Betriebssysteme und Netzwerke Vorlesung 3

Artur Andrzejak

Wiederholung Vorlesung 2

Zwei Aufgaben eines BS?

Cache?

Welche CPU Register gibt es?

Call Stack *oder* Cow Steak?

Ablauf Systemaufruf?

Welche Daten findet man auf dem Stack?

Modi eines CPU? Bei IA32?

Umfragen: https://pingo.coactum.de/301541

Shell-Programmierung

Bemerkung zum Stoff der Vorlesung

Hauptteil

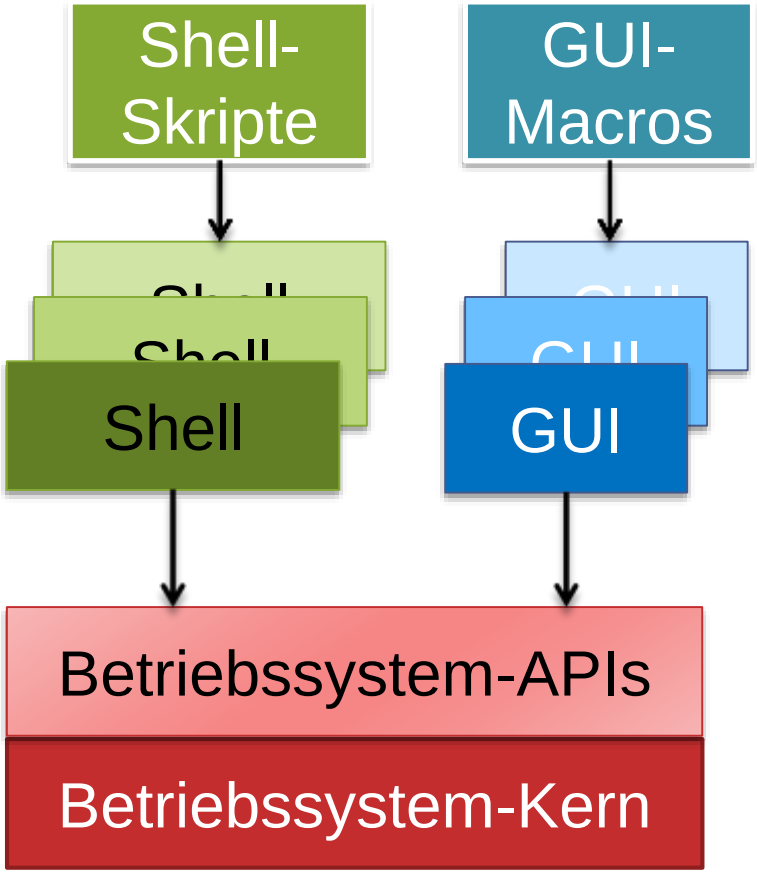
- Konzepte, Prinzipien, Algorithmen zum Bau der Betriebssysteme
  - Ca. 90%
- Selten direkt verwendbar
  - Nur wenn Sie Linux / BS mitentwickeln ☺
- Indirekt häufig verwendbar
  - z.B. Probleme der Ressourcenzuteilung treten bei Multithreading-Anwendungen auf

Komplementärteil

- Konzepte und Wissen zur Verwendung der BS
  - Ca. 10%
- Direkt verwendbar, z.B. bei Anwendungs-programmierung
  - BS-Schnittstellen
  - Shell, Shell-Skripte
- Den Teilnehmern mit Programmiererfahrung ggf. schon bekannt

Shell und Shell-Skripte

- **Shells** (oder **Kommandozeilen-interpreter**, CLI) setzen Texteingaben in Aufrufe von BS-APIs um und stellen die Ausgaben des BS dar
  - Sie nutzen selbst BS-APIs
- Meist mit einer eingebauten Programmiersprache
- Die Programme heissen **Shell-Skripte**



Shells - Wichtigste Beispiele

- command.com („DOS-Shell“)
  - MS-DOS, Windows 9x-Linie und NT-Linie
- Windows PowerShell
  - Windows XP, Server 2003, Vista, Windows 7 und 8
- Unix-Shells
  - Unix, Linux, MacOS
  - sh (Bourne-Shell), bash (Bourne-Again-Shell), ksh (Korn-Shell), csh (C Shell) und zsh (Z-Shell), …
- Wir fokussieren uns auf die Unix-Shells
  - Auch unter Windows einsetzbar, z.B. via Cygwin project

Kurze Geschichte der Shells /1

- Bourne-Shell – 1978
  - Für Unix Version 7; Erfinder Steve Bourne
  - Ursprung aller modernen Shells unter Unix
  - Mächtige Prg.-Sprache aber *mangelnde Interaktivität*
- C-Shell – 1979
  - Entwickelt in BSD-Laboren, von Bill Joy
  - Syntax angeleht an der Prg.-Sprache C, nicht kompatibel zu Bourne-Shell => etwas ungewöhlich
  - Populärer Clone: **tcsh**

Kurze Geschichte der Shells /2

- Korn-Shell – 1983
  - Weiterentwicklung der Bourne-Shell
  - Von David Korn für das System V bei AT&T entwickelt
  - *Interaktiv*, mächtige Programmierelemente
  - Versionen: Original, 1986, 1988, 1993
- Bourne-Again-Shell (bash) - 1989
  - De-facto Standard auf GNU/Linux-Systemen
  - Weiterentwicklung von Bourne und Korn Shells
  - Versionen 1 bis 4 (Cygwin: 4.1.10)
- Z-Shell - von Paul Falstad - 1990
  - „Eierlegende Wollmilchsau“
  - Emuliert Bourne-Shell, C-Shell und Korn-Shell

Die wichtigsten 15+ Linux CLI-Befehle (Link)

- Dateien und Verzeichnisse:
  - ls, cd, pwd, mv, mkdir, rm, rmdir, chmod
  - which, ln, tar, du
- Anzeigen und Editieren
  - echo, more, less, tail, cat, grep, sort, wc, nano, vim
- Netzwerk
  - nslookup, wget, ping
- Generell
  - history, man, sudo
- Prozesse
  - ps, top, nohup <Befehl>

Hello World in Bash

- Um herauszufinden, wo bash-Programm liegt …
  - which bash => /bin/bash
- Skript-Inhalt in Datei hello.sh schreiben
  - #! /usr/bin/bash
  - echo Hello World
    - Bem. „**#!**“ (Hash-Bang bzw. She-Bang) sagt dem Linux/Unix, welches Programm die Datei interpretieren soll
- Berechtigung zum Ausführen setzen
  - chmod +x hello.sh
- Ausführen
  - ./hello.sh

Umfrage (https://pingo.coactum.de/301541)

- Die Vorteile von Shell-Skripten gegenüber „normalen“ Anwendungsprogrammen (geschrieben in C/C++, Rust, Java, Python..) sind u.a.:
  - A. Die Shell-Skripte können auf gewisse Funktionen des BS zugreifen, die den Anwendungsprogrammen vorbehalten bleiben
  - B. Die Shell-Skripte ermöglichen eine schnellere und einfachere Entwicklung von (einfachen) Automatisierungsaufgaben
  - C. Die Shell-Skripte führen vergleichbare Aufgaben schneller aus, da sie meist kürzer sind
  - D. Die Shell-Skripte erlauben höhere Portabilität zwischen den BS, insbesondere innerhalb einer Familie (z.B. Unix)

Variablen, Quoting und Parameter

- Zuweisen von Werten von Variablen in Skripten
  - new_var=10 # Das „#“ leitet ein Kommentar ein
  - new_var2=“Ein String.“ # Kein Leerzeichen vor/nach „=„
  - myhost=$(hostname) # $(X) führt X aus; Ausgabe als Wert
  - myhost=`hostname` # alternative Schreibweise zu $( )
- Anzeigen und Quoting von Variableninhalten
  - echo “new_var hat den Wert $new_var“
  - echo “\$myhost = $myhost“ # => $myhost = pvs13
- Parameter
  - Parameter an ein Skript werden als $1, $2, … referenziert
  - Bsp. **./myskript a b c** => $1 ist “a“, $2 ist “b“ usw.

Eine Aufgabe

- Ein paar Befehle …
  - cd <Pfad> : wechselt das Arbeitsverzeichnis zu <Pfad>
  - pwd : gibt den aktuellen Pfad an
  - ls : zeigt den Inhalt des Arbeitsverzeichnises an
- Aufgabe: schreiben Sie ein Shell-Skript „myls“, das beim Aufruf „./myls x“
  - zum Verzeichnis x wechselt
  - den Inhalt von x anzeigt
  - und zum ursprunglichen Arbeitsverzeichnis wechselt
- Länge ca. 1+4 Zeilen

Eine Aufgabe - Lösung

- Aufgabe: schreiben Sie ein Shell-Skript „myls“, das beim Aufruf „./myls x“
  - zum Verzeichnis x wechselt
  - den Inhalt von x anzeigt
  - und zum ursprunglichen Arbeitsverzeichnis wechselt
- Eine mögliche Lösung:

```
#! /usr/bin/bash
lastdir=$(pwd)
cd $1
ls
cd $lastdir
```

Was macht dieses „Mystery“-Skript?

```
#!/bin/bash
echo -n "Enter a number: “
read num
i=2

while [ $i -lt $num ] do
  if [ `expr $num % $i` -eq 0 ] then
    echo “Sorry, $num is not!“
    exit
  fi
  i=`expr $i + 1`
done

echo “Yes, $num is one of those!"
```

Mehr Informationen zu Shell-Skripting

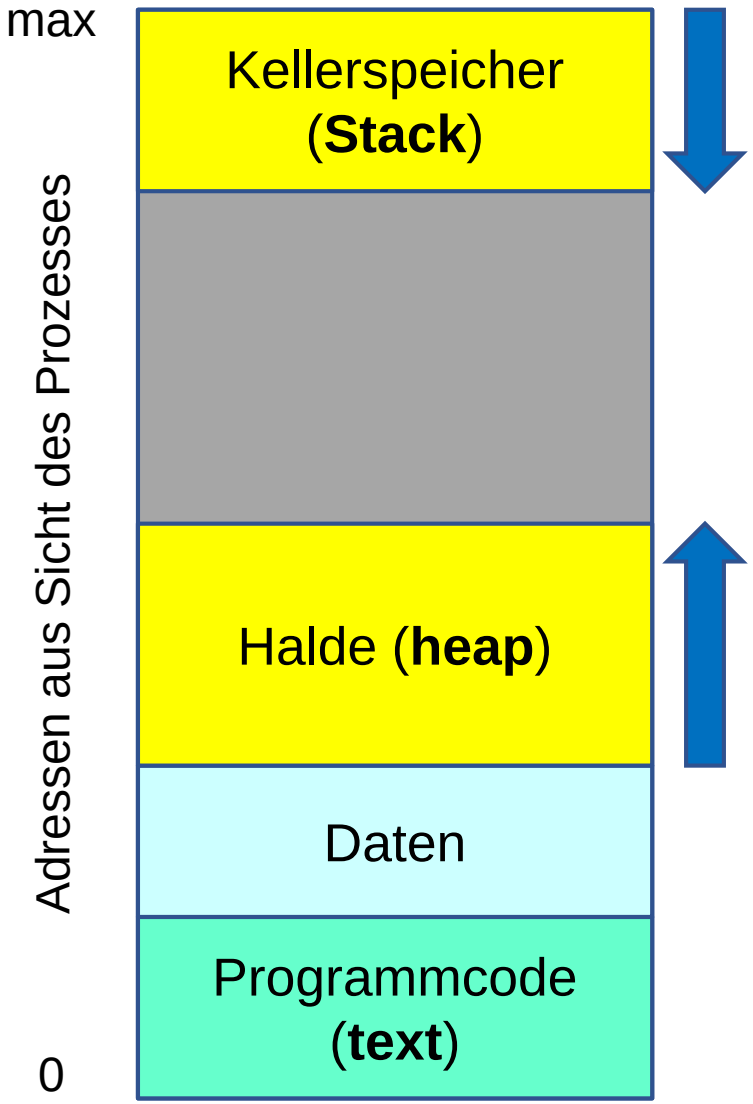
- Bash scripting Tutorial
  - https://linuxconfig.org/bash-scripting-tutorial
- Bash-Skripting-Guide für Anfänger
  - https://wiki.ubuntuusers.de/Shell/Bash-Skripting-Guide_f%C3%BCr_Anf%C3%A4nger/
- A quick guide to writing scripts using the bash shell
  - https://www.panix.com/~elflord/unix/bash-tute.html
- Shell programming with bash: by example, by counter-example
  - http://matt.might.net/articles/bash-by-example/
- Linux-Praxisbuch: Shellprogrammierung
  - https://de.wikibooks.org/wiki/Linux-Praxisbuch:_Shellprogrammierung

Prozesse - Grundlagen

Prozesse

- **Prozess** := ein Programm in Ausführung
- Aktives Gebilde, im Gegensatz zu einem **Programm**
  - Ein Programm wird zum Prozess, wenn das BS den **Programmcode** in den Speicher lädt und startet
  - Gleiches Programm mehrmals gestartet => mehrere Prozesse
- Besteht aus viel mehr als nur dem Programmcode
  - Im Hauptspeicher: mehrere Bereiche
  - In der „Buchhaltung“ des BS: Beschreibung des Prozesses und seiner aktiven Ressourcen (z.B. offenen Dateien)
  - In der CPU: Registerinhalte, u.a. Programmzähler

Prozess im Hauptspeicher



Temporäre Daten

- Rücksprungadressen, lokale Variablen, …

Dynamisch angeforderter Speicher, z.B. für Objekte

Globale Variablen, Konstanten

Ausführbarer Code

Process Control Block (PCB)

- Eine Beschreibung des Prozesses aus der Sicht des BS („Buchhaltung“ des BS)

Prozesskennung (**pid**)
Prozesszustand (kommt noch)
CPU-Programmzähler (Kopie)
Kopien von Inhalten anderer CPU-Register
Schedulinginformation (Scheduling: **Ablaufsteuerung**)
Speichergrenzen, Daten zu Speichermanagement

Liste der offenen Dateien
Status von I/O-Geräten
Abrechnungsinformationen, Nutzungsstatistiken
Threads (später)
…

PCB in Linux

- PCB wird in Linux als eine C struct (Verbund) namens **task_struct** gespeichert
  - Alle aktiven Prozesse sind als eine doppelt-verkette Liste von task_struct‘s dargestellt
  - Zeiger **current** zeigt zu dem aktuell ausgeführten Prozess

Feld in PCB	Beschreibung
pid_t pid	Prozesskennung (process id)
long state	Zustand
unsigned int time_slice	Scheduling-Information
struct task_struct *parent	Zeiger auf den PCB des Elternprozesses
struct list_head children	Anfang der Liste von Kinderprozessen
struct files_struct *files	Liste der geöffneten Dateien
struct mm_struct *mm	Infos zum Adressraum des Prozesses
…	…

Systemaufrufe zur Prozessverwaltung

POSIX Prozessverwaltung

Aufruf	Beschreibung
pid = **fork**()	Erzeugen eines (Kind)Prozesses
s = **execve** (name, argv, environp)	Speicherabbild eines Prozesses ersetzen
pid = **waitpid** (pid, &statloc, options)	Warten auf Beendigung eines Kindprozesses
_exit (status)	Prozess beenden und status zurückgeben (0 = alles OK, link)

- **fork()**: erzeugt eine identische Kopie des Prozesses (und startet diese) – ein Kindprozess
- **execve()**: das Speicherabbild (= Programmcode, Daten ..) des aufrufenden Prozesses wird durch den Inhalt einer Datei mit Pfad „name“ ersetzt

Video: Fork, Exec, Waitpid

- **Unix system calls (1/2)**
- Teil 1: von 22:35 min bis 24:20 min [02c]
- Teil 2: von 27:07 min bis 31:50 min
- Link: https://www.youtube.com/watch?v=xHu7qI1gDPA

```
if fork() == 0:
    … // new (child) process
else:
    … // original (parent) process
```

Details von fork()

- Der Aufruf von **pid = fork()** erzeugt ein Kindprozess (K), aber der Eltern-Prozess (E) läuft auch weiter!
- Was passiert nach der Rückkehr von fork() in K / in E?
- E: Die Prozessnummer (process ID) wird in Variable pid geschrieben, E läuft normal weiter
- K: pid wird auf 0 gesetzt (wichtig!), K läuft „weiter“, als ob es bis jetzt der Prozess E wäre
  - Also **keine** Ausführung von K ab dem Code-Start, d.h. main()!
- Wie erreichen wir in unserem Programm, dass sich K und E verschieden verhalten?

„Do it yourself“ - Mini-Shell

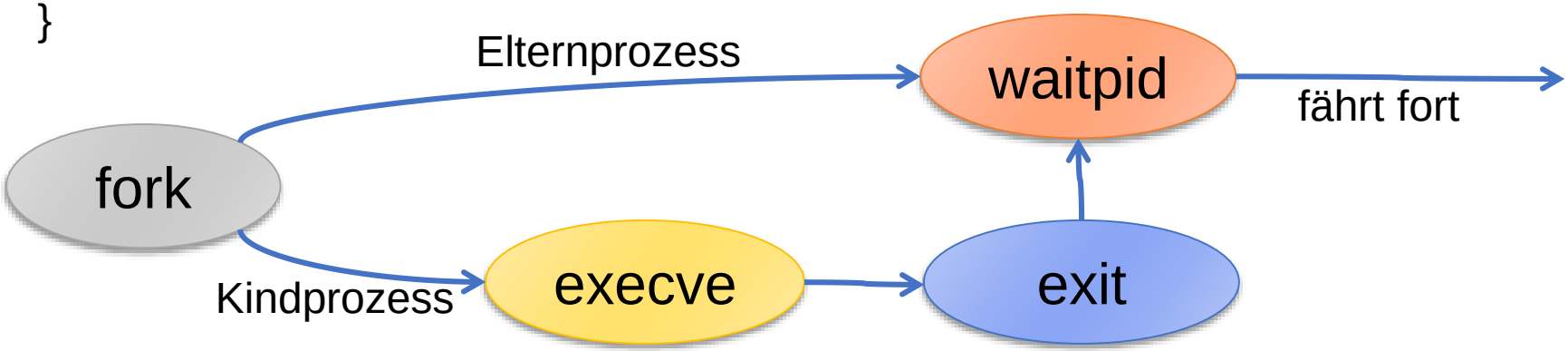
```
#define TRUE 1
while (TRUE) {
    type_prompt ();                                  /* Prompt ausgeben */
    read_command (command, parameters);              /* Befehl einlesen */

    if ( ?????? ) {              /* Kindprozess erzeugen */
        /* Code des Elternprozesses */
        ???????;                 /* Auf das Ende des Kindprozesses warten */
    } else {
        /* Code des Kindprozesses */
        ??????? ;                /* Befehl command ausführen */
    }
}
```

„ Do it yourself“ - Mini-Shell (Auflösung)

```
#define TRUE 1
while (TRUE) {
    type_prompt();                                  /* Prompt ausgeben */
    read_command (command, parameters);             /* Befehl einlesen */

    if ( fork() != 0 ) {                            /* Kindprozess erzeugen */
        /* Code des Elternprozesses */
        waitpid (-1, &status, 0);          /* Auf Ende des Kindprozesses warten */
    } else {
        /* Code des Kindprozesses */
        execve (command, parameters, 0);      /* Befehl command ausführen */
    }
}
```



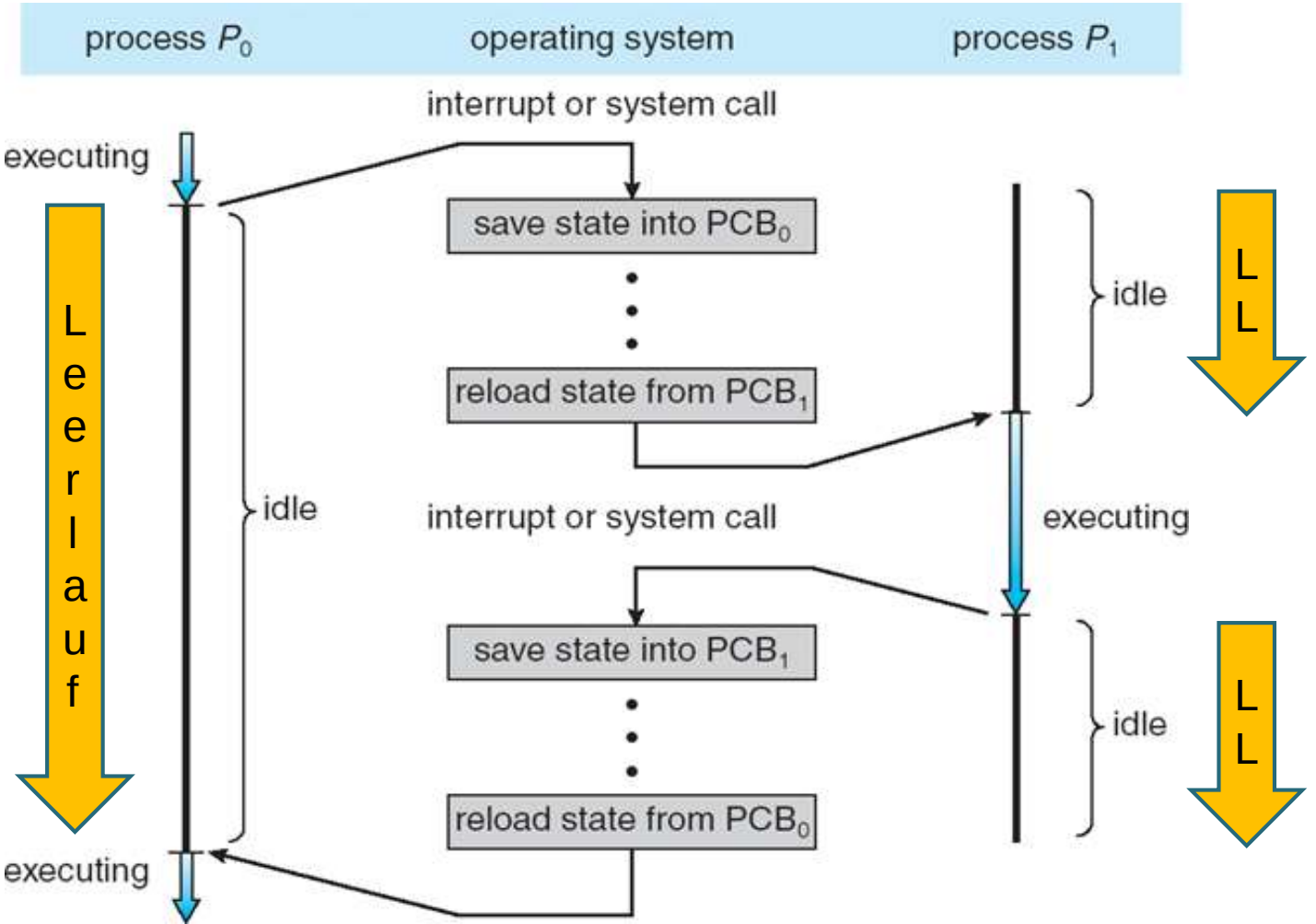
Prozesse - Verwaltung

Prozesse und Multiprogrammierung

- Erinnerung: die Multiprogrammierung (zuerst bei OS/360) war ein erheblicher Fortschritt
- Es bedeutete zugleich einen 10-100 fachen Zuwachs an BS-Komplexität – warum?
- Viele neue Funktionen waren notwendig:
- Verwaltung und Scheduling (**Ablaufsteuerung**) von Prozessen
- Kommunikation der Prozesse untereinander
- Schutz voneinander => komplexe Speicherverwaltung

Wechsel (Switch) zwischen Prozessen

- Prozesswechsel ist recht komplex
  - Aufwand ist abhängig von der CPU und dem BS



Zusammenfassung

- Shell-Programmierung
- Prozesse in einem BS
  - Process control block (PCB)
- Prozessverwaltung:
  - Systemaufrufe
- Struktur von BS (Zusatzfolien)
- Quellen
  - Tannenbaum, Kapitel 1 und 2
  - Silberschatz et al., Kapitel 3
  - Manche Abbildungen: William Stallings, Operating Systems: Internals and Design Principles, 6/e (Link)

Danke schön.

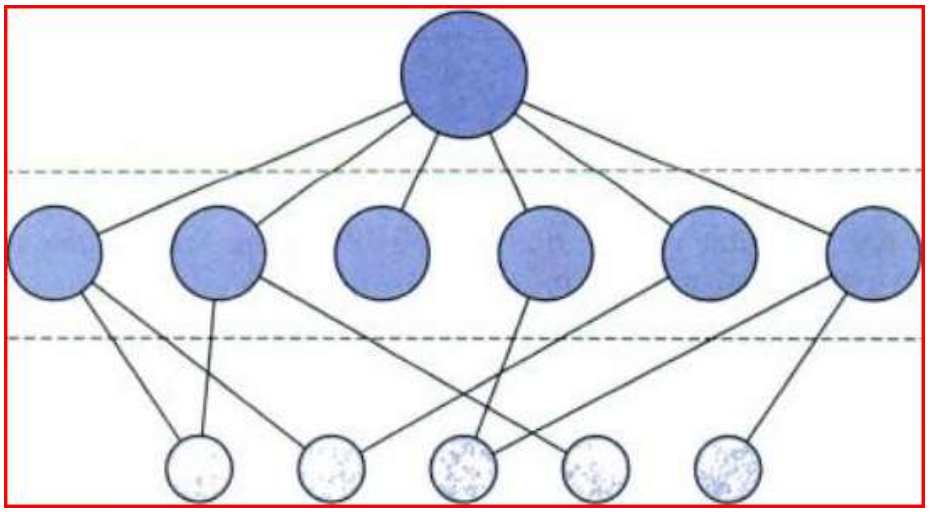
Architektur von Betriebssystemen
- Zusatzfolien -

Typische Architekturen von BS

- Historische Entwicklungen führten zu einigen Typen von BS-Architekturen
  - Monolithische Systeme
  - Geschichtete Systeme
  - Modulare Systeme
  - Mikrokerne
  - Client-Server-Systeme
  - Virtuelle Maschinen

Monolithische Systeme

- Grundstruktur:
  - Eine **Hauptfunktion**, die die angeforderten Dienstprozeduren aufruft
  - Eine Reihe von **Dienstprozeduren**, die die Systemaufrufe ausführen
  - Eine Menge von **Hilfsprozeduren**, die den Dienstprozeduren helfen



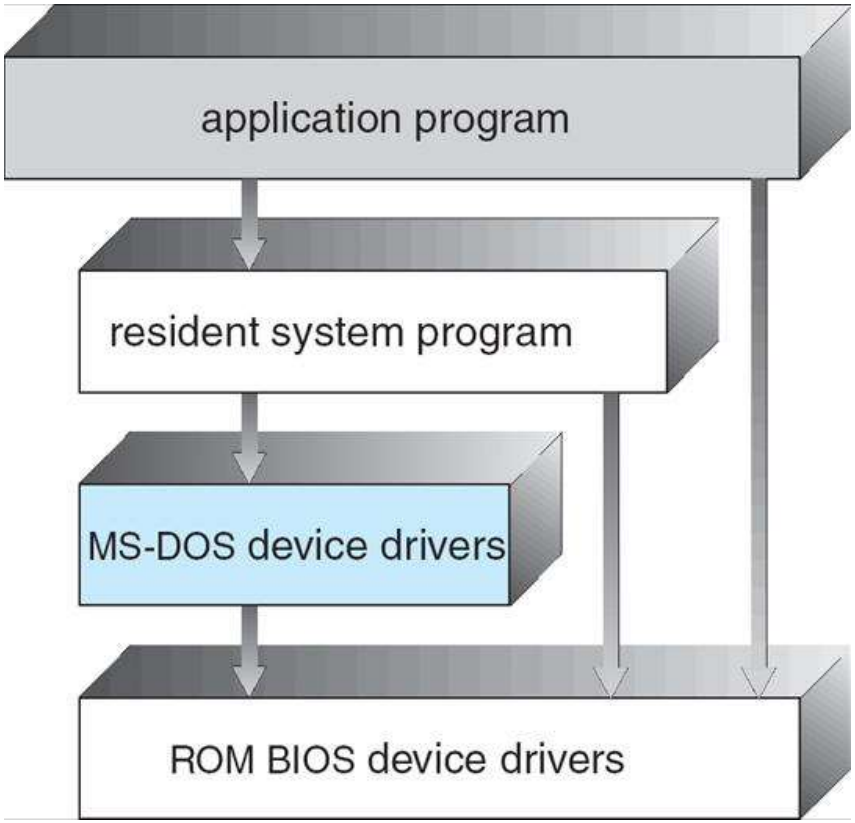
Hauptfunktion

Dienstprozeduren

Hilfsprozeduren

Monolithische Systeme – MS-DOS

- Beispiel: **MS-DOS** – geschrieben, um die größte Funktionalität im kleinsten Speicher zu unterbringen
  - Keine Unterteilung in Module
  - Obwohl es gewisse Struktur besitzt, sind die Schnittstellen (APIs) und Schichten der Funktionalität nicht gut separiert

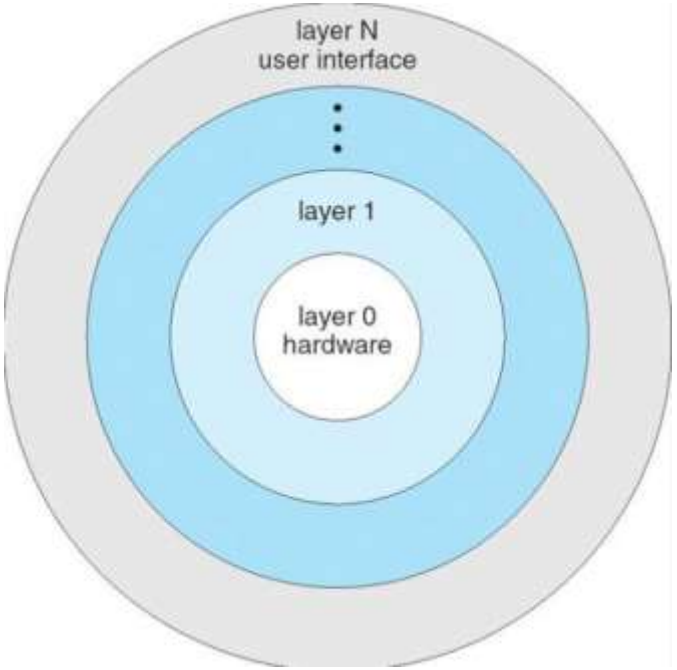


Geschichtete Systeme

- Hier ist ein BS in eine Anzahl von Schichten (Ebenen) unterteilt, von denen jede auf den darunterliegenden Schichten aufgebaut ist
  - Die untere Schicht (Schicht 0) ist die Hardware, die höchste (Schicht N) ist die Benutzeroberfläche
- Mit Hilfe der Modularität sind die Schichten so gewählt, dass jede Funktion und jeder Dienst nur die untergeordneten Schichten verwendet
- Beispiele
  - Das **THE-System**, gebaut von E. W. Dijkstra (1968) an der Tech. Hochschule Eindhoven (5 Schichten)
  - Das **MULTICS-System**

Geschichtete Systeme

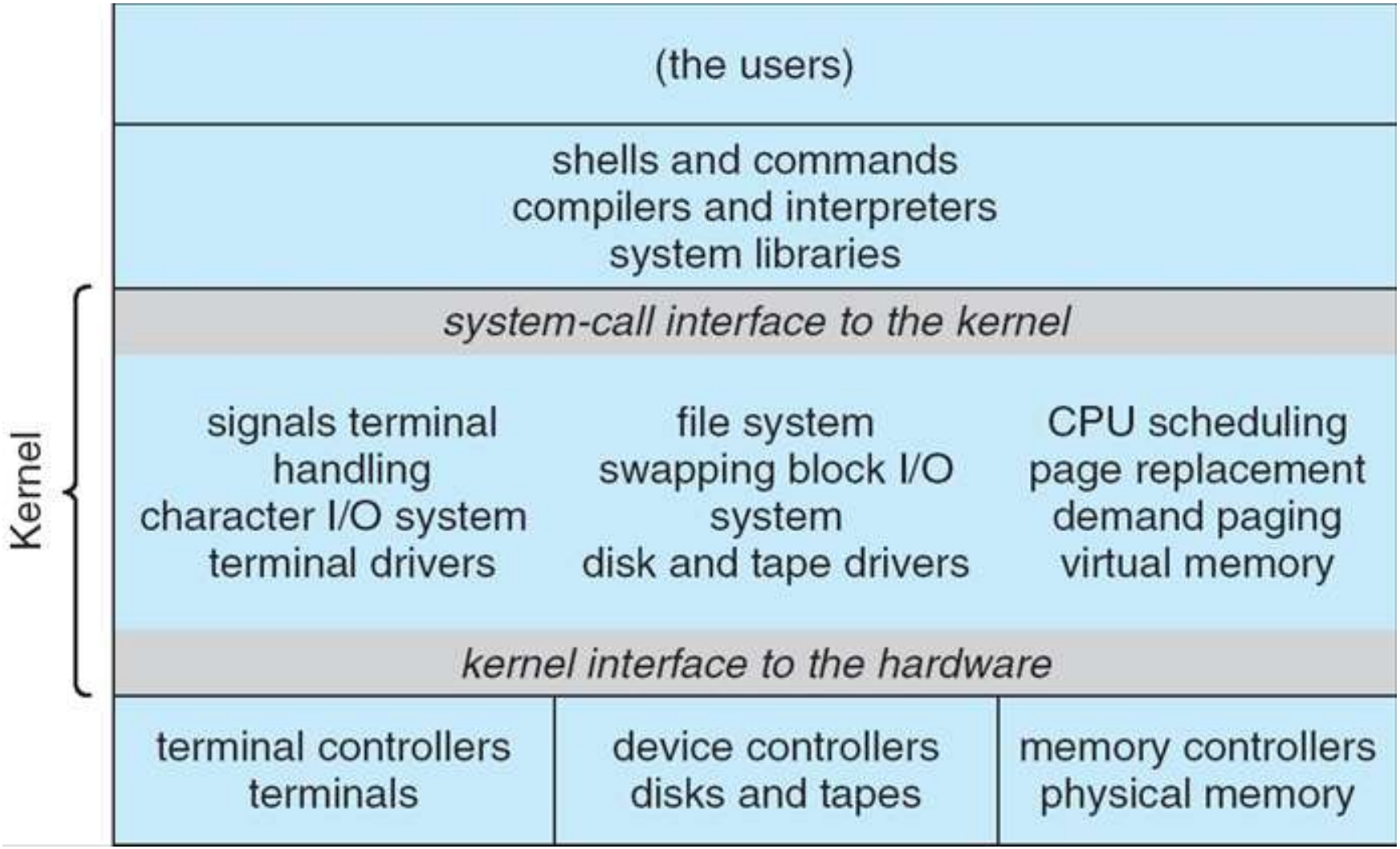
- Das **MULTICS-System** war als eine Folge konzentrischer Ringe organisiert
- Wenn Code vom Ring n+1 eine Prozedur aus Ring n aufrufen wollte, musste im Prinzip ein Systemaufruf stattfinden,
  - Dabei wurden die Parameter überprüft
- Die Prozesse vom Ring n waren (durch Hardware) vom Zugriff durch die Prozesse aus Ringen n+1, n+2, … geschützt



Geschichtete Systeme - UNIX

- Das ursprüngliche UNIX hatte nur beschränkte Strukturierung (begrenzt durch die Hardware)
  - Es bestand aus zwei trennbaren Teilen
- **Systemprogramme**
- **Kern**(**el**)
  - Dieser Bestand aus allem unter der Systemaufruf-Schnittstelle und über der physischen Hardware
  - Stellte das Dateisystem, CPU-Scheduling, Speicherverwaltung und andere Betriebssystem-Funktionen bereit
  - Eine (zu große) Vielzahl von Funktionen für diese Ebene

Geschichtete Systeme - UNIX



Modulare Systeme

- Die meisten modernen BS nutzen **Kern-Module (loadable kernel module, LKM)**
  - Jedes Kernmodul implementiert eine Teilfunktionalität und kann zur Laufzeit nachgeladen oder entfernt werden
  - Beispiele: Linux, FreeBSD, OS X (Link)
- Vorteile?
- Erweiterbarkeit ohne Overhead und Reboot
  - Ohne LKM, jede neue Funktionalität müsste in den Basis-Kernel eingebunden werden => sehr großer Kernel
    - Bei Änderungen nötig: 1. Kernel neu kompilieren, 2. Reboot
  - Mit LKM kann neue Funktionalität (via ein Modul) zur Laufzeit geladen werden, nur wenn es benötigt wird
- Beschleunigte Entwicklung: Jedes Modul kann separat kompiliert werden

Quiz - Wie fehlerfrei sind Betriebssysteme?

- Wie viele Fehler könnte ein monolithisches BS mit 5 Mio. Codezeilen haben?
  - Erfahrungsgröße: 10 Fehler pro 1000 Codezeilen in soliden industriellen Systemen
  - D.h. ein monolithisches BS mit 5 Mio. Codezeilen hat (wahrscheinlich) ca. 50.000 Fehler im Kern
  - Deshalb gab es den Reset-Knopf ☺

Mikrokerne

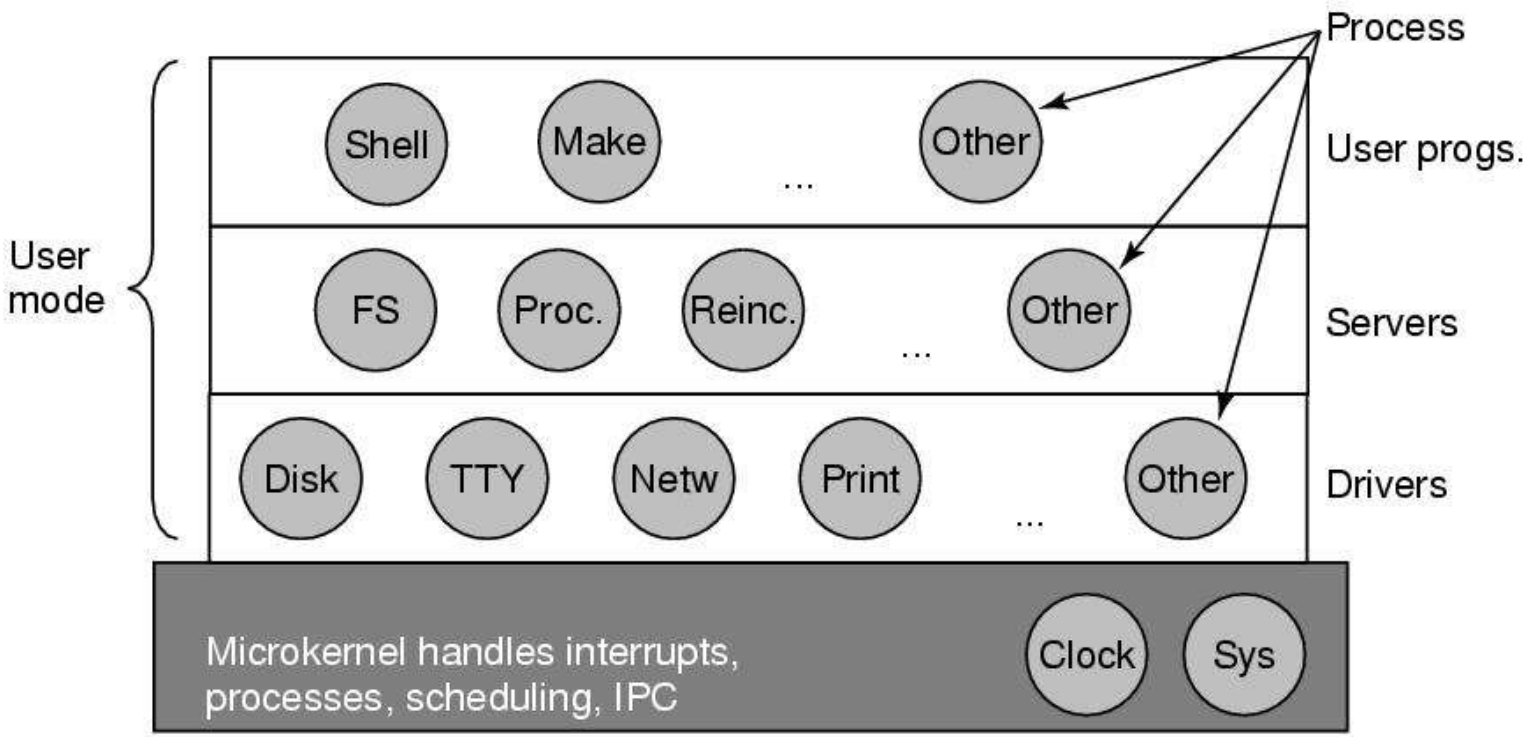
- Deshalb hat man versucht, so wenig Code wie möglich in dem Kernmodus (= im Kernel) auszuführen
  - Fehler im Kern können das System sofort zu Fall bringen
  - Idee: Das BS wird in sehr kleine, wohldefinierte Module aufgespalten
- Nur eines von denen – der **Mikrokern** – wird im Kernmodus ausgeführt
- Alle anderen Module (auch Gerätetreiber) werden im Benutzermodus ausgeführt
  - Ein Fehler hier wird das System nicht sofort lahmlegen

Mikrokerne /2

- Die Kommunikation erfolgt zwischen diesen Modulen durch **Nachrichtenaustausch** (**message passing**)
- Vorteile:
  - Einfacher, den Mikrokern zu erweitern
  - Einfacher, das BS auf neue Architekturen zu portieren
  - Verlässlicher, da weniger Code im Kernel-Modus ausgeführt wird
  - Mehr Sicherheit
- Nachteile:
  - Hoher Overhead bei der Kommunikation zwischen den Modulen
  - Geschichte der frühen Versionen von Windows NT

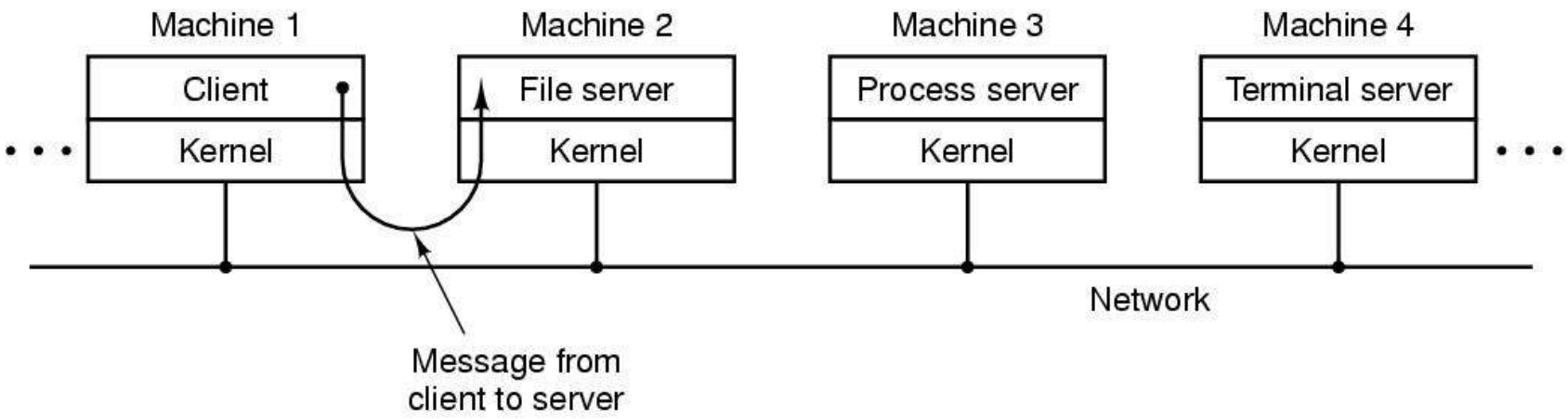
Mikrokerne – **MINIX 3**

- Mikrokern: 3200 Zeilen C und 800 Zeilen Assembler-Code
  - 35 Kernaufrufe
- **Reincarnation-Server**: Automatischer Neustart von Modulen
- POSIX-Konform, Open-Source: www.minix3.org



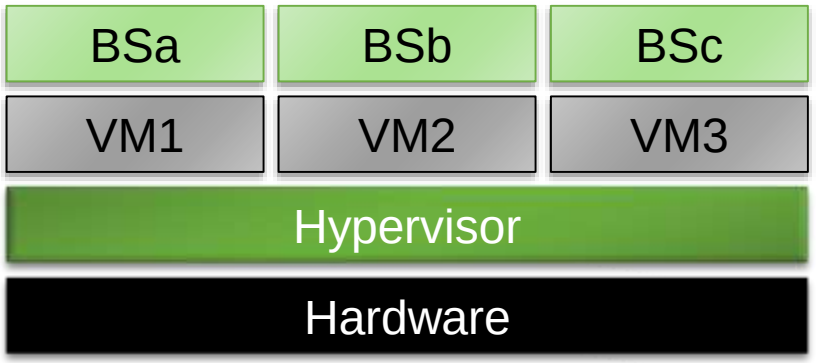
Das Client-Server-Modell

- Eine Variante der Mikrokern-Idee ist die Einteilung der Prozesse in zwei Klassen
  - **Server**, von denen jeder einige Dienste zu Verfügung stellt
    - Im Prinzip (eine Menge von) Mikrokernen
  - **Clients**, die solche Dienste nutzen
    - Weniger kritische Prozesse
- Kommunikation auch über Nachrichten
- Verallgemeinerung: Verteilung der Prozesse



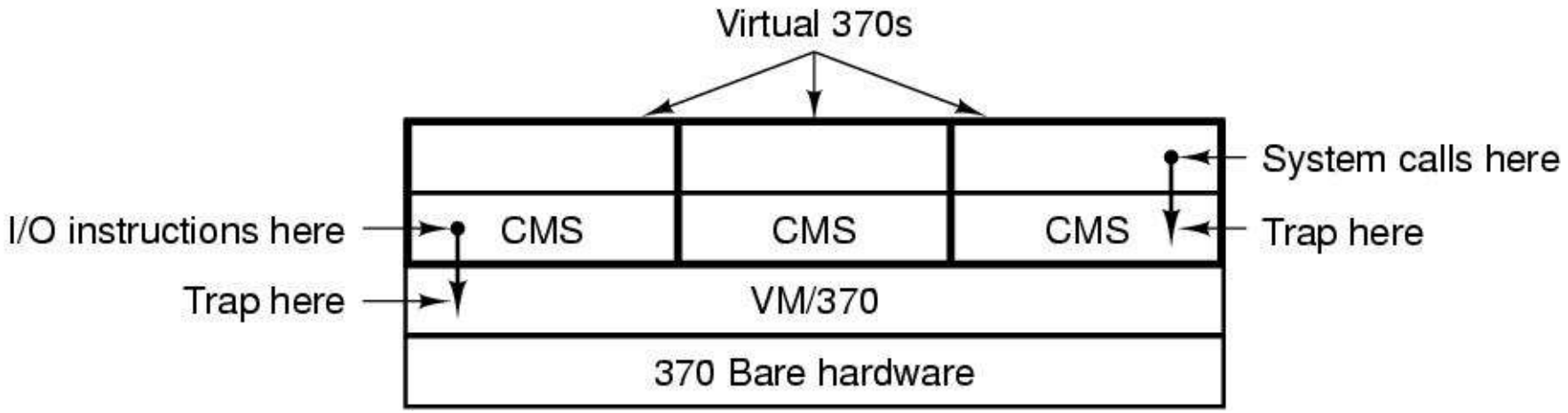
Virtuelle Maschinen /1

- Beobachtung: Das Konzept der erweiterten Maschine ist unabhängig von dem der **Multiprogrammierung** (MP)
- Idee: (1) man simuliert mehrere blanke Maschinen und (2) lässt in jeder von ihnen ein eigenes BS laufen
- Die Simulation erfolgt durch eine Softwareschicht (i.A. Hypervisor)
  - Die simulierte Hardware nennt man **virtuelle Maschine** (**VM**)
- (1): MP entsteht durch den Wechsel zwischen VMs
- (2): Jede VM wird durch eigenes BS in eine erweiterte Maschine umgewandelt

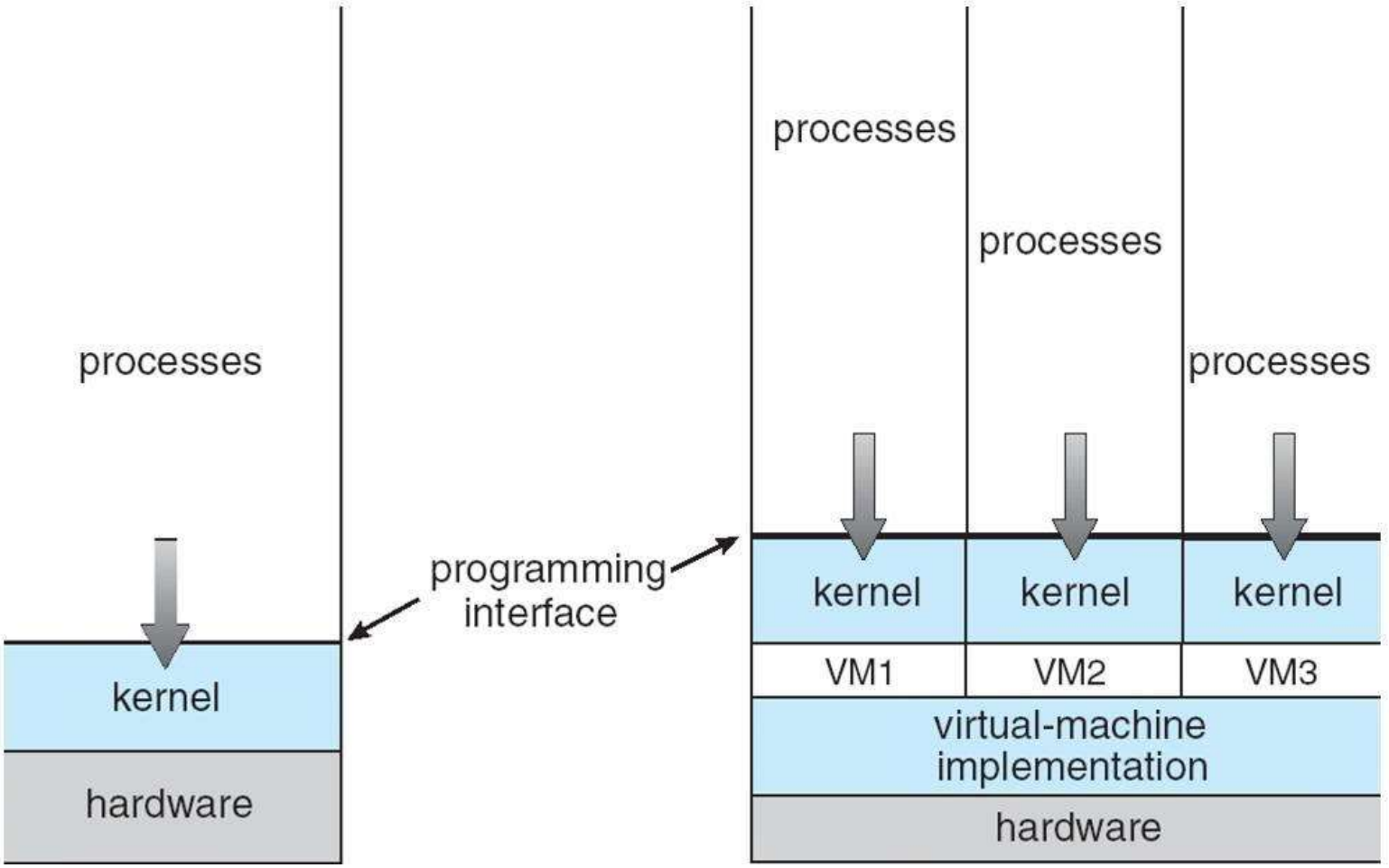


Virtuelle Maschinen /2

- Dieses Konzept wurde bei einem Nachfolger von OS/360 umgesetzt: **VM/370** (später **z/VM**)
- VM/370 simulierte mehrere VMs
  - Und sorgte (durch Wechsel zwischen ihnen) für MP
- Jede VM bekam ein eigenes BS => erweiterte Maschine
  - Damals oft OS/360 oder CMS (Conversational Monitor System)
- Die Idee der VM wurde vor ca. 10 Jahren im PC-Bereich wiederentdeckt: VMWare, XEN, VirtualBox, ..



Native Ausführung vs. Virtuelle Maschinen



Native Maschine

Ausführung mit VMs