

Verteilte Systeme

Prof. Dr. Martin Becke

CaDS - HAW Hamburg

Version 0.9

Inhalt

1 Prozesse und Threads

- Threads
- Prozesse
- Virtualisierung
- Cluster und Grid

Threads

Motivation für VS

- ▶ Parallelität
- ▶ Effiziente Ressourcennutzung
- ▶ Reaktionsfähigkeit
- ▶ Einfachere Kommunikation und Synchronisation
- ▶ Granularität

Threads

Herausforderung für VS

- ▶ Deadlocks
- ▶ Race Conditions
- ▶ Verteilung/Skalierung über nodes

Threads

Thread Modelle

- ▶ Many-to-One (User Thread)
- ▶ One-to-One-Thread-Modell (Kernel Thread)
- ▶ N :M-Thread-Modell (Nicht so üblich)

Threads

VS Anwendung Threadpool

- ▶ Effiziente Ressourcennutzung
- ▶ Einfache Verwaltung

Threads

VS Anwendung ThreadPool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
    public static void main(String[] args) {
        int numThreads = 5;
        ExecutorService executor =
            Executors.newFixedThreadPool(numThreads);

        for (int i = 0; i < 10; i++) {
            Runnable task = new ExampleTask(i);
            executor.execute(task);
        }

        executor.shutdown();
    }
}
```

Threads

VS Anwendung ThreadPool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {
    public static void main(String[] args) {
        int numThreads = 5;
        ExecutorService executor =
            Executors.newFixedThreadPool(numThreads);

        for (int i = 0; i < 10; i++) {
            Runnable task = new ExampleTask(i);
            executor.execute(task);
        }

        executor.shutdown();
    }
}
```


Threads

VS Anwendung ThreadPool

```
class ExampleTask implements Runnable {  
    private int taskId;  
  
    public ExampleTask(int taskId) {  
        this.taskId = taskId;  
    }  
  
    @Override  
    public void run() {  
        System.out.println("Task " + taskId + " is  
            running on thread: " +  
            Thread.currentThread().getName());  
    }  
}
```

Listing 3 – ExecutorService-Klasse III

Threads

VS Anwendung Threadpool

- ▶ Im besten Fall Umsetzung mit Tasks und Interface Strukturen
- ▶ Anzahl Threads maximal gleich der Anzahl der verfügbaren Prozessorkerne bei rechenintensiven Aufgaben
- ▶ Mehr Threads als Prozessorkerne nur sinnvoll bei blockierenden Strukturen

Threads

Threadpool Task

- ▶ Modularität
- ▶ Kapselung
- ▶ Idempotenz
- ▶ Zustandslosigkeit
- ▶ Fehlertoleranz
- ▶ Skalierbarkeit
- ▶ Kommunikation

Threads

Multithreaded Clients

- ▶ Oft mit verschiedenen Arten von Aufgaben konfrontiert
- ▶ Aufgaben unabhängig voneinander und gleichzeitig
- ▶ Verwendung von Threads typischerweise zur Organisation

Threads

Multithreaded Clients

- ▶ Reaktionsfähigkeit
- ▶ Modularität und Wartbarkeit
- ▶ Bessere Ressourcennutzung
- ▶ Vereinfachte Kommunikation
- ▶ Beispiel Browser : Surfen und gleichzeitiger Datei-Download

Threads

Thread-Level-Parallelism (TLP)

- ▶ Maß dafür, wie viele Threads in einer Anwendung gleichzeitig ausgeführt werden können
- ▶ Berechnung basiert auf Speedup

$$TLP = \frac{Speedup}{p} = \frac{\frac{T_1}{T_p}}{p} \quad (1)$$

- ▶ Beispiel TLP von 1,5 -2 bedeutet 1,5 bis 2 Threads gleichzeitig auszuführen.

Threads

Single Threaded Process

- ▶ Nur ein einzigen Ausführungsstrang (Thread)
- ▶ Eine Aufgabe zur Zeit abgearbeitet
- ▶ Daher sequenziellen Reihenfolge der Aufgaben
- ▶ Beispiel nodejs

Threads

Single Threaded Process

- ▶ Einfachheit
- ▶ Skalierbarkeit
- ▶ Geringerer Ressourcenverbrauch
- ▶ Nutzt Rechenleistung von Mehrkernprozessoren nicht voll aus
- ▶ Gut im Cluster-Betrieb für Single-Node Architekturen
- ▶ Notwendig : Blocking mit Timeout oder Non-Blocking IO

Threads

Single Threaded Process

```
const http = require("http");

const server = http.createServer((req, res) => {
  console.log("Anfrage empfangen");

  // Simuliere eine zeitaufwaendige Operation
  setTimeout(() => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hallo Welt!");
  }, 1000);
});

server.listen(3000, () => {
  console.log("Server laeuft auf Port 3000");
});
```

Threads

Non-Blocking I/O

- ▶ Asynchrone I/O (AIO)
- ▶ I/O-Multiplexing (auch bekannt als Event-Driven I/O)
- ▶ Non-Blocking Sockets
- ▶ Beispiel für NIO ist `java.nio.channels` (Beispiel Script)

Threads

Server

- ▶ Iterativ
- ▶ Nebenläufig
- ▶ Beispielimplementierungen im Script

Threads

Iterativer Server

- ▶ Einfache Implementierung
- ▶ Geringerer Ressourcenverbrauch
- ▶ Herausforderung : Skalierbarkeit über Kerne
- ▶ Herausforderung : Reaktionsfähigkeit

Threads

Nebenläufiger Server

- ▶ Reaktionsfähigkeit
- ▶ Skalierbarkeit über Kerne
- ▶ Herausforderung : Komplexität
- ▶ Herausforderung : Ressourcenverbrauch

Prozesse

Prozesse

- ▶ In VS große Rolle
- ▶ Prozesse äquivalent zum Task
- ▶ Task mobiler als Prozess

Prozesse

Probleme Skalierung

- ▶ Unterschiedliche Prozessorarchitekturen
- ▶ Betriebssystemabhängigkeiten
- ▶ Byte-Reihenfolge
- ▶ Speicherverwaltung
- ▶ Kommunikationsprotokolle
- ▶ Leistungsunterschiede
- ▶ Softwarebibliotheken

Prozesse

Strategien

- ▶ Abstraktion
- ▶ Plattformunabhängige Programmiersprachen und Laufzeitumgebungen
- ▶ Standardbibliotheken und Protokoll
- ▶ Cross-Kompilierung
- ▶ Automatisiertes Testen
- ▶ Leistungsunterschiede
- ▶ Softwarebibliotheken

Prozesse

Probleme

- ▶ Speicherzugriffsverletzungen
- ▶ Ressourcenkonflikte
- ▶ Nebenkanalangriffe
- ▶ Privilegienerweiterung
- ▶ Prozess-Interaktionen

Prozesse

Abstraktionen

- ▶ DOS
- ▶ NOS

Prozesse

Probleme Betrieb

- ▶ Lokalisierung des Prozess
- ▶ Kommunikation und Koordination
- ▶ Konsistenz und Zustand
- ▶ Fehlertoleranz

Prozesse

Halte-Problem

- ▶ Zeitüberschreitung (Timeouts)
- ▶ Überwachung und Fehlererkennung
- ▶ Begrenzung der Ressourcennutzung
- ▶ Deadlock-Erkennung und -Auflösung
- ▶ Modulare und robuste Software-Designs
- ▶ Testen und formale Verifikation
- ▶ Graceful Degradation und Selbstheilung

Virtualisierung

Motivation

- ▶ Isolation und Sicherheit
- ▶ Optimaler Ressourcen-Nutzung
- ▶ Flexibilität und Skalierbarkeit
- ▶ Logische Einheiten (VM)
- ▶ Einfach erstellt, gelöscht, migriert oder skaliert
- ▶ Schnelles Hinzufügen oder Entfernen von Ressourcen
- ▶ Einfache Verwaltung, Wartung und Testbarkeit

Virtualisierung

Geschichte

- ▶ Wurzel in den frühen Jahren der modernen Informatik (1960)
- ▶ Erste IBM Virtualisierungsplattform : CP-40-System
- ▶ Time-Sharing und Multi-User-Betriebssysteme
- ▶ Einführung von Netzwerken in den 1980 ermöglichte neue Verteilung
- ▶ 1990 WWW enabler für heutige Struktur und Kommunikationstechnologien
- ▶ Virtualisierung in den 2000er durch VMware und Xen
- ▶ Containertechnologien wie Docker in den 2010er

Virtualisierung

Fokus

- ▶ Hardware-Virtualisierung
- ▶ Betriebssystem-Virtualisierung
- ▶ Anwendungs-Virtualisierung
- ▶ Speicher- und Netzwerkvirtualisierung

Virtualisierung

Hardware-Virtualisierung

- ▶ Hardware in virtuelle Instanzen aufgeteilt
- ▶ Effiziente Nutzung von Ressourcen und die Isolierung
- ▶ Verschiedener Systeme und Anwendungen auf der gleichen physischen Hardware
- ▶ Nutzt Funktionen der CPU und anderer Komponenten für HW-Unterstützung
- ▶ Zentrale Herausforderung Leistung

Virtualisierung

Hardware-Virtualisierung Herausforderungen

- ▶ Teilung der Ressourcen und Isolation
- ▶ Management von Input/Output (I/O)-Operationen
Technologien wie I/O-Virtualisierung und Direct Memory Access (DMA)-Remapping sollen Effekte mindern
- ▶ Kompatibilität
Besonderer Fokus auf Hypervisor

Virtualisierung

Hypervisor

- ▶ Typ 1-Hypervisoren (auch Bare-Metal-Hypervisoren)
Typ 1-Hypervisoren sind VMware ESXi und Microsoft Hyper-V
- ▶ Typ 2-Hypervisoren
VMware Workstation und Oracle VirtualBox

Virtualisierung

Hardware-Virtualisierung Ansätze

- ▶ Vollvirtualisierung
- ▶ Paravirtualisierung

Virtualisierung

Betriebssystemvirtualisierung

- ▶ Logischen Einheiten Container
- ▶ Gleicher Kernel und gleiche Systembibliotheken
- ▶ Basis ist gleiche Betriebssystem API

Virtualisierung

Betriebssystemvirtualisierung Isolation

- ▶ Herausforderung Isolierung der Container
- ▶ Trennung nicht so stark wie bei VM
- ▶ Ansätze durch AppArmor, SELinux und Seccomp

Virtualisierung

Vor- und Nachteile

- ▶ V : Leistung
- ▶ V : Skalierbarkeit
- ▶ V : Kosteneffizienz
- ▶ N : Sicherheit
- ▶ N : Kompatibilität

Virtualisierung

Betriebssystemvirtualisierung Beispieltechnologien

- ▶ Kubernetes
- ▶ Docker
- ▶ OpenStack

Virtualisierung

Kubernetes

- ▶ Open-Source-Orchestrierungssystem/
Cluster-Management-System
- ▶ Automatisierung der Bereitstellung, Skalierung und
Verwaltung
- ▶ Ursprünglich von Google entwickelt
- ▶ Kubernetes unterstützt z.B. Containerd und CRI-O (nicht
Docker-Daemons, Container Runtime Interface (CRI)
unterstützt docker images)

Virtualisierung

Docker

- ▶ Open-Source-Plattform für die Containerisierung
- ▶ Der Ursprung vieler Entwicklungen
- ▶ Bietet eine leichtgewichtige Virtualisierung
- ▶ Docker-Container sind plattformübergreifend
- ▶ Der Begriff docker ist mehrdeutig (Docker-Daemon, Docker-CLI, Docker-Image)

Virtualisierung

Docker Alternativen

- ▶ Podman
- ▶ Buildah
- ▶ LXC (Linux Containers)
- ▶ rkt (ausgesprochen "Rocket" - obsolete)
- ▶ containerd
- ▶ CRI-O

Virtualisierung

OpenStack

- ▶ Open-Source-Cloud-Computing-Plattform, die Infrastruktur als Service (IaaS) bietet
- ▶ Eigene Cloud-Infrastruktur mit verschiedenen Komponenten
- ▶ Kann in kubernetes eingesetzt werden

Virtualisierung

Speichervirtualisierung

- ▶ Physische Speicherressourcen in einem logischen Pool
- ▶ Dynamisch und flexibel Nutzern zuweisbar
- ▶ Einfachere Verwaltung von Speicherressourcen

Virtualisierung

Speichervirtualisierung - SAN

- ▶ Storage Area Networks (SANs)
- ▶ SAN ist ein dediziertes Hochgeschwindigkeitsnetzwerk
- ▶ Einfachere Verwaltung von Speicherressourcen

Virtualisierung

Netzwerkvirtualisierung

- ▶ Physische Netzwerkressourcen in logische Einheiten abstrahiert
- ▶ Aufbau virtueller Netzwerke
- ▶ Beispiel für Netzwerkvirtualisierung ist SDN

Virtualisierung

Desktopvirtualisierung

- ▶ VDI-Lösungen wie VMware Horizon oder Windows 365
- ▶ Verbesserte Verwaltung und Wartung von Desktop-Betriebssystemen
- ▶ Desktops zentral verwaltet
- ▶ Bisher relativ hohe Kosten

Virtualisierung

Virtualisierung als Dienst

- ▶ Software as a Service (SaaS)
- ▶ Platform as a Service (PaaS)
- ▶ Infrastructure as a Service (IaaS)

Virtualisierung

Virtualisierung als Dienst - Vorteile

- ▶ Einfachere Bereitstellung und Skalierung
- ▶ Kosteneffizienz
- ▶ Fokus auf Kernkompetenzen
- ▶ Globale Präsenz und Leistung
- ▶ Erleichterte Integration und Zusammenarbeit
- ▶ Erhöhte Sicherheit und Compliance

Virtualisierung

Virtualisierung als Dienst - Bietet

- ▶ Investition
- ▶ Sicherheit und Datenschutz
- ▶ Optimierung der Anwendungsleistung
- ▶ Kundensupport und Service Level Agreements

Virtualisierung

Virtualisierung als Dienst - Schwierigkeiten

- ▶ Kosten
- ▶ Technische Kompatibilität
- ▶ Datenmigration
- ▶ Fehlende Regulierung

Virtualisierung

Virtualisierung als Dienst - Beispiel Netflix Deployment

- ▶ Frontend und API
- ▶ Microservices
- ▶ Container-Orchestrierung
- ▶ Datenbanken und Caching
- ▶ Datenspeicherung
- ▶ Content Delivery
- ▶ Big Data und Analyse
- ▶ Monitoring und Logging
- ▶ Sicherheit
- ▶ Automatisierung und Infrastruktur als Code
- ▶ CI/CD (Continuous Integration und Continuous Deployment)
- ▶ Resilienz und Fehlertoleranz

Virtualisierung

Virtualisierung als Dienst - Beispiel Netflix Software

- ▶ Netflix OSS (Open Source Software) ist eine Sammlung von Open-Source-Projekten
- ▶ Eureka : Ein Service-Discovery-System
- ▶ Hystrix : Eine Latenz- und Fehler-Toleranz-Bibliothek
- ▶ Chaos Monkey : Ein Tool zur Überprüfung der Fehlertoleranz
- ▶ Dynamite : Eine hochverfügbare, verteilte und skalierbare Datenbank-Engine

Virtualisierung

Virtualisierung als Dienst - Beispiel Netflix Software

- ▶ Cassandra : Eine hochverfügbare, verteilte NoSQL-Datenbank
- ▶ Spinnaker : Eine Multi-Cloud-Continuous-Delivery-Plattform
- ▶ Atlas : Ein skalierbares und erweiterbares Monitoring-System
- ▶ Lemur : Ein Tool zur Verwaltung von TLS-Zertifikaten
- ▶ Titus : Netflix' hauseigener Container-Orchestrierungs-Service
- ▶ Genie : Eine Plattform für die Verwaltung und Ausführung von Big Data-Jobs
- ▶ Weiter Client-Bibliotheken und SDKs

Cluster

Einleitung

- ▶ Gruppe von Computern oder Servern
- ▶ Hochverfügbarkeit, Fehlertoleranz und Leistungsverbesserung
- ▶ In der Regel homogene HW
- ▶ Bietet einheitliche Schnittstelle
- ▶ Einsatz : High-Performance-Computing, Web-Hosting, Datenbank-Management
- ▶ Arten (Grundlage) : Beowulf und Wolfpack

Cluster

Beowulf

- ▶ 1990er Jahren von Thomas Sterling und Donald Becker entwickelt
- ▶ Handelsüblicher Hardware und Open-Source-Software
- ▶ Hauptanwendung HPC
- ▶ Idee als Konkurrenz zu Großrechner
- ▶ Nutzen z.B. Message Passing Interface (MPI) oder Parallel Virtual Machine (PVM)
- ▶ kosteneffizient

Cluster

Wolfpack

- ▶ 1990er Jahren ursprünglich von Microsoft
- ▶ Teil der Windows NT Server-Produktlinie
- ▶ Wolfpack bietet Hochverfügbarkeit und Fehlertoleranz

Cluster

InfiniBand

- ▶ Konkurrenz zu klassischen Ethernet Strukturen
- ▶ Geringere Latenz und hohe Bandbreite (bis zu mehreren hundert Gigabit pro Sekunde)
- ▶ Von der InfiniBand Trade Association (IBTA) als Standard definiert
- ▶ Basis ist seriell, punkt-zu-punkt Kommunikationsprotokoll
- ▶ Beispiel für Einsatz : Frankfurter Börse

Grid

Einleitung

- ▶ Geografisch verteilte und heterogene Computerressourcen
- ▶ Unterschiedlichen Hardware- und Softwarekonfigurationen
- ▶ Ressourcen in der Regel autonom