# Einführung in die Betriebssysteme

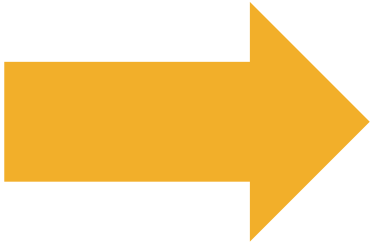
Martin Spörl

# Virtualisierung

# Grundlagen I

**Früher**

- Silos
  - Pro Server 1 Aufgabe
  - Dedizierte Server
  - Keine Ressourcen Teilung => führt zu nicht genutzten Ressourcen (schlechte Auslastung)
- Monolithische Infrastruktur
  - Ein Server macht alles (UI, App, DB)
  - keine Skalierbarkeit
  - schwer wartbar
  - Schlechter Überblick



**Virtualisierung**

- teilen von physikalischen Ressourcen
- sehr hohe Skalierbarkeit
- hohe Ausfallsicherheit
- sehr gute Wartbarkeit

# Grundlagen II

## Simulation

- Nachahmen durch ein Modell
- Modell soll internes Verhalten des realen Systems kopieren
- Ziel: Durch Modelle Erkenntnisse über reales System
- Beispiele
  - Flugsimulator
  - Spice
  - Crashtest

## Emulation

- Nachahmen eines sichtbaren Verhaltens
- internes Verhalten des emulierten System muss nicht mit originalem System übereinstimmen
- Beispiele:
  - DOSBox
  - QEMU

## Virtualisierung

- Nachbilden eines real existierenden Systems
- Verhalten und Funktion wird durch nachbilden identisch
- Beispiele:
  - KVM
  - VMWare ESXi

oft kombiniert (z.B. Prozessor virtualisiert; BIOS wird emuliert)

# Arten I – Applikationsvirtualisierung

## Grundlagen

- Idee: entkoppeln der Anwendung von Betriebssystem
- Anwendung läuft in einer isolierten Umgebung (Sandbox)
- Anwendung nicht unbedingt auf dem OS installiert

## Beispiele

- VMware ThinApp
- Citrix XenApp

# Arten II – Desktop Virtualisierung

## Grundlagen

- Idee: Entkoppeln des Desktop (Arbeitsumgebung) von Hardware / OS
- Nutzen von virtueller Desktop Infrastruktur („VDI“)
- Ermöglicht User standardisierte und leistungsfähige Umgebung von überall zu nutzen

## Beispiele

- Microsoft VDI
- VMware Horizon
- Citrix XenDesktop

# Arten III - Hardware-/ Server Virtualisierung

**Grundlagen**

- Idee: die gesamte Software wird von der Hardware entkoppelt
- Systeme laufen in virtuellen Maschinen
- ermöglicht eine bessere Auslastung der physikalischen Ressourcen

**Beispiele**

- Microsoft Hyper-V
- VMware ESXi
- Oracle Virtualbox
- VMware Workstation

# Arten IV - Netzwerkvirtualisierung

## Grundlagen

- Idee: Netzwerkressourcen in logische Einheiten zusammenfassen
- Physikalische Netzwerk erhält virtuelle Schicht
  - Schicht besteht aus verschiedenen, voneinander isolierten Segmenten
  - Administratoren können Netzwerk dynamisch anpassen (ohne z.B. Verkabelung zu ändern)

## Beispiele

- Virtual Private Network (VPN)
- Virtual Local Area Network (VLAN)
- Software-defined Network (SDN)

# Arten V - Speichervirtualisierung

## Grundlagen

- Idee: Speicherressourcen in logische Einheiten zusammenfassen
- Auch als Software-Defined-Storage bezeichnet
- Speichermedien werden in Pools zusammengefasst
- Server / Betriebssystem bekommen logische Einheiten aus Storage Pool zugewiesen
- Kann Speicherkapazität erhöhen (z.B. Thin-Provisioning)

## Beispiele

- NAS-Systeme
- Enterprise Storage System (z.B. HPE 3Par)
- VMware vSAN

Storage Virtualisierung != RAID

Storage Virtualisierung nutzt RAID

# Hardwarevirtualisierung

- Die wohl meistverbreiteste Virtualisierung neben Desktop- und Netzwerkvirtualisierung
- Benötigt auf der Hardware ein spezielles Betriebssystem („Hypervisor“)
- OS virtualisiert notwendige Hardware für virtuelle Computer
- Heute oft gepaart mit diversen Emulatoren für bestimmte Hardwaretypen


**Software Assisted Virtualization**

- Virtualisierung wird von Software durchgeführt
- Instruktionen, Interrupts, I/O Requests, usw. wurden meist emuliert bzw. übersetzt
- Sehr unperformant

**Hardware Assisted Virtualization**

- Virtualisierung wird bereits durch Hardware durchgeführt
- Instruktionen werden direkt auf der CPU ausgeführt (in unterschiedlichen Kontexten)
- Hohe Performance
- Hohe Sicherheit

# Komponenten für Virtualisierung

## Host-Rechner

- Rechner, der alles ausführt
- i.d.R. CPU mit Virtualisierungsunterstützung

## Hypervisor

- Virtualisierungsschicht zwischen realer Hardware und Gastsystem
- Verwaltet & Überwacht Gastsysteme
- Wird oft auch „Virtual Machine Monitor/Manager“ (VMM) genannt

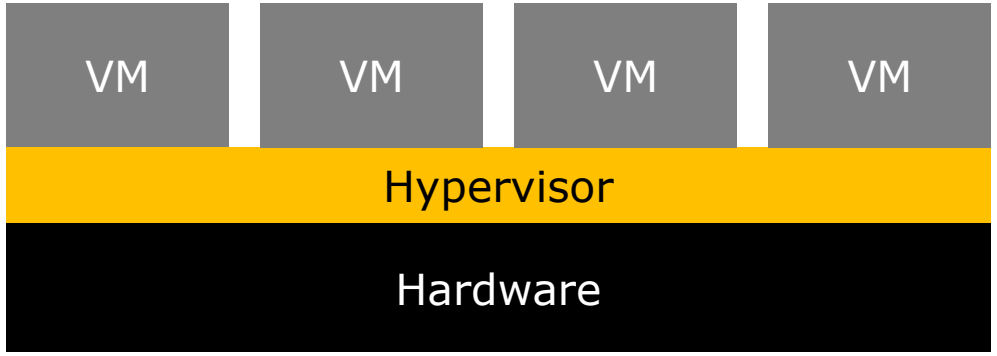
## Gastsystem

- Zusätzliches Betriebssystem auf dem Rechner
- Kann anderes Betriebssystem als Hypervisor sein
- Muss der Prozessorarchitektur des Host-Rechners entsprechen (z.B. x86 kann kein ARM virtualisieren)

# Hypervisor

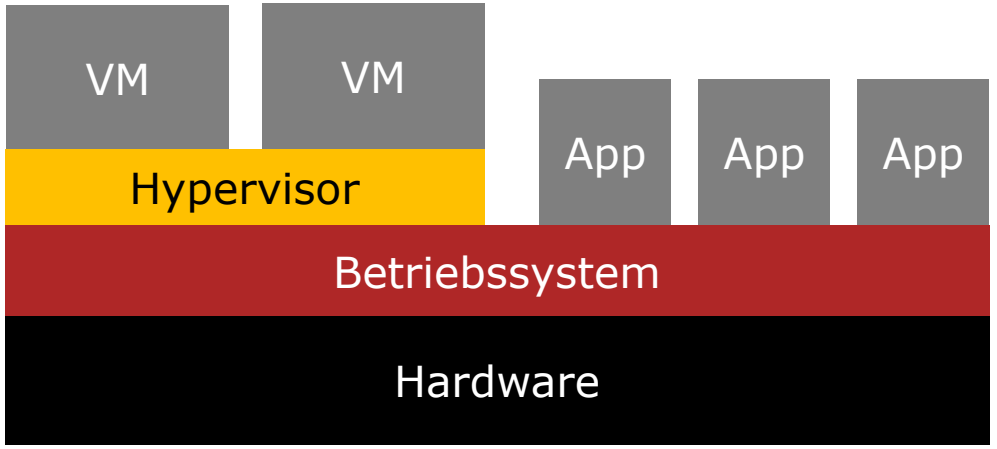
**Typ 1**

- Hypervisor ist das Betriebssystem
- Hardware muss von den Treibern des Hypervisors unterstützt sein




**Typ 2**

- Hypervisor setzt auf Betriebssystem auf
- Kann OS Treiber nutzen



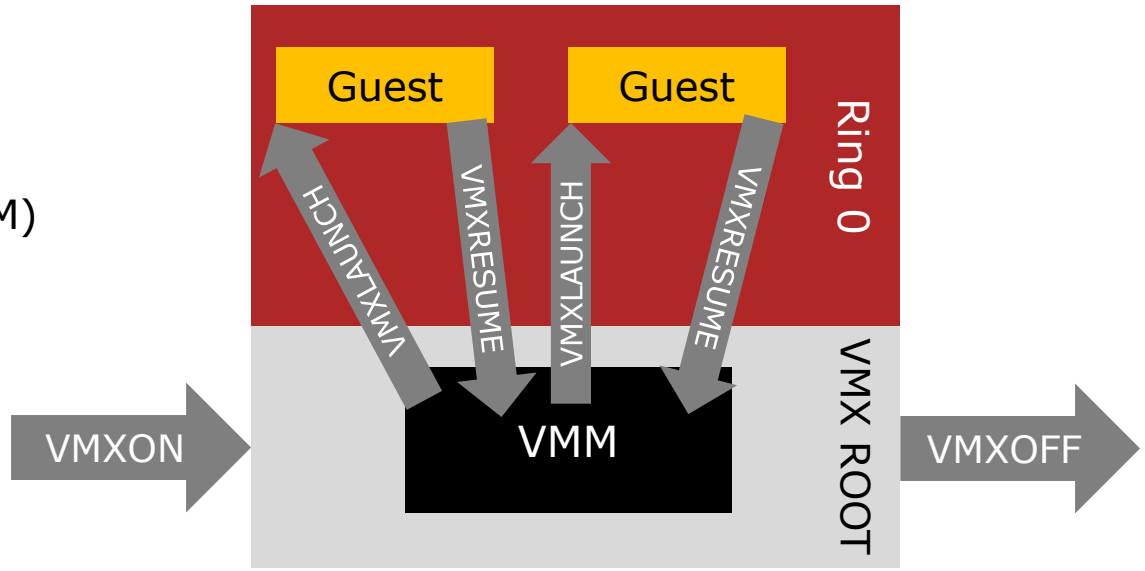
# Virtualisierung in der Praxis – Intel VT-x / AMD-V

## Grundlagen

- CPU unterstützte Virtualisierung
- Grundidee: Einführung der Virtual Machine Extension (VMX)
  - Einfügen eines -1ten Ring (vor Ring 0): VMX ROOT (nur für Hypervisor gedacht!)
  - 10 neue Instruktionen für das Überwachen und Verwalten von VMs
  - VMs laufen damit in eigenem Ring-0 Kontext ohne VMM zu gefährden
- Bringt diverse weitere Erweiterungen (Auszug)
  - Interrupt Mapping (Interrupts direkt an VM)
  - I/O Device Assignment (z.B. PCI Passthrough) – Geräte direkt an VM
- Muss im BIOS aktiviert werden
- Auf fast allen modernen CPUs vertreten (nur sehr sehr sehr wenige Ausnahmen)

## Funktionsweise (vereinfacht)

- VMXON" aktiviert VMX ROOT Mode
- Hypervisor läuft in VMX ROOT
- Hypervisor kann mit „VMLAUNCH" & „VMRESUME" zwischen den VMs wechseln
- „VMXOFF" beendet VMX ROOT Mode



# Virtualisierung in der Praxis – Intel EPT / AMD RVI

**erste Generation**

- Nur VMM hat Zugriff auf Page Translation Table
- VMM emulierte Page Translation Table für Guest („Shadow Page Table“)
- Problem:
  - Guest kann emulierte Page Translation Table anpassen
  - VMM muss Tabelle auf seine eigene synchronisieren

Performance sinkt!

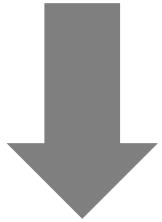
**Lösung**

- Implementierung von Second Level Address Translation (SLAT)
- Intel CPU: Extended Page Tables (EPT)
- AMD CPU: Rapid Virtualization Indexing (RVI)
- Idee:
  - Translation Lookaside Buffer (TLB) weißt, dass er auch von Gast gesehen wird
  - TLB kann nun Gast-spezifischen Memory selbst verwaltet
  - VMM muss Page Translation Table nicht mehr synchronisieren
- **Massiv Performancesteigerung bei Memory intensiven Applikation**

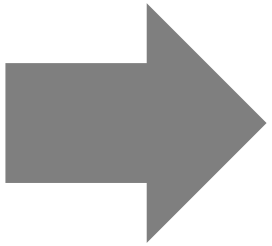
# Container Grundlagen I

**Virtualisierung**

- Pro VM wird OS Stack neu aufgebaut
- Verbraucht viel Leistung (CPU, Memory, Speicher)

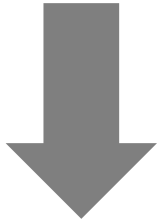


Wunsch auf Applikationsebene zu isolieren



**Containervirtualisierung**

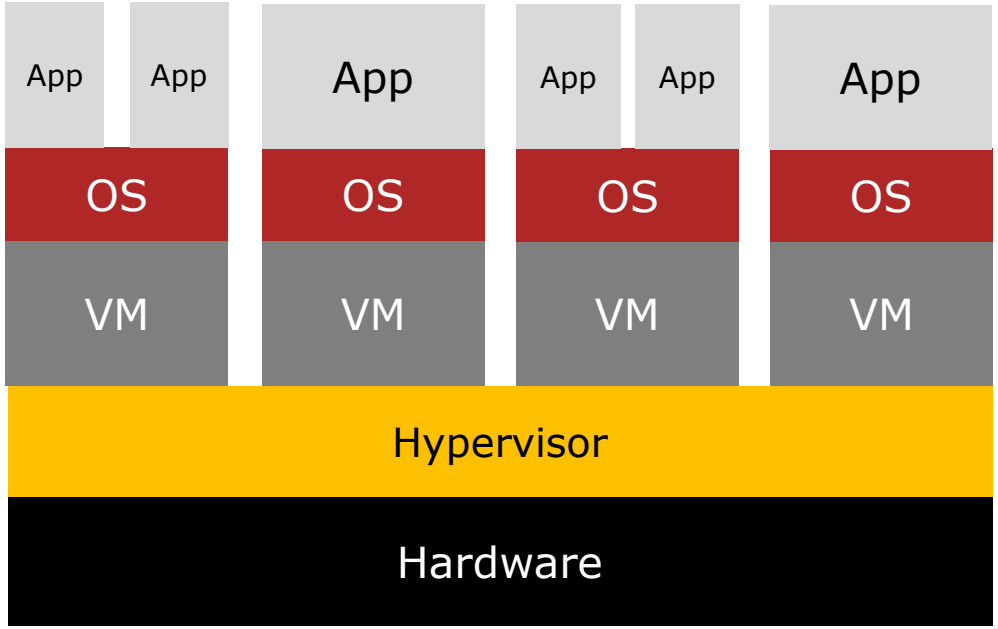
- teilen sich den Betriebssystemkern
- Isolieren durch verändern der Sichtbarkeiten
- Werden so portable zwischen Betriebssystem
- Container sind selbst eigene Prozesse



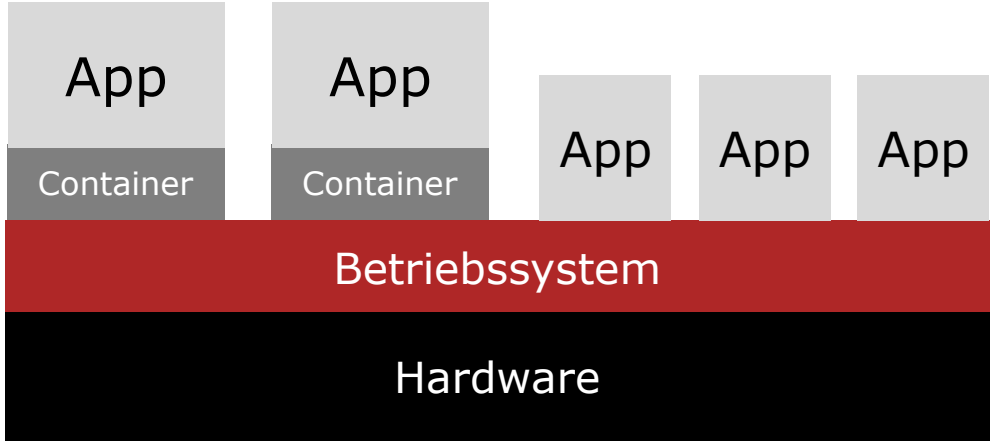
Kann unterschiedlich erreicht werden

# Container Grundlagen II

## Virtualisierung




## Containervirtualisierung



# Container Plattformen

## LXC & LXD

- „Linux Containers“
- Kann innerhalb der Container verschiedene Betriebssysteme betreiben
- wird gern als „Zwischenschritt zu Microservices“ betrachtet (wegen OS Unterstützung in Container)

## Docker

- Container Implementierung
- Nutzt cgroups & namespaces zur Isolierung der Container
- Die wohl am meisten genutzte / verbreitetste Implementierung

## rkt (Rocket)

- Ursprüngliche Idee: Container Plattform, die Probleme von Docker nicht hat (fehlende Features, Sicherheit)
- Kann mit systemd und init arbeiten

# Docker – Funktionsweise

## chroot

- „change root“
- Verschiebung des Root Verzeichnisses für bestimmten Prozess (+Kind-Prozesse
  - für Prozess wird so aus „/home/myuser/container1“ „/“
  - Prozess kann nicht außerhalb der Wurzel agieren
- Problem: Da Kernel Funktion nicht in „Jail“ wechselt, ist mit besonderen Rechten ein Ausbruch möglich

## Namespace

- Entwickelt um Problem mit chroot zu lösen
- Innerhalb des neue Namespaces, sehen Prozess und Kind-prozess nur sich selbst (keine anderen PIDs)
- Docker nutzt folgende Namespaces:
  - **PID namespace** für Prozessisolation.
  - **NET namespace** um Netzwerkports zu verwalten.
  - **IPC namespace** um Interprocess Communication zu verwalten.
  - **MNT namespace** um Mountpoints zu verwalten.
  - **UTS namespace** um Kernel und Versionszeiger zu isolieren.

## cgroups

- Kernel Control Groups
- limitieren Prozess bzgl. Ressourcen (Verfügbarkeit, Sichtbarkeit)
- erlaubt es Docker Hardware Ressourcen zwischen Containern zu teilen
- Docker nutzt u.a. folgende cgroups:
  - **Memory cgroup** um Speicher zu verwalten und zu limitieren.
  - **CPU cgroup** um CPU Nutzung zu verwalten.
  - **BlkIO cgroup** um BlockIO zu messen und zu limitieren
  - **Devices cgroup** für Lese-/Schreibzugriff auf Geräte.

# Docker I – Wichtige Befehle

| Befehl | Erklärung | (Basic) Syntax | Beispiel |
|---|---|---|---|
| docker build | Erstellt ein Dockerimage | docker build --tag <image tag> | docker build --tag myimage |
| docker run | Erstellt und startet einen Dockercontainer | docker run <image name> | docker run -d myimage |
| docker ps | Zeigt alle laufenden Dockercontainer an | docker ps | docker ps |
| docker stop | Stoppt einen Dockercontainer | docker stop <container id> | docker stop 9dd4e3 |
| docker rm | Löscht ein Dockercontainer | docker rm <container name> | docker rm mycontainer |
| docker start | Startet einen Dockercontainer | docker start <container name> | docker start mycontainer |

# Docker II

**Projektaufbau**

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ tree
.
├── dockerfile
└── html
    └── index.html

1 directory, 2 files
```

**dockerfile**

FROM httpd:2.4

COPY ./html/ /usr/local/apache2/htdocs/

Basis Image

**Build Prozess**

*docker build ./ --tag apache2sample*

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker build ./ --tag apache2sample
Sending build context to Docker daemon  3.584kB
Step 1/2 : FROM httpd:2.4
2.4: Pulling from library/httpd
f17d81b4b692: Pull complete
06fe09255c64: Pull complete
0baf8127507d: Pull complete
07b9730387a3: Pull complete
6dbdee9d6fa5: Pull complete
Digest: sha256:90b34f4370518872de4ac1af696a90d982fe99b0f30c9be994964f49a6e2f421
Status: Downloaded newer image for httpd:2.4
 ---> 55a118e2a010
Step 2/2 : COPY ./html/ /usr/local/apache2/htdocs/
 ---> e973505cfa2d
Removing intermediate container fdee1f9852ed
Successfully built e973505cfa2d
Successfully tagged apache2sample:latest
martin@ubuntu:~/Desktop/Vorlesung/Docker$
```

# Docker III

**Container starten**

*docker run --name myapache -d -p 8000:80 apache2sample*

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker run --name myapache -d -p 8000:80 apache2sample
e8b18e898a3d9738edbd473a31a4090efd0bf3ac60f3aa4214068bce3df8937b
martin@ubuntu:~/Desktop/Vorlesung/Docker$
```

Container Port 80 mapped to Host Port 8000

Full Container ID

# Docker IV

**Laufende Container auflisten**

*docker ps*

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
e8b18e898a3d        apache2sample       "httpd-foreground"       About a minute ago   Up About a minute   0.0.0.0:8000->80/tcp   myapache
martin@ubuntu:~/Desktop/Vorlesung/Docker$
```

Short Container ID

Image (tag) Name

Port mapping

# Docker V

**Laufende Container stoppen**

*docker stop <full | short container id>*

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker stop 19ac13361048
19ac13361048
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
martin@ubuntu:~/Desktop/Vorlesung/Docker$
```

# Docker VI

**Container löschen**

*docker rm myapache*

```
martin@ubuntu:~/Desktop/Vorlesung/Docker$ docker rm myapache
myapache
martin@ubuntu:~/Desktop/Vorlesung/Docker$
```