“ wichtige Eigenschaften der Geräte
- ▶ Deshalb gibt es Systemaufrufe, die transparent Befehle der Anwendung an den Treiber übermittelt
  - ▶ **Escape** oder **Back Door**
- ▶ UNIX: **ioctl** (devicefile, command, \*data)
- ▶ Windows: **DeviceIoControl** () in Win32 API
  - ▶ Objekt-Handle (equivalent zu Dateideskriptor devicefile)
  - ▶ Befehlsnummer ("control code")
  - ▶ Puffer für Eingabe
  - ▶ Länge der Eingabe
  - ▶ Puffer für Ausgabe
  - ▶ Länge der Ausgabe
  - ▶ OVERLAPPED-Struktur, Daten für einen asynchronen Aufruf

# Gerätedateien - Device Files ([Link](#))

---

- ▶ In UNIX-artigen Systemen wird auf I/O-Geräte wie Festplatten, Drucker, virtuelle Terminals über ein spezielles Dateisystem zugegriffen - /dev
  - ▶ Hatten wir schon bei der Vorlesung über Dateisysteme
  - ▶ Beispiel für uniforme Handhabung von I/O-Geräten
- ▶ Beispiele (alle im Verzeichnis /dev) ([Link](#))
  - ▶ **fd**: Floppy Disk
  - ▶ **hd**: Festplatten von Typ IDE (hda, hdb, hdc, hdd)
  - ▶ **lp**: Drucker
  - ▶ **pt**: Pseudo-Terminals (z.B. xterm)
  - ▶ **ht0**: 1. Bandlauferk (Tape) (ht1, ht2, usw.)
  - ▶ **md0**: 1. Meta-Disk-Gruppe (wichtig für RAID)

# Zusatzfolien: Drei Verfahren der Ein-/Ausgabe

# Beispiel - Druckerausgabe per Interrupts

---

## Code direkt nach System-aufruf

- ▶ copyFromUser (buffer, p, count);
- ▶ // Ausgabe, falls Drucker frei
- ▶ if(\*printer\_status\_reg !=  
    READY);
  - ▶ \*printer\_data\_register = p[0];
- ▶ scheduler( );
- ▶ // nun ist dieser Prozess im  
Zustand „waiting“, bis das letzte  
Zeichen ausgegeben wurde

## Code der Interrupt-Routine (kernel)

- ▶ if (count == 0) {
  - ▶ unblock\_user( );
- ▶ } else {
  - ▶ \*printer\_data register = p[i];
  - ▶ count = count - 1;
  - ▶ i = i + 1;
- ▶ }
- ▶ acknowledge\_interrupt( );
- ▶ return\_from\_interrupt( );

# Zusatzfolien: IT-Angriffe

# Insider-Angriffe

---

## ▶ **Logische Bombe**

- ▶ Ein Codestück, das zu einem festgelegten Zeitpunkt den Betrieb stört oder Schaden anrichtet

## ▶ Anwendungen

- ▶ Erpressung nach Entlassung: z.B. ein von einem (momentan angestellten) Programmierer / Admin geschriebener Code, welcher das System kritisch ändert
  - ▶ Erzwingt Wiedereinstellung des Programmierers / Admins als 'ein hochbezahlter „Berater“'
- ▶ Politische Ziele, z.B. Protest zu einem bestimmten Datum
  - ▶ Oft Infektion via Viren





# Insider-Angriffe /2

---

## ▶ **Falltüren (trap doors)**

- ▶ Umgehen der Passwort-Abfrage durch zusätzlichen Code
- ▶ Abhilfe: Code-Reviews

## ▶ **Login-Spoofing**

- ▶ Falsche Login-Maske, die die Passwörter sammelt
- ▶ Bei einem Loginversuch wird das eingegebene Passwort gespeichert und ein Scheitern des Logins simuliert
- ▶ Abhilfe: Login mit einer Tastensequenz starten, die ein Benutzerprogram nicht abfangen kann



# Formatstring – Angriff /1

---

- ▶ Die korrekte Verwendung von printf ist
  - ▶ `printf (formatstring, param1, param2,...)`
- ▶ Man kann aber printf wie `puts` verwenden
  - ▶ `char s[100], g[100] = "Hello ";`
  - ▶ `gets(s); strcat(g, s);`
  - ▶ `printf(g);`
- ▶ Der Programmierer hat dadurch (unbeabsichtigt) ermöglicht, bei „gets“ einen Formatstring anzugeben

# Formatstring – Angriff /2

---

- ▶ printf / sprint haben einen den Formatierungscode „%n“
  - ▶ Dieser berechnet, wie viele Zeichen bis zu dieser Stelle ausgegeben wurden, und speichert diese Zahl im nächsten zu verwendenden printf-Parameter ab
  - ▶ `int i = 0;`
  - ▶ `printf(„Bla bla %n“, &i);`
  - ▶ => Variable i wird auf 8 gesetzt
- ▶ Wir können also damit in den Speicher schreiben
  - ▶ Damit kann man einen Formatstring erzeugen, der die Rücksprungadresse modifiziert
  - ▶ Details sind etwas komplizierter, siehe hier: [Link](#)

# Werden Systeme Sicherer?

---

- ▶ Die Methode des Pufferüberlaufs wird allmählich verschwinden
- ▶ Der Trend geht dahin, sichere Programmierpraktiken anzuwenden und danach **automatische Codescanning-Tools** zu verwenden, um Fehler zu finden
  - ▶ Z.B. die Tools PREfix und PREfast von Jon Pincus, Microsoft
- ▶ Es werden zunehmend **typsichere Sprachen** (Java, C#) verwendet
  - ▶ Hier können Daten wie Strings nie „ausgeführt“ werden
- ▶ In moderner HW (MMUs) kann man **in der Seitentabelle festlegen, ob eine Seite ausführbar ist**
  - ▶ Die BS nutzten das kaum, werden aber künftig mehr

# Zusatzfolien:

## Betriebssystem „Android“

# Android – Verkaufszahlen 3Q2011

---

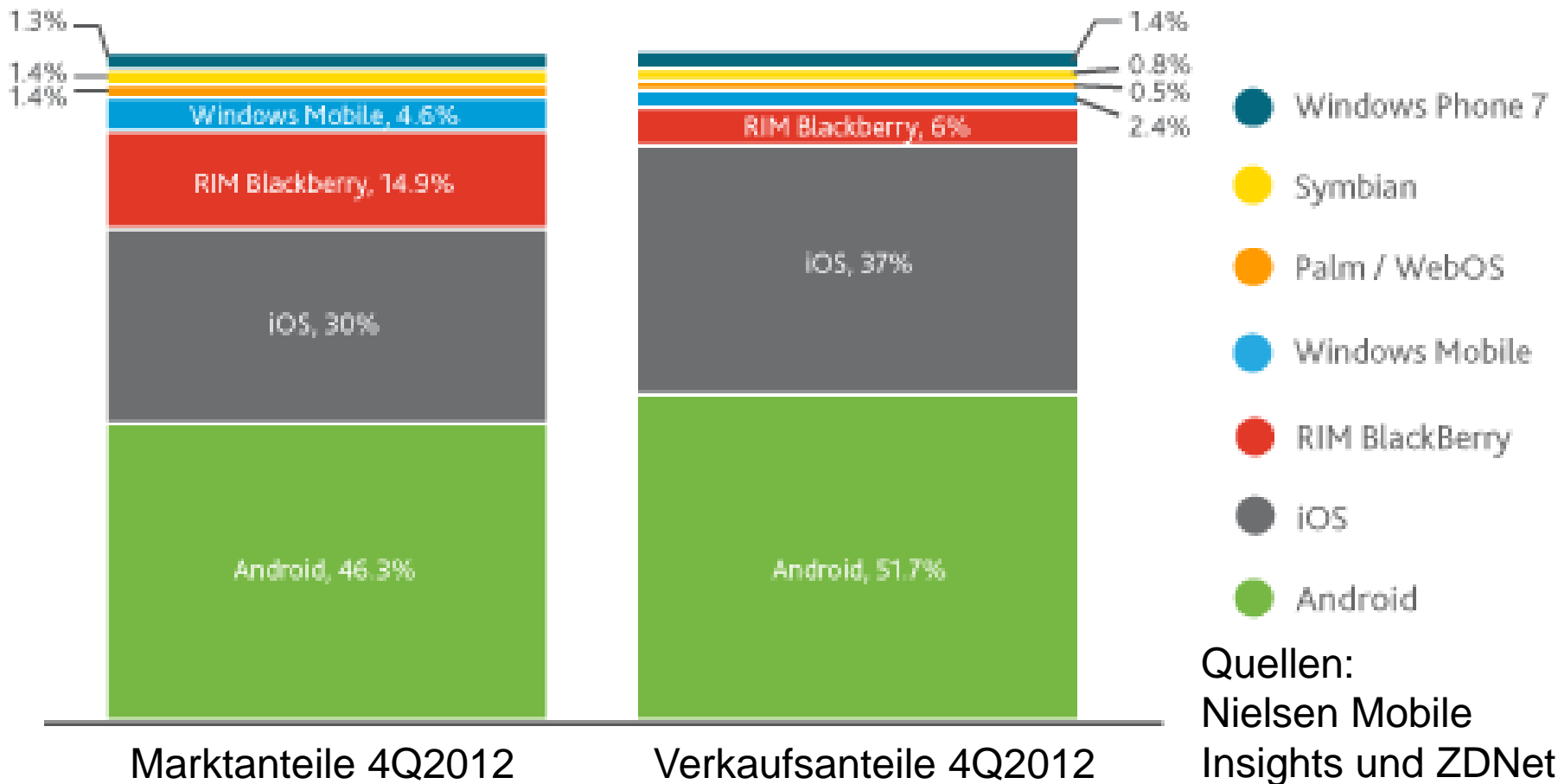
- ▶ Verkaufszahlen von Handys (weltweit, 3. Quartal 2011): 440 Mio, davon 115 Mio Smartphones ([Link](#))

BS	3Q11 Anteil (%)
Android	52.5
Symbian	16.9
iOS	15.0
Research in Motion	11.0
Bada	2.2
Microsoft	1.5
Andere	0.9

BS	3Q <sup>10</sup> Anteil (%)
Android	25.3
Symbian	36.3
iOS	16.6
Research in Motion	15.4
Bada	1.1
Microsoft	2.7
Andere	2.5

# Android – Marktanteile 4Q2012

- ▶ Marktanteile und Verkaufsanteile von Handys (weltweit, 4. Quartal 2012) ([Link](#))



# Dalvik - Java Virtual Machine bis Android 4+

---

- ▶ „Dalvik“ JVM emulierte eine Java Registermaschine im Gegensatz zum Kellerautomaten (Sun's JVM)
  - ▶ Kellerautomat hat i.W. keine Register und speichert Zwischenergebnisse und Methodenparameter auf dem Stack
  - ▶ Dalvik VM kann aber Registerarchitektur moderner Prozessoren (z.B. ARM) besser nutzen
- ▶ Das Programm dx in Android-SDK konvertiert „normale“ Java-Binärdateien (.class) in das **Dalvik Executable-Format** (.dex)
  - ▶ Zur Laufzeit wird Dex-Code durch JIT-Compiler übersetzt
- ▶ Ab Android 5.0 wurde Dalvik durch den Ahead-of-time-Compiler Android Runtime (ART) ersetzt
  - ▶ Dex-Bytecode wird einmalig bei Installation einer App in Maschinencode umgewandelt (dex2oat-Tool)