Linux-Kernel: Architektur und Funktionsprinzipien

Einleitung: Dieses Dokument bietet eine detaillierte technische Einführung in den Linux-Kernel, indem es seine Architektur und fundamentalen Funktionsprinzipien erläutert. Es definiert den Kernel als die monolithische, zentrale Abstraktionsschicht zwischen Hardware und Anwendungen, die in der privilegierten Ebene (Ring 0) operiert. Im Fokus stehen die wichtigsten funktionalen Kernkomponenten wie Speicherverwaltung (Memory Management), Prozessmanagement (CFS Scheduler), das Virtual File System (VFS) und den Netzwerk-Stack. Weiterhin wird die strikte Trennung zwischen Kernelspace und Userspace dargelegt, insbesondere wie Anwendungen über den Systemaufrufmechanismus (Syscalls) kontrolliert vom Userspace in den Kernelspace wechseln, um privilegierte Operationen durchzuführen. Moderne Sicherheitsmechanismen wie LSM (SELinux/AppArmor) sowie Namespaces und cgroups werden als essenzielle Komponenten für Ressourcenisolation und Systemsicherheit vorgestellt.

1. Etymologie

Der Begriff *Kernel* (engl. „Kern") entstammt der Analogie zu natürlichen Systemen (Nusskern, Atomkern) und bezeichnet in der Systemarchitektur die innerste, essenzielle Komponente – im Gegensatz zu peripheren Schichten wie Shell oder Middleware. In der Betriebssystemtheorie wird der Begriff durch die Taxonomie von Mikro-, Monolith- und Hybrid-Kerneln präzisiert, wobei Linux als monolithischer Kernel mit modularer Erweiterbarkeit klassifiziert wird.

2. Definition und Konzept

Der Kernel bildet die zentrale Abstraktionsschicht zwischen Hardware-Ressourcen und Anwendungssoftware. Als monolithischer Kernel mit modularer Erweiterbarkeit implementiert Linux eine privilegierte Ausführungsumgebung (Ring 0) mit direktem Hardware-Zugriff und kontrolliertem Interface für unprivilegierte Prozesse (Ring 3).

3. Funktionale Kernkomponenten

Speicherverwaltung (Memory Management Unit)

Komponenten: Virtuelle Adressräume, Paging-Mechanismen, Copy-on-Write, Demand Paging (`/proc/meminfo` , `mmap(2)`)

> ***Erläuterung:*** Jeder Prozess erhält einen isolierten virtuellen Adressraum (typisch 48 Bit auf x86_64). Der Kernel teilt physischen RAM in Pages (4 KB) auf und mappt diese dynamisch. **Copy-on-Write** bedeutet: Bei `fork()` werden Speicherseiten zunächst nur als Referenz kopiert – erst bei Schreibzugriff erfolgt die tatsächliche Duplizierung (spart Ressourcen). **Demand Paging** lädt Programmcode erst bei Bedarf in den RAM statt beim Start komplett. `mmap()` erlaubt direktes Mapping von Dateien in den Speicher (z.B. für Shared Libraries).

Prozessmanagement (Process Scheduler)

Komponenten: Multitasking via Completely Fair Scheduler (CFS), Prozessdeskriptoren, Context Switching (`fork(2)` , `exec(2)`)

> ***Erläuterung:*** Der **CFS** (seit Kernel 2.6.23) verteilt CPU-Zeit nach einem fairen, zeitbasierten Modell: Prozesse mit geringer bisheriger CPU-Nutzung erhalten Vorrang vor solchen, die bereits viel Rechenzeit erhalten haben. Jeder Prozess wird durch einen **Prozessdeskriptor** (`task_struct`) repräsentiert, der essenzielle Informationen wie PID, Ausführungszustand, Speicher-Mappings und geöffnete Dateideskriptoren enthält. Ein **Context Switch** (5-10 µs) sichert die CPU-Register des aktuellen Prozesses und lädt die des nächsten. `fork()` erzeugt einen identischen Kind-Prozess (Copy-on-Write für Effizienz), während `exec()` das Programm im aktuellen Prozess durch ein neues ersetzt – die typische Kombination `fork()` → `exec()` ermöglicht das Starten neuer Programme.

Device-Subsystem

Komponenten: Unified Device Model, Character/Block Devices, Device Tree, dynamisches Device Management (`udev` , `/dev/*` , `/sys/class/*`)

> ***Erläuterung:*** **Character Devices** (z.B. `/dev/tty` , Tastatur) übertragen Daten zeichenweise ohne Buffer. **Block Devices** (z.B. `/dev/sda` , Festplatten) nutzen gepufferten Zugriff in Blöcken (512 B - 4 KB). `udev` erstellt Device-Nodes dynamisch beim Anstecken von Hardware (hot-plug). `/sys/class/` zeigt Gerätehierarchie und Parameter (z.B. `/sys/class/net/eth0/address` für MAC-Adresse).

Virtual File System (VFS)

Komponenten: Abstraktionsschicht für heterogene Dateisysteme (ext4, Btrfs, XFS), Superblock-/Inode-Strukturen, Page Cache

> ***Erläuterung:*** VFS bietet einheitliche Operationen (`open()` , `read()` , `write()`) unabhängig vom zugrundeliegenden Dateisystem. **Inodes** speichern Metadaten (Zugriffsrechte, Dateigröße, Zeitstempel), jedoch nicht den Dateinamen – dieser wird im zugehörigen Directory-Eintrag gespeichert. Der **Page Cache** hält häufig genutzte Dateibereiche im Arbeitsspeicher und beschleunigt so Zugriffe drastisch, ohne dass Anwendungen davon explizit Kenntnis haben müssen. Die virtuellen Dateisysteme `/proc` und `/sys` benötigen keinen Festplattenspeicher – sie repräsentieren Kernel-Datenstrukturen zur Laufzeit als lesbare Dateien und ermöglichen so Systemintrospektion.

Inter-Process Communication (IPC)

Komponenten: Signals, Pipes, Shared Memory, Message Queues, Sockets (`pipe(2)` , `shm_open(3)`)

> ***Erläuterung:*** **Signals** sind asynchrone Benachrichtigungen (z.B. `SIGTERM` zum Beenden). **Pipes** (`|` in Shell) verbinden stdout/stdin zweier Prozesse. **Shared Memory** ist der schnellste IPC-Mechanismus (direkter gemeinsamer RAM-Zugriff, kein Kopieren). **Sockets** ermöglichen Netzwerkkommunikation und lokale IPC via Unix Domain Sockets (`/tmp/mysql.sock`).

Netzwerk-Stack

Komponenten: OSI Layer 2–4 Implementation, Netfilter Framework, Socket-API (`socket(2)` , `bind(2)`)

> ***Erläuterung:*** Der Kernel implementiert den vollständigen TCP/IP-Stack: Ethernet-Frames werden auf Layer 2 verarbeitet, IP-Routing erfolgt auf Layer 3, und TCP/UDP-Verbindungen werden auf Layer 4 gemanagt. **Netfilter** bildet das Framework für Paketfilterung und Network Address Translation, wobei `iptables` oder `nftables` als Userspace-Frontends dienen. Programme nutzen die standardisierte Socket-API: `socket()` erstellt einen Kommunikations-Endpoint, `bind()` weist diesem eine lokale Adresse/Port zu, `listen()` / `accept()` ermöglichen Server-Funktionalität, während `connect()` für Client-Verbindungen verwendet wird.

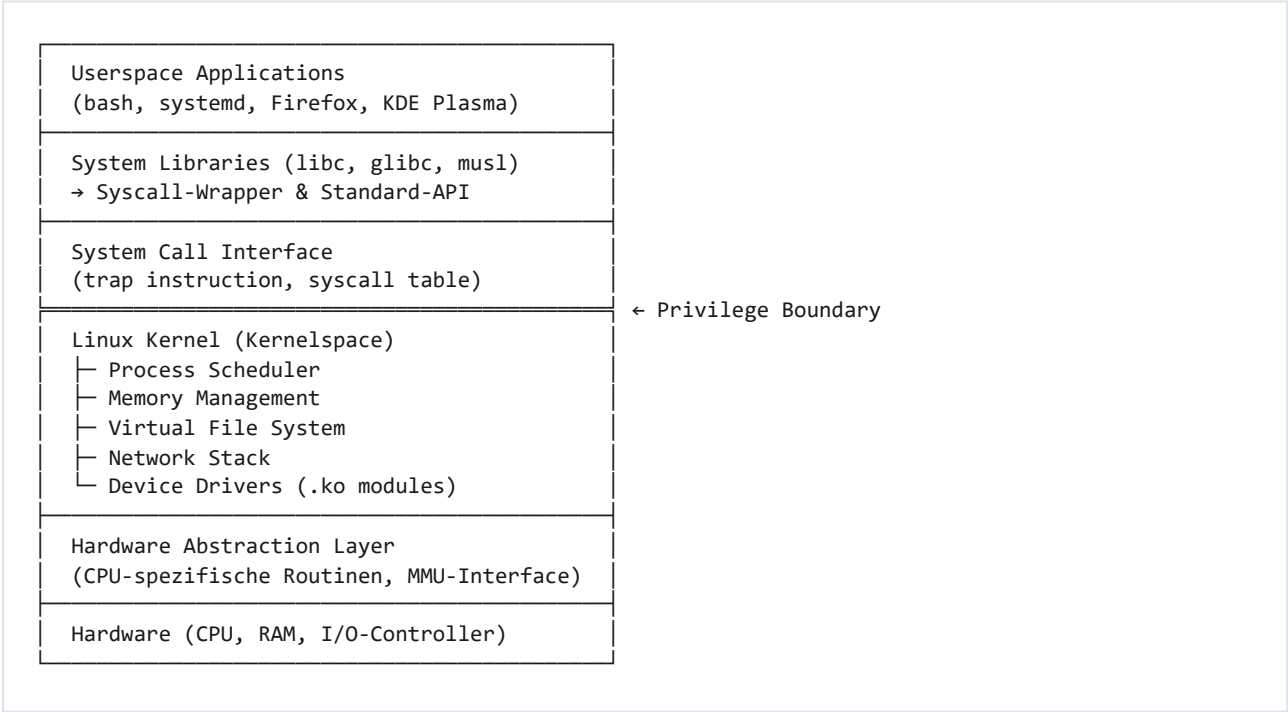
Sicherheitsarchitektur

Komponenten: DAC (Discretionary Access Control), Capabilities, LSM (Linux Security Modules): SELinux, AppArmor, Namespaces, cgroups

> ***Erläuterung:*** **DAC** (Discretionary Access Control) umfasst die klassischen UNIX-Rechte (read/write/execute für User/Group/Other). **Capabilities** zerlegen die monolithischen Root-Rechte in granulare Einzelberechtigungen – beispielsweise erlaubt `CAP_NET_BIND_SERVICE` das Binden an privilegierte Ports (<1024) ohne vollständige Root-Privilegien. **SELinux** und **AppArmor** implementieren Mandatory Access Control (MAC), wobei selbst der Root-User restriktiven Sicherheitsrichtlinien unterliegt. **Namespaces** isolieren Ressourcen-Sichten verschiedener Prozessgruppen (PID-, Network-, Mount-Namespaces) und bilden die Grundlage für Container-Technologien. **Control Groups (cgroups)** ermöglichen die hierarchische Limitierung und Priorisierung von Ressourcen (CPU, RAM, I/O) pro Prozessgruppe.

4. Systemarchitektur (Schichtenmodell)

Die folgende Darstellung zeigt den Aufbau eines typischen Linux-Systems als gestuftes Modell vom Userspace bis zur Hardware. Sie macht sichtbar, wo Grenzen verlaufen (Privilegien, Sicht auf Speicher, Schnittstellen).



Schicht	Aufgabe / Funktion	Linux-spezifische Details
Hardware	Physische Komponenten (CPU, RAM, Geräte)	x86_64, ARM, PCI, USB
Kernel	Herzstück, Kontrolle & Vermittlung	Monolithisch, Kernel Modules (.ko), Syscalls
– Speicherverwaltung	RAM, Paging, virtueller Speicher	mmap, Swapping, /proc/meminfo
– Prozessverwaltung	CPU-Zeit, Prozesse, Kommunikation	fork(), exec(), CFS Scheduler
– Geräteverwaltung	Einheitliche Schnittstelle über Treiber	/dev, udev, Device Nodes
– Dateisystem	Zugriff auf Daten, Einheitlichkeit	VFS, EXT4, Btrfs, /proc, /sys
– Syscalls	Schnittstelle für Programme	open(), write(), socket()
– Sicherheit	Schutz, Rechte, Isolation	root, POSIX-Rechte, SELinux, AppArmor, cgroups
Systembibliotheken	Vereinfachen Syscalls für Programme	glibc, musl
Anwendungen	Software im Userspace	KDE Plasma, GNOME, Firefox, bash

5. Kernel-/Userspace-Separation

Dimension	Kernelspace	Userspace
Privilegierung	Ring 0 (supervisor mode)	Ring 3 (user mode)
Speicherzugriff	Direkter physischer Zugriff	Virtueller Adressraum isoliert
Fehlerauswirkung	System-Crash (Kernel Panic)	Prozessabsturz (SIGSEGV)
Interface	Hardware-Register, Interrupts	Syscalls, `/proc` , `/sys`
Ausführungskontext	Interrupt-/Process-Context	Prozess-Thread-Context

6. Systemaufrufmechanismus (Syscalls)

Programme kommunizieren mit dem Kernel ausschließlich über definierte Systemaufrufe (→ `man 2 syscalls`). Diese bilden eine stabile ABI (Application Binary Interface): `open(2)`, `read(2)`, `write(2)`, `socket(2)`, `mmap(2)`, `clone(2)` etc.

Ablauf: Userspace → libc-Wrapper → Software-Interrupt/SYSCALL-Instruction → Kernel-Handler → Return

Detaillierter Ablauf eines Syscalls

1. Anwendung ruft libc-Funktion auf

Beispiel: `fd = open("/etc/passwd", O_RDONLY);` in C-Code

2. libc bereitet Syscall vor

Die Library-Funktion lädt Syscall-Nummer (z.B. `__NR_open` = 2 auf x86_64) in CPU-Register `rax`, Argumente in `rdi`, `rsi`, `rdx`

3. Privilegienwechsel via SYSCALL-Instruktion

Die CPU wechselt von Ring 3 (Userspace) nach Ring 0 (Kernelspace) und springt zur Kernel-Entry-Point-Adresse, die im MSR-Register (Model-Specific Register) hinterlegt ist

4. Kernel empfängt Aufruf

Der Syscall-Dispatcher (`entry_SYSCALL_64`) validiert die Syscall-Nummer und ruft den entsprechenden Kernel-Handler auf (z.B. `do_sys_open()`)

5. Handler führt Kernel-Operation aus

Der Handler prüft Berechtigungen, alloziert einen Dateideskriptor, interagiert mit dem VFS-Subsystem und gibt den File-Descriptor zurück

6. Rückkehr in Userspace

Der Kernel platziert den Rückgabewert in Register `rax`, wechselt via `SYSRET`-Instruktion zurück zu Ring 3, und die Anwendung setzt ihre Ausführung fort

Praktisches Beispiel beobachten

Wenn man den Befehl

```
strace -e open cat /etc/hostname
```

ausführt, kann man sehen, welche Systemaufrufe das Programm `cat` macht. Ein Systemaufruf (syscall) ist die Stelle, an der ein normales Programm den Kernel um Hilfe bittet, etwas zu tun, das nur das Betriebssystem selbst darf – zum Beispiel eine Datei öffnen.

Die Ausgabe

```
open("/etc/hostname", O_RDONLY) = 3
```

bedeutet: Das Programm `cat` ruft den Systemaufruf `open()` auf, um die Datei `/etc/hostname` im Lesemodus (`O_RDONLY`) zu öffnen. Der Kernel bestätigt die erfolgreiche Operation, indem er den **Dateideskriptor 3** zurückgibt – eine ganzzahlige Handle-Nummer, über die das Programm künftig auf diese geöffnete Datei zugreift.

Der Dateideskriptor **3** fungiert somit als interner Identifikator für diese Datei im Kontext des `cat` -Prozesses. Anschließend liest `cat` die Daten aus FD 3 und schreibt sie mittels `write()` auf FD 1 (Standardausgabe, typischerweise das Terminal). Dieses Beispiel mit `strace` veranschaulicht die praktische Interaktion zwischen Anwendungen und Kernel über die Syscall-Schnittstelle.

Performance-Overhead: Ein Syscall kostet ~100-200 ns (Context Switch, Sicherheitsprüfungen). Daher puffern Libraries Operationen (z.B. `stdio.h` sammelt `write()` -Aufrufe).

Sicherheitsaspekt: Kernel validiert alle Parameter (z.B. prüft Pointer-Adressen auf Userspace-Range), um Exploits zu verhindern.

Architekturparadigma: Monolithischer Kernel mit modularer Erweiterbarkeit

Lizenz: GNU General Public License v2 (GPLv2)

Entwicklungsmodell: Open-Source, community-driven (kernel.org)