

Betriebssysteme

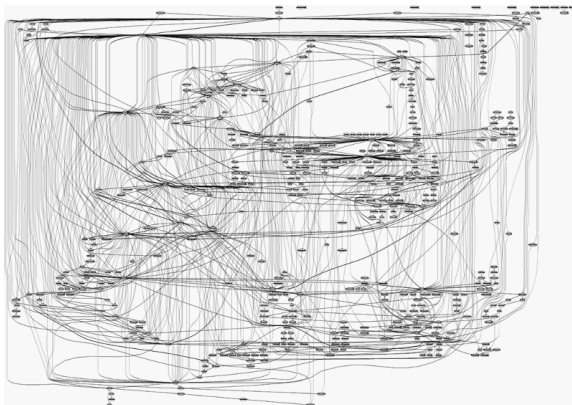
Robert Kaiser

(HTTP: <http://www.cs.hs-rm.de/~kaiser>

E-Mail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

2. Betriebssystemstrukturen



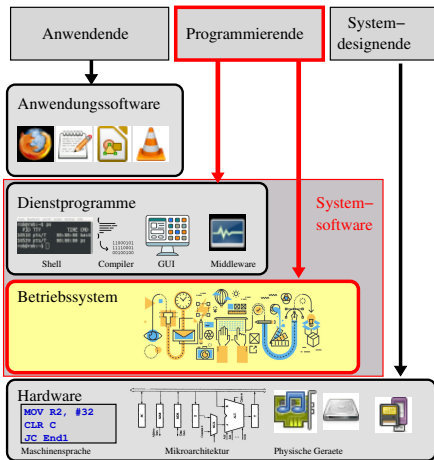
<https://www.freecodecamp.org/news/building-and-installing-the-latest-linux-kernel-from-source-6d8df5345980/>

Betriebssystemstrukturen



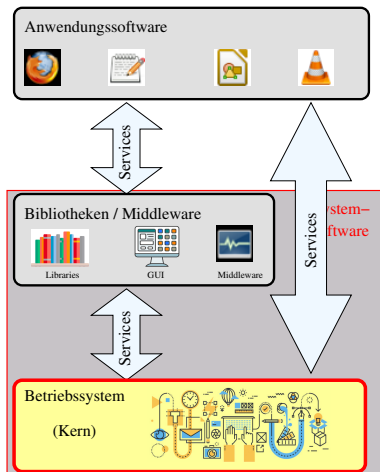
- ① Betriebsmodi
- ② Monolithische Systeme
- ③ Client / Server-Strukturen (Mikrokerne)
- ④ Virtuelle Maschinen
- ⑤ Zusammenfassung

Benutzungs- und Kernmodus



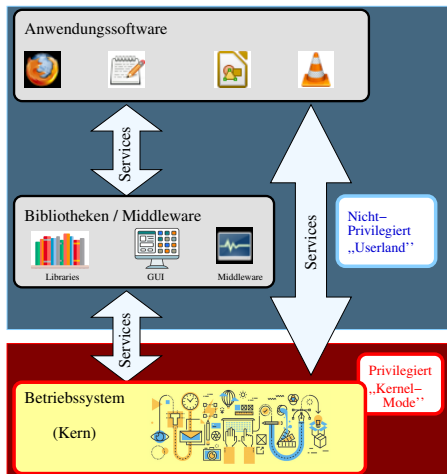
- Systemsoftware bietet „Dienste“ durch:
 - ▶ Bibliotheken / Middleware
 - ▶ Kern-Dienste (Systemcalls)
- Rechnerarchitektur besitzt (mind.) 2 Betriebsmodi:
 - ▶ Nicht-privilegiert: Benutzungsmodus
 - ▶ Privilegiert: Kernmodus
- Übergang durch Systemaufruf (= „TRAP“-Befehl)
- Systemaufruf-Schnittstelle (= „system call interface“)

Benutzungs- und Kernmodus



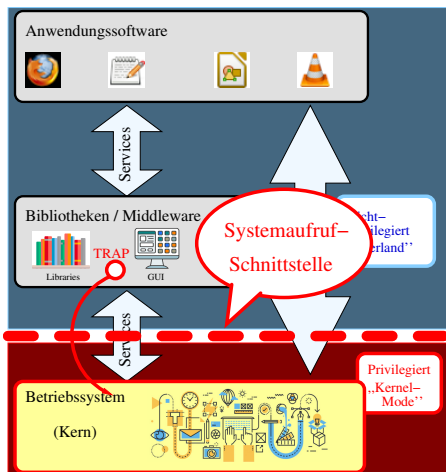
- Systemsoftware bietet „Dienste“ durch:
 - ▶ Bibliotheken / Middleware
 - ▶ Kern-Dienste (Systemcalls)
- Rechnerarchitektur besitzt (mind.) 2 Betriebsmodi:
 - ▶ Nicht-privilegiert: Benutzungsmodus
 - ▶ Privilegiert: Kernmodus
- Übergang durch Systemaufruf (= „TRAP“-Befehl)
- Systemaufruf-Schnittstelle (= „system call interface“)

Benutzungs- und Kernmodus



- Systemsoftware bietet „Dienste“ durch:
 - ▶ Bibliotheken / Middleware
 - ▶ Kern-Dienste (Systemcalls)
- Rechnerarchitektur besitzt (mind.) 2 Betriebsmodi:
 - ▶ Nicht-privilegiert: Benutzungsmodus
 - ▶ Privilegiert: Kernmodus
- Übergang durch Systemaufruf (= „TRAP“-Befehl)
- Systemaufruf-Schnittstelle (= „system call interface“)

Benutzungs- und Kernmodus

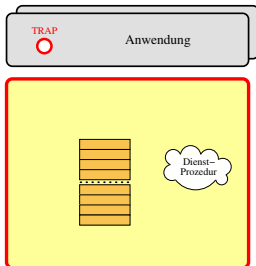


- Systemsoftware bietet „Dienste“ durch:
 - ▶ Bibliotheken / Middleware
 - ▶ Kern-Dienste (Systemcalls)
- Rechnerarchitektur besitzt (mind.) 2 Betriebsmodi:
 - ▶ Nicht-privilegiert: Benutzungsmodus
 - ▶ Privilegiert: Kernmodus
- Übergang durch Systemaufruf (= „TRAP“-Befehl)
- Systemaufruf-Schnittstelle (= „system call interface“)

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

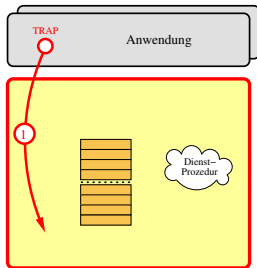
Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

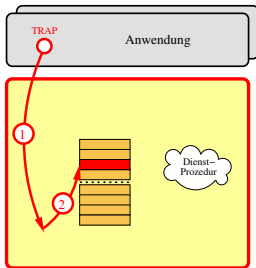
Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

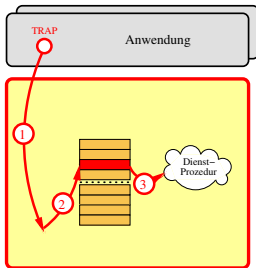
Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

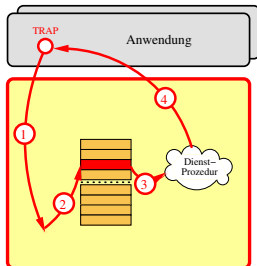
Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

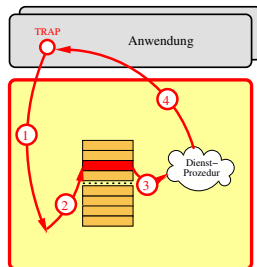
Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

Durchführung eines Kernaufrufs



Benutzungsprogramme und
Betriebssystem befinden sich im
Arbeitsspeicher



Anwendungen laufen im
Benutzungsmodus

Das Betriebssystem läuft im
Kernmodus

- 1 Anwendungsprogramm springt über TRAP in den Kern und führt den Code selbst aus.
- 2 BS Code bestimmt die Nummer des angeforderten Dienstes.
- 3 BS Code lokalisiert Prozedur-Code für Systemaufruf und ruft sie auf.
- 4 Kontrolle wird an das Anwendungsprogramm zurückgegeben.

**Wichtig: Kern selbst ist passiv (Menge von
Datenstrukturen und Prozeduren)**

Systemaufruf-Beispiel



- „Stub“ Code der Funktion `open()` in der Linux C-Bibliothek ¹

```
int open(char *name, int flags, mode_t mode)
```

```
...                               __libc_open:
int fd;                           push    %ebx
fd = open("myfile",              mov     0x10(%esp),%edx
        0_RDWR|O_CREAT,         mov     0xc(%esp),%ecx
        0777);                  mov     0x8(%esp),%ebx
                                mov     $0x5,%eax
                                int     $0x80
if(fd < 0)                        pop     %ebx
    perror("open()");           cmp     $0xffff001,%eax
...                              jae     __syscall_error
^^I^^I^^I                      ret
                                ^^I^^I^^I
```

¹ Hinweis: UNIX Manual

- Systemaufrufe in Manual Section 2 (z.B.: `open(2)`, `close(2)`, `read(2)`, `write(2)`, ...)
- C-Bibliotheksfunktionen in Manual Section 3 (z.B.: `printf(3)`, `scanf(3)`, `malloc(3)`, ...)

Systemaufruf-Beispiel



- „Stub“ Code der Funktion `open()` in der Linux C-Bibliothek ¹

```
int open(char *name, int flags, mode_t mode)
```

```
...  
int fd;  
fd = open("myfile",  
          O_RDWR|O_CREAT,  
          0777);  
if(fd < 0)  
    perror("open()");  
...  
~~I~~I~~I
```

```
__libc_open:  
    push    %ebx  
    mov     0x10(%esp),%edx  
    mov     0xc(%esp),%ecx  
    mov     0x8(%esp),%ebx  
    mov     $0x5,%eax  
    int     $0x80  
    pop     %ebx  
    cmp     $0xfffff001,%eax  
    jae     __syscall_error  
    ret  
~~I~~I~~I
```

Argumente (vom
Stack) in Register

¹ Hinweis: UNIX Manual

- Systemaufrufe in Manual Section 2 (z.B.: `open(2)`, `close(2)`, `read(2)`, `write(2)`, ...)
- C-Bibliotheksfunktionen in Manual Section 3 (z.B.: `printf(3)`, `scanf(3)`, `malloc(3)`, ...)

Systemaufruf-Beispiel



- „Stub“ Code der Funktion `open()` in der Linux C-Bibliothek ¹

```
int open(char *name, int flags, mode_t mode)
```

```
...
int fd;
fd = open("myfile",
          O_RDWR|O_CREAT,
          0777);
if(fd < 0)
    perror("open()");
...
^^I^^I^^I
```

```
__libc_open:
    push    %ebx
    mov     0x10(%esp),%edx
    mov     0xc(%esp),%ecx
    mov     0x8(%esp),%ebx
    mov     $0x5,%eax
    int     $0x80
    pop     %ebx
    cmp     $0xfffff001,%eax
    jae     __syscall_error
    ret
^^I^^I^^I
```

Argumente (vom
Stack) in Register

Funktionsnummer:
5 = `open()`

¹ Hinweis: UNIX Manual

- Systemaufrufe in Manual Section 2 (z.B.: `open(2)`, `close(2)`, `read(2)`, `write(2)`, ...)
- C-Bibliotheksfunktionen in Manual Section 3 (z.B.: `printf(3)`, `scanf(3)`, `malloc(3)`, ...)

Systemaufruf-Beispiel



- „Stub“ Code der Funktion `open()` in der Linux C-Bibliothek ¹

```
int open(char *name, int flags, mode_t mode)
```

```
...
int fd;
fd = open("myfile",
          O_RDWR|O_CREAT,
          0777);
if(fd < 0)
    perror("open()");
...
^^I^^I^^I
```

```
__libc_open:
    push    %ebx
    mov     0x10(%esp),%edx
    mov     0xc(%esp),%ecx
    mov     0x8(%esp),%ebx
    mov     $0x5,%eax
    int     $0x80
    pop     %ebx
    cmp     $0xffffffff,%eax
    jae     __syscall_TRAP-Befehl
    ret
^^I^^I^^I
```

Argumente (vom Stack) in Register

Funktionsnummer:
5 = `open()`

__syscall_TRAP-Befehl

¹ Hinweis: UNIX Manual

- Systemaufrufe in Manual Section 2 (z.B.: `open(2)`, `close(2)`, `read(2)`, `write(2)`, ...)
- C-Bibliotheksfunktionen in Manual Section 3 (z.B.: `printf(3)`, `scanf(3)`, `malloc(3)`, ...)

Betriebsmodi




Aus Hardware-Sicht: Betriebsarten (Modi) des Prozessors

- Bei den meisten Architekturen: zwei Stufen
(Vier bei x86, die meisten BS nutzen aber nur zwei davon)
 - ▶ privilegiert
 - ▶ nicht-privilegiert
- Privilegierter Modus erlaubt Zugriff / Manipulation der „Maschinenkonfiguration“
 - ▶ Sperren von Unterbrechungen
 - ▶ Zugriff auf Speicherverwaltungs-Hardware
 - ▶ Privilegierte Maschinenbefehle
- Übergang in höhere Stufe durch Ausnahmebedingungen („Exceptions“)
 - ▶ Unterbrechungen („Interrupts“)
 - ▶ Explizite TRAP-Befehle
 - ▶ Schutzverletzungen („Faults“)
- Rückkehr zu niedrigerer Stufe durch spezielle Maschinenbefehle

Betriebsmodi (3)



Aus Betriebssystemsicht

- Im Benutzungsmodus (= nicht-privilegierten Modus):
 - ▶ Beschränkter Zugriff auf Betriebsmittel
 - ▶ Zugriff auf nicht-zugeteilte BM² löst Fault aus
 - ▶ Unerlaubte Operation³ löst Fault aus
 - ▶ Systemcall (= expliziter TRAP-Befehl) löst TRAP aus
- Im Kernmodus (= privilegierten Modus):
 - ▶ Uneingeschränkter Zugriff auf alle Betriebsmittel
(→ Ist auch notwendig zu deren Verwaltung)
 - ▶ Unerlaubte Operation oder Schutzverletzung im Kernmodus ⇒  !


² „Schutzverletzung“

³ z.B. Division durch Null, illegaler Befehl, alignment-Fehler...

Betriebsmodi (3)



Aus Betriebssystemsicht

- Im Benutzungsmodus (= nicht-privilegierten Modus):
 - ▶ Beschränkter Zugriff auf Betriebsmittel
 - ▶ Zugriff auf nicht-zugeteilte BM² löst Fault aus
 - ▶ Unerlaubte Operation³ löst Fault aus
 - ▶ Systemcall (= expliziter TRAP-Befehl) löst TRAP aus
- Im Kernmodus (= privilegierten Modus):
 - ▶ Uneingeschränkter Zugriff auf alle Betriebsmittel
(→ Ist auch notwendig zu deren Verwaltung)
 - ▶ Unerlaubte Operation oder Schutzverletzung im Kernmodus ⇒  !



⇒ **BS-Kern-Code muss fehlerfrei sein!**



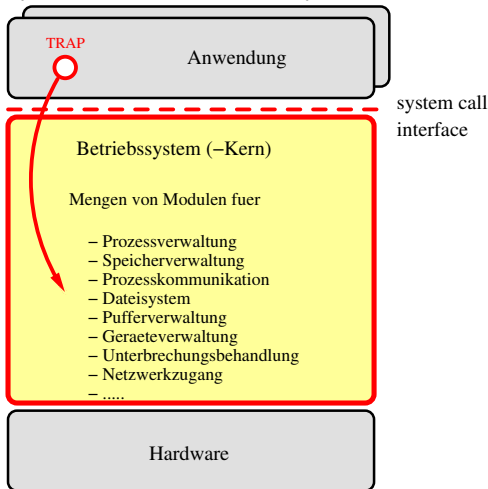
² „Schutzverletzung“

³ z.B. Division durch Null, illegaler Befehl, alignment-Fehler...

Monolithische Systeme



Vorwiegende Struktur aller kommerzieller Betriebssysteme (z.B. UNIX, Windows)

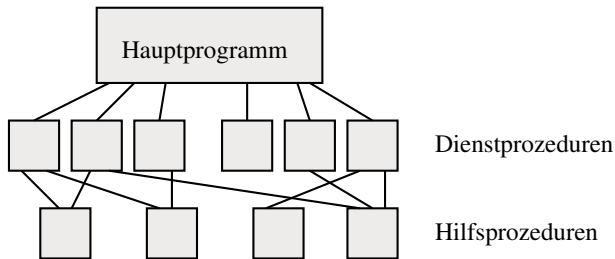


Durch Binder zu einem
„Klumpen“ zusammengebunden
(bis auf dyn. ladbare
Kernmodule)

Einfaches Strukturmodell



Innere Struktur eines monolithischen BS:

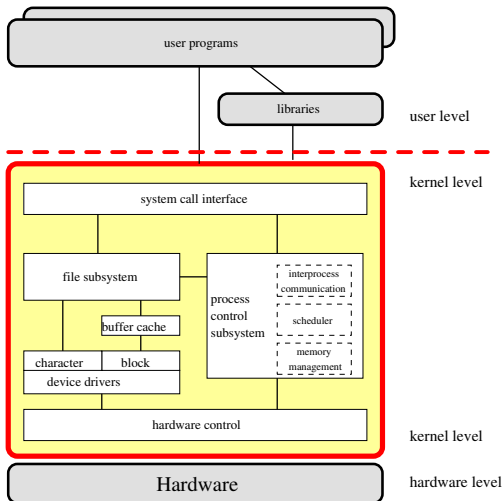


Da der Betriebssystemkern passiv ist und der Code aus einer Menge von Prozeduren besteht, heißt ein solches Betriebssystem auch prozedurorientiert.

Beispiel: UNIX

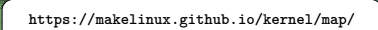


Blockdiagramm des Systemkerns⁴

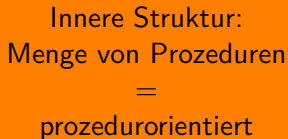


⁴aus [Bach]: The Design of the UNIX Operating System

216340



2636 ■



<https://makelinux.github.io/kernel/map/>

2636 ■



© 2007 - 2010 Constantine Shulyupin www.MakeLinux.net/kernel/mmap

Beispiel: UNIX Codeumfang



Der UNIX-Betriebssystemkern

- monolithisch, aber portierbar
- Beispiel: 4.3BSD UNIX Kern (1987)
 - Lines of Code 116.470 (nicht mehr!)
 - C-Anteil 97,1%
 - maschinenunabhängig 41,5%
 - maschinenabhängig 58,5%
 - davon Gerätetreiber 35,5%
 - Netzwerktreiber 14,8%

Neben dem Betriebssystemkern wird ein Großteil der UNIX-Systemfunktionalität durch sogenannte Dämon-Prozesse erbracht.

- Beispiel: Linux Kern SLOC (ohne Leerzeilen, ohne Kommentarzeilen):
 - Linux 1.0.0 (1994) 176.250
 - Linux 2.2.0 (1999) 1.800.847
 - Linux 2.6.0 (2003) 5.929.913
 - Linux 3.2 (2012) 14.998.651
- Windows Server 2003 (Gesamtsystem) ca. 50 Mio Zeilen

Vergleich zu sonstigen Codegrößen⁶



Project	No. of Files	eLOC ⁵
Linux Kern 2.6.17	15.995	4.142.481
Firefox 1.5.0.2	10.970	2.172.520
MySQL 5.0.25	1973	894.768
PHP 5.1.6	1316	479.892
Apache Http 2.0.x	275	89.967

⁵The effective lines of code (eLOC) are measured using the following method:

1. Get the number of lines of code
2. Subtract whitespace lines
3. Subtract comment lines
4. Subtract the lines that contains only block constructs

⁶[http:](http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm)

[//msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm](http://msquaredtechnologies.com/m2rsm/rsm_software_project_metrics.htm)



Client / Server-Strukturen (Mikrokern)



- Problem monolithischer Systeme: Kern wird immer **umfangreicher und komplexer**, damit zwangsläufig auch **fehlerträchtiger**.
- Aller Code, der im privilegierten Modus läuft, hat Zugriff auf alle Betriebsmittel und zählt damit immer zur „**Trusted Code Base**“.
- Nicht alle Anwendungen benötigen wirklich alle Dienste, die ein Kern anbietet
- Art und Anzahl der Dienste werden aber durch den Kern fest vorgegeben
- **Mikrokern**-Ansatz: Alle Dienste, deren Funktion technisch auch ohne privilegierte Operationen realisiert werden kann, werden aus dem Kern ausgelagert

Mikrokern-Ansatz

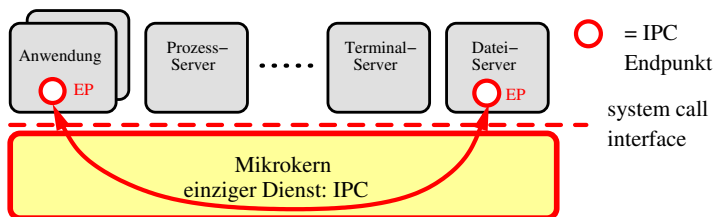


- Dienste wie Dateisystem, Netzwerkprotokolle, Speicherverwaltung, Prozesssteuerung, sogar Gerätetreiber müssen nicht zwangsläufig im Kern angesiedelt sein.
- Auslagerung großer Teile der Funktionalität eines BS-Kerns in Programme auf Anwendungsebene.
- „Server“-Prozesse, die wie Anwendungsprogramme ohne besondere Privilegien arbeiten, bieten diese Dienste (*Services*) an.
- Als „gewöhnliche“ Anwendungen haben Server nur jeweils Zugriff auf die ihnen zugewiesenen Betriebsmittel.
- Übrig bleibt ein **minimaler** Kern, als *Mikrokern* bezeichnet
- Dieser bietet nur noch Dienste zur Kommunikation zwischen Klienten (Anwendungen) und Servern untereinander an.

Mikrokern



• Client / Server Architektur



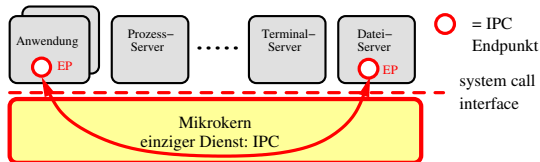
- Dienste werden durch Nachrichten per **Interprozesskommunikation** (*inter process communication* – IPC) von Servern angefordert. (*send & receive*)
- Server liefern Dienste ebenfalls mit IPC-Nachrichten. (*reply & wait*)
- Weitere Konzepte im μ Kern: Adressräume und Threads

Client / Server-Struktur (2)



Vorteile:

- **Isolation** der „Systemteile“ gegeneinander
z.B.: Ausfall eines Servers betrifft nur dessen Klienten
→ Klienten können ihre „Trusted Code Base“ feingranular auswählen
- **Erweiterbarkeit**, Anpassungsfähigkeit und flexible Konfigurierbarkeit
z.B.: Mehrere unterschiedliche, sogar konkurrierende Dienste können gleichzeitig betrieben werden.
- **Nachrichtenbasiert**: Prinzipielle Möglichkeit, Klienten und Server transparent auf verschiedenen Knoten eines Verteilten System zu betreiben.



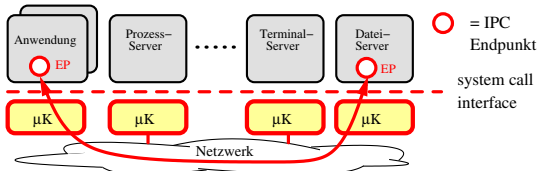
Ein Betriebssystem, das auf einem über Nachrichten realisierten Beauftragungsprinzip beruht, heißt auch nachrichtenorientiert. Nachrichtenorientiertheit und Prozedurorientiertheit sind funktional gleichwertig, nachrichtenorientierte Systeme leiden aber häufig unter Ineffizienz.



Client / Server-Struktur (2)

Vorteile:

- **Isolation** der „Systemteile“ gegeneinander
z.B.: Ausfall eines Servers betrifft nur dessen Klienten
→ Klienten können ihre „Trusted Code Base“ feingranular auswählen
- **Erweiterbarkeit**, Anpassungsfähigkeit und flexible Konfigurierbarkeit
z.B.: Mehrere unterschiedliche, sogar konkurrierende Dienste können gleichzeitig betrieben werden.
- **Nachrichtenbasiert**: Prinzipielle Möglichkeit, Klienten und Server transparent auf verschiedenen Knoten eines Verteilten System zu betreiben.



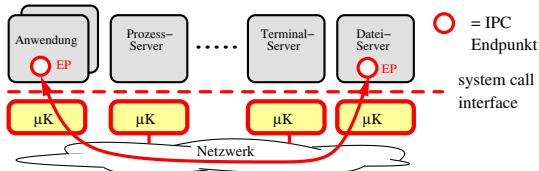
Ein Betriebssystem, das auf einem über Nachrichten realisierten Beauftragungsprinzip beruht, heißt auch nachrichtenorientiert. Nachrichtenorientiertheit und Prozedurorientiertheit sind funktional gleichwertig, nachrichtenorientierte Systeme leiden aber häufig unter Ineffizienz.

Client / Server-Struktur (2)



Vorteile:

- **Isolation** der „Systemteile“ gegeneinander
z.B.: Ausfall eines Servers betrifft nur dessen Klienten
→ Klienten können ihre „Trusted Code Base“ feingranular auswählen
- **Erweiterbarkeit**, Anpassungsfähigkeit und flexible Konfigurierbarkeit
z.B.: Mehrere unterschiedliche, sogar konkurrierende Dienste können gleichzeitig betrieben werden.
- **Nachrichtenbasiert**: Prinzipielle Möglichkeit, Klienten und Server transparent auf verschiedenen Knoten eines Verteilten System zu betreiben.



Ein Betriebssystem, das auf einem über Nachrichten realisierten Beauftragungsprinzip beruht, heißt auch nachrichtenorientiert. Nachrichtenorientiertheit und Prozedurorientiertheit sind funktional gleichwertig, nachrichtenorientierte Systeme leiden aber häufig unter Ineffizienz.

Policy and Mechanism



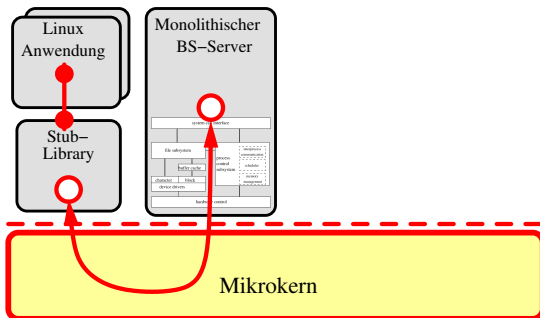
- Kriterium für auszulagernde Dienste: **Trennung von Strategie und Mechanismus** („*separation of policy and mechanism*“)
- Beispiel: Speicherverwaltung:
 - ▶ **Strategie** (*policy*): Zuteilung von Speichersegmenten an Prozesse
 - ▶ **Mechanismus** (*mechanism*): Konfiguration der Hardware-Speicherverwaltungseinheit (*Memory Management Unit* – MMU)
- μ Kern-Implementierung:
 - ▶ Der IPC-Dienst überträgt neben Daten auch Rechte (→ z.B. Zugriffsrechte auf Speicherbereiche)
 - ▶ Wird solch ein Zugriffsrecht übertragen, so konfiguriert der μ Kern bei der Übertragung die Hardware entsprechend.

⇒ Strategie im Server, Mechanismus im μ Kern.
- Ein μ Kern sollte klein und wenig komplex sein
- Das ist jedoch kein hinreichendes Kriterium (**Mikrokern \neq kleiner Kern!**)
- Entscheidend ist die (möglichst weitgehende) Policy-Freiheit

Single Server



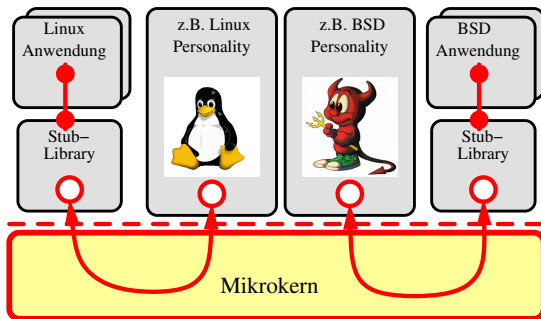
- Ansatz: Bestehendes, monolithisches BS in Server umwandeln
→ („OS personality“)
- Anwendungen finden die gleichen Dienste vor, können unverändert bestehen bleiben
- Vorteil: Mehrere Betriebssysteme in einem Rechner
- Nachteil: Große Trusted Code Base (...und Performance-Verlust)



Single Server



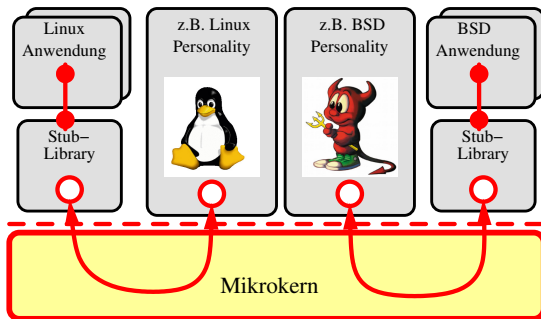
- Ansatz: Bestehendes, monolithisches BS in Server umwandeln
→ („OS personality“)
- Anwendungen finden die gleichen Dienste vor, können unverändert bestehen bleiben
- Vorteil: Mehrere Betriebssysteme in einem Rechner
- Nachteil: Große Trusted Code Base (...und Performance-Verlust)



Single Server



- Ansatz: Bestehendes, monolithisches BS in Server umwandeln
→ („OS personality“)
- Anwendungen finden die gleichen Dienste vor, können unverändert bestehen bleiben
- Vorteil: Mehrere Betriebssysteme in einem Rechner
- Nachteil: Große Trusted Code Base (...und Performance-Verlust)



Beispiele



Forschung

- Minix / Minix3 (VU Amsterdam)
- Singularity (Microsoft Research)
- EROS/CoyotOS (Johns Hopkins University)
- L4 Microkernel Familie
 - ▶ Ursprünglich: Jochen Liedtke, GMD
 - ▶ Weiterentwicklungen:
 - ★ Uni Karlsruhe: L4Ka, Pistachio
 - ★ TU Dresden: Fiasco, Nova Hypervisor
 - ★ UNSW Sydney: seL4

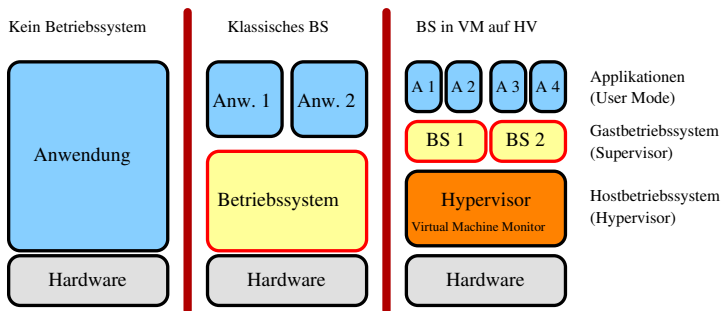
Kommerziell

- QNX Neutrino (Blackberry QNX)
- Chorus OS (Chorus Systems)
- PikeOS (SYSGO AG)

Virtuelle Maschinen



- Trennen der Funktionen „Mehrprogrammbetrieb“ und „erweiterte Maschine“
- Virtualisierung durch „*Virtual Machine Monitor*“ (auch: „*Hypervisor*“)
- virtuelle Maschinen als mehr oder weniger identische Kopien der unterliegenden Hardware
- In jeder virtuellen Maschine: übliches Betriebssystem.



Beispiel: VM/370

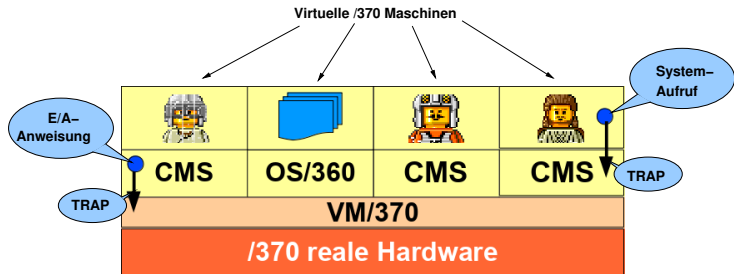


- **1970:** Das offizielle IBM-Produkt für Timesharing-Betrieb der /360, TSS/360, kam zu spät, war zu groß und zu langsam.
- In der Zwischenzeit: IBM Scientific Center Cambridge, Mass. Eigenentwicklung, wurde als Produkt (ursprünglich CP/CMS) akzeptiert, erlangte als VM/370 weite Verbreitung.
- Unterste Ebene: virtuelle Maschinen als identische Kopien der unterliegenden Hardware mit Nachbildung von Anwendungs- und Supervisor-Modi, I/O, Unterbrechungen,
- Simulation mehrerer /370 Rechner.
- Effizient durch gegebene „Virtualisierbarkeit“ der /370 Architektur.

Beispiel: VM/370 (2)



- Betriebssysteme in virtuellen Maschinen: z.B. ein Stapelverarbeitungssystem (OS/360) und eine Menge von Einbenutzer-Dialogsystemen (CMS, Conversational Monitor System) gleichzeitig möglich.



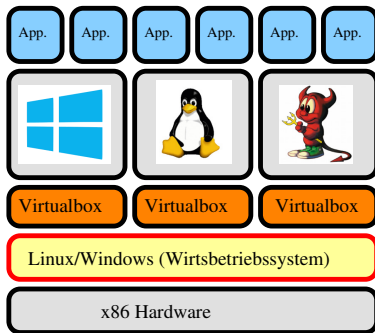
- Heute: z/VM: erlaubt z.B. 100 unabhängige Linux-Systeme auf einem IBM Mainframe.

Virtualisierbarkeit



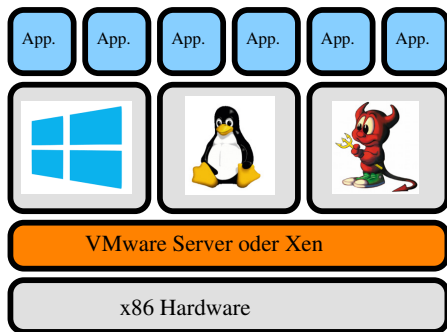
- Anforderung: **Identisches Verhalten der VM.**
- Ein ausgeführtes Programm kann nicht feststellen, ob es von einer VM oder einer realen Maschine ausgeführt wird.
- Möglichkeiten dazu;
 - ▶ **Emulation:** Komplette Nachbildung der Hardware in Software
 - Ineffizient!
Beispiele: Bochs (x86), JVM
 - ▶ **Virtualisierung:** Nur ein geringer Teil der Befehle muss emuliert werden, die meisten Befehle werden von der realen Hardware ausgeführt.
 - Annähernd keine Effizienzeinbußen
 - Voraussetzung: Architektur muss „virtualisierbar“ sein.
(Der x86 war das zunächst nicht!)
Beispiele: VM/370 (s.o.), Qemu, VirtualBox, VMware
 - ▶ **Paravirtualisierung:** Falls nicht virtualisierbar: Privilegierte Befehle des Gast-BS durch „Hypercalls“ (= Aufrufe in den Hypervisor) ersetzen.
 - Effizienz wie Virtualisierung (u.U. sogar noch besser)
 - Nachteil: Quellcode des Gast-BS muss angepasst werden.
 - Beispiele: Xen, KVM, OpenVZ, Hyper-V

Beispiel: VMware Workstation, Virtualbox



- Erlaubt beliebige Betriebssysteme für x86-Architektur auf Linux oder Windows
- Jedes Gastbetriebssystem kann abstürzen, ohne den Rest zu beeinflussen

Beispiel: VMware Server, Xen



- Xen: Paravirtualisierung: Gastssysteme müssen angepasst werden (Quellcode Voraussetzung)
- VMware Server: klassischer VM Monitor (eigentlich: „JIT-Paravirtualisierung“)

Zusammenfassung



- ❶ Grundverständnis einer Betriebssystemschnittstelle
- ❷ Strukturierungsprinzipien von Betriebssystemen:
 - ▶ Monolithische Struktur
 - ▶ Client/Server-Struktur (Mikrokern)
 - ▶ Virtuelle Maschinen