

3.2 Schnittstellen (2)

Administratoren:

Werkzeuge zur Benutzerverwaltung, langfristige Systemsteuerung

Programme:

„*Supervisor Calls (SVC)*“,

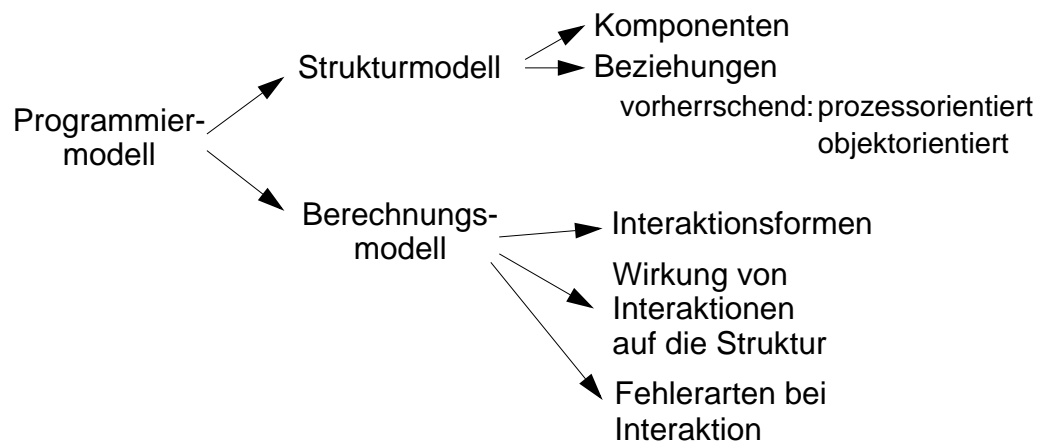
„*Application Programmer Interface (API)*“

Hardware:

Gerätetreiber

3.3 Programmiermodelle

- Betriebssystem realisiert ein Programmiermodell
 - ◆ Keine Notwendigkeit genauer Kenntnisse über Hardwareeigenschaften und spezielle Systemsoftwarekomponenten
 - ◆ Schaffung einer begrifflichen Basis zur Strukturierung von Programmsystemen und ihrer Ablaufsteuerung



3.3 Programmiermodelle (2)

- Beispiele für Strukturkomponenten
 - ◆ Dateien (Behälter zur langfristigen Speicherung von Daten)
 - ◆ Prozesse (in Ausführung befindliche Programme)
 - ◆ Klassen (Vorlagen zur Bildung von Instanzen)
 - ◆ Instanzen/Objekte
 - ◆ Prozeduren
 - ◆ Sockets (Kommunikationsendpunkte, „Kommunikationssteckdosen“)
 - ◆ Pipes (Nachrichtenkanäle)
- Beispiele für Beziehungen
 - ◆ A kann B referenzieren, beauftragen, aufrufen, modifizieren
 - ◆ Pipe P verbindet A und B

3.3 Programmiermodelle (3)

- Beispiele für Interaktionsformen
 - ◆ Prozedur-(Methoden-)Aufruf
 - ◆ Nachrichtenaustausch
 - ◆ Gemeinsame Speichernutzung
- Wirkung von Interaktionen auf die Struktur
 - ◆ Erzeugung und Tilgung von Prozessen
 - ◆ Instanziierung von Objekten
- Fehlerarten bei Interaktion
 - ◆ Verlust, Wiederholung oder Verspätung von Nachrichten
 - ◆ Abbruch aufgerufener Methoden, Ausnahmebehandlung

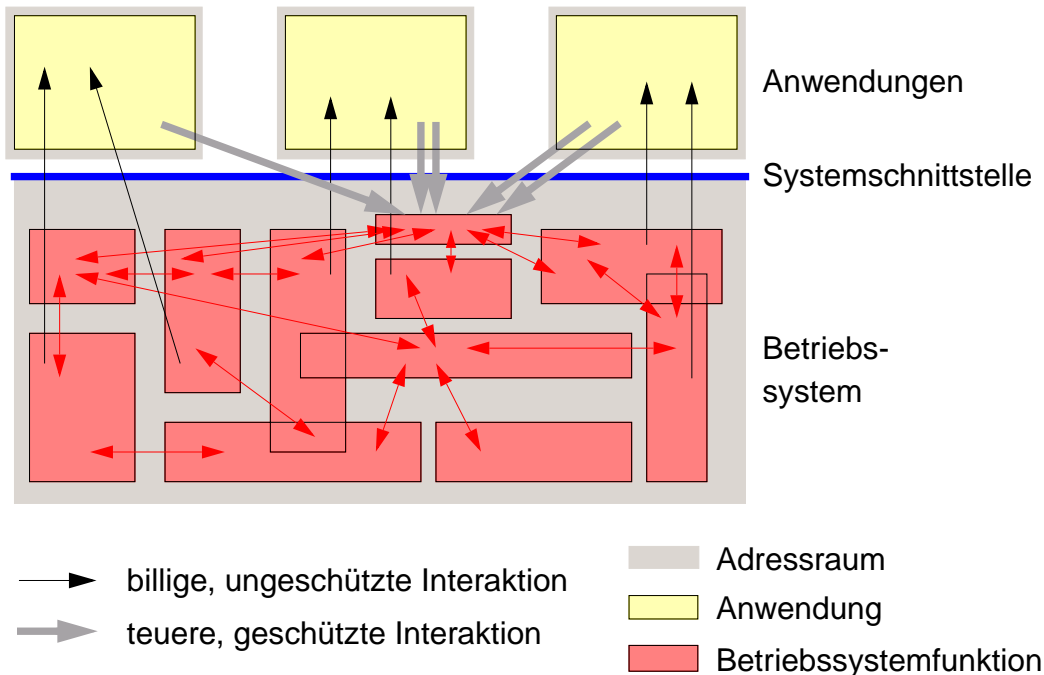
3.4 Ablaufmodelle

- Betriebssystem realisiert eine Ablaufumgebung
- Bereitstellung von Hilfsmitteln zur Bearbeitung von Benutzerprogrammen und zur Steuerung ihrer Abläufe.
 - ◆ Laden und Starten von Programmen
 - ◆ Überwachung des Programmablaufs
 - ◆ Beenden und Eliminieren von Programmen
 - ◆ Abrechnung (*Accounting*)

4 Betriebssystemarchitekturen

- Umfang zehntausende bis mehrere Millionen Befehlszeilen
 - ◆ Strukturierung hilfreich
- Verschiedene Strukturkonzepte
 - ◆ monolithische Systeme
 - ◆ geschichtete Systeme
 - ◆ Minimalkerne
 - ◆ offene objektorientierte Systeme

4.1 Monolithische Systeme



Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [B-Intro.fm, 2001-10-22 16.10]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B - 22

4.1 Monolithische Systeme (2)

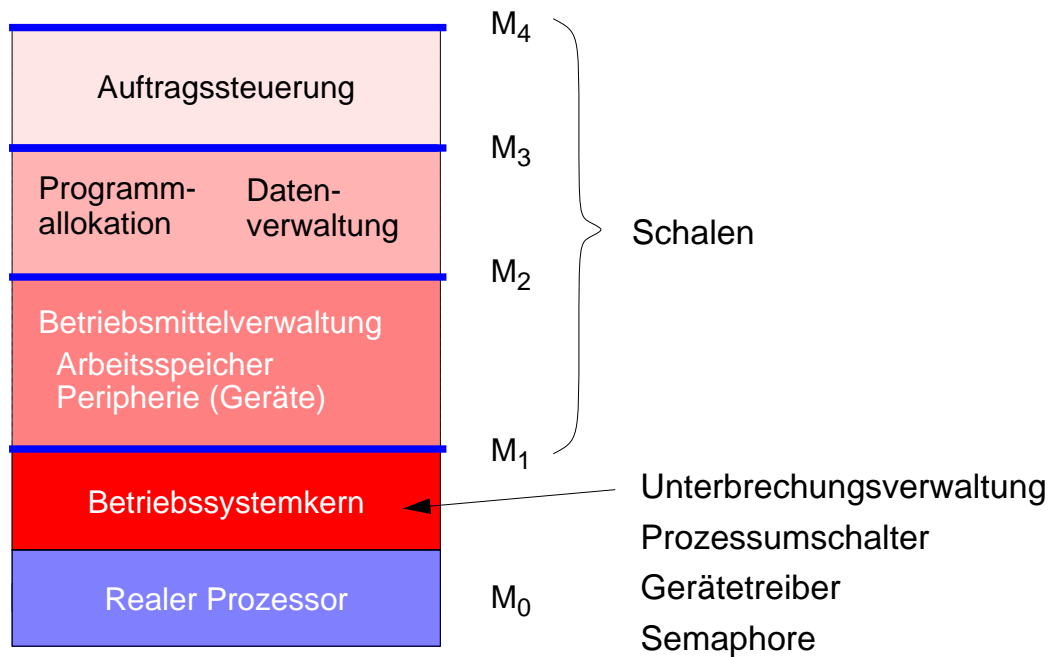
- ★ Vorteile
 - ◆ Effiziente Kommunikation und effizienter Datenzugriff innerhalb des Kerns
 - ◆ Privilegierte Befehle jederzeit ausführbar
- ▲ Nachteile
 - ◆ Keine interne Strukturierung (änderungsunfreundlich, fehleranfällig)
 - ◆ Kein Schutz zwischen Kernkomponenten (Problem: zugekaufte Treiber)

Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [B-Intro.fm, 2001-10-22 16.10]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B - 23

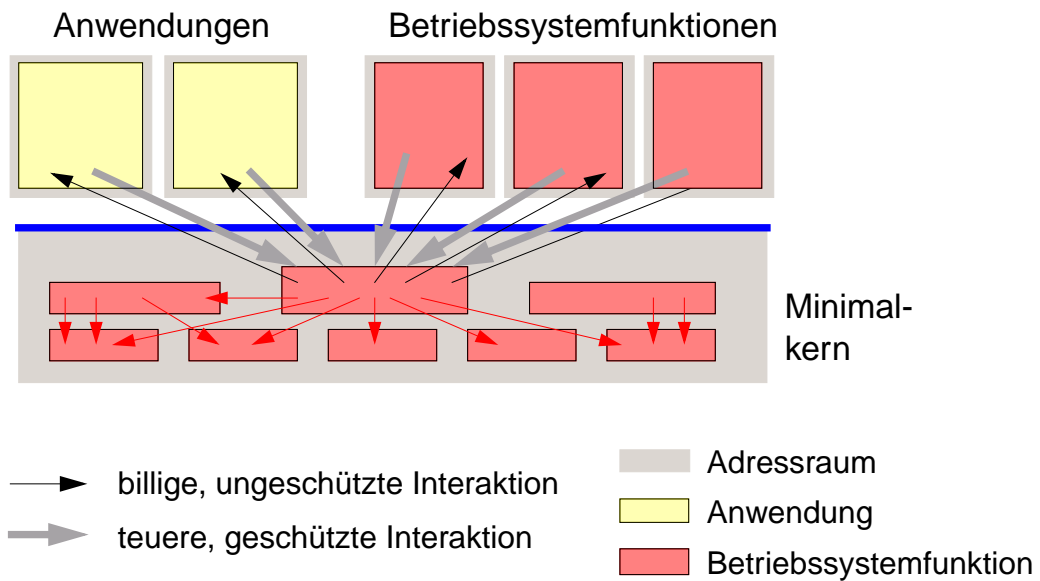
4.2 Geschichtete Systeme



4.2 Geschichtete Systeme (2)

- ★ Vorteile
 - ◆ Schutz zwischen verschiedenen BS-Teilen
 - ◆ Interne Strukturierung
- ▲ Nachteile
 - ◆ Mehrfacher Schutzraumwechsel ist teuer
 - ◆ Unflexibler und nur einseitiger Schutz (von unten nach oben)

4.3 Minimalkerne



Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [B-Intro.fm, 2001-10-22 16.10]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B - 26

4.3 Minimalkerne (2)

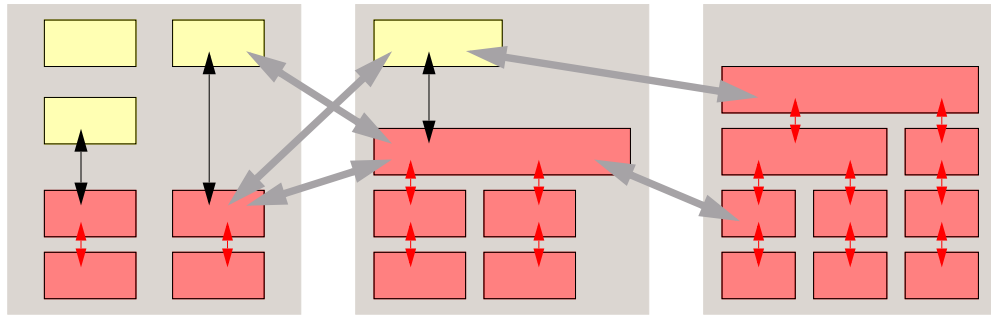
- ★ Vorteile
 - ◆ Gute Modularisierung
 - ◆ Schutz der Komponenten voreinander
- ▲ Nachteil
 - ◆ Kommunikation zwischen Modulen ist teuer

Systemprogrammierung I

© 1997-2001, Franz J. Hauck, Inf 4, Univ. Erlangen-Nürnberg [B-Intro.fm, 2001-10-22 16.10]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

B - 27

4.4 Objektbasierte, offene Systeme



—▶ billige, durch Objektkapselung geschützte Interaktion

■ Adressraum

■ Anwendungsobjekte

—▶ teure, durch Adressraumgrenze geschützte Interaktion

■ Betriebssystemobjekte

■ Sicherung der Modulgrenzen durch Programmiermodell und Software

◆ z.B. Objektorientierung und Byte-Code-Verifier in Java

4.4 Objektbasierte, offene Systeme (2)

★ Vorteile

- ◆ Schutz auf mehreren Ebenen (Sprache, Code-Prüfung, Adressraum)
- ◆ Modularisierung und Effizienz möglich

▲ Nachteile

- ◆ Komplexes Sicherheitsmodell

5 Geschichtliche Entwicklung

5.1 1950–1960

1950

- ◆ Einströmige Stapelsysteme
(*Single-stream batch processing systems*)
Aufträge zusammen mit allen Daten werden übergeben und sequentiell bearbeitet
- ◆ Steuerung durch Auftragsabwickler
(*Resident monitor, Job monitor*)
Hilfsmittel: Assembler, Compiler, Binder und Lader, Programmbibliotheken

1960

5.2 1960–1965

1960

- ◆ Autonome periphere Geräte → Überlappung von Programm-bearbeitung und Datentransport zw. Arbeitsspeicher und peripheren Geräten möglich
 - Wechselpufferbetrieb (abwechselndes Nutzen zweier Puffer)
 - Mehrprogrammbetrieb (*Multiprogramming*)
 - Spooling (*Simultaneous peripheral operation on-line*)
- ◆ Mehrere Programme müssen gleichzeitig im Speicher sein → Auslagern von Programmen auf Sekundärspeicher
- ◆ Programme müssen während des Ablaufs verlagerbar sein (*Relocation problem*)
- ◆ Echtzeitdatenverarbeitung (*Real-time processing*), d.h. enge Bindung von Ein- und Ausgaben an die physikalische Zeit

1965

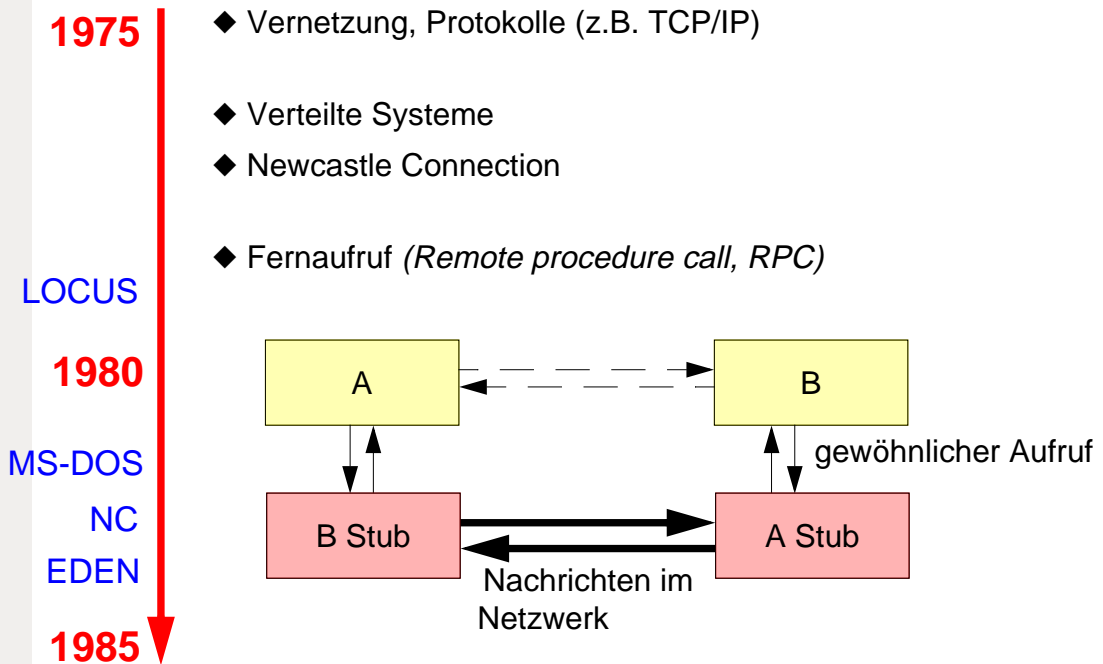
5.3 1965–1970

- 1965**
- ◆ Umsetzung von Programmadressen in Speicherorte zur Laufzeit: Segmentierung, Seitenadressierung (*Paging*)
- OS/360**
- ◆ Virtueller Adressraum: Seitentausch (*Paging*)
Seiten werden je nach Zugriff ein- und ausgelagert
 - ◆ Interaktiver Betrieb (*Interactive processing, Dialog mode*)
- THE**
- ◆ Mehrbenutzerbetrieb, Teilnehmersysteme (*Time sharing*)
- MULTICS**
- ◆ Problem: Kapselung von Prozessen und Dateien → geschützter Adressraum, Zugriffsschutz auf Dateien
 - ◆ Dijkstra: Programmsysteme als Menge kooperierender Prozesse (heute *Client-Server*)
- UNIX**
- ◆ Problem: Prozessinteraktion bei gekapselten Prozessen → Nachrichtensysteme zur Kommunikation, gemeinsamer Speicher zur Kooperation
- 1970**

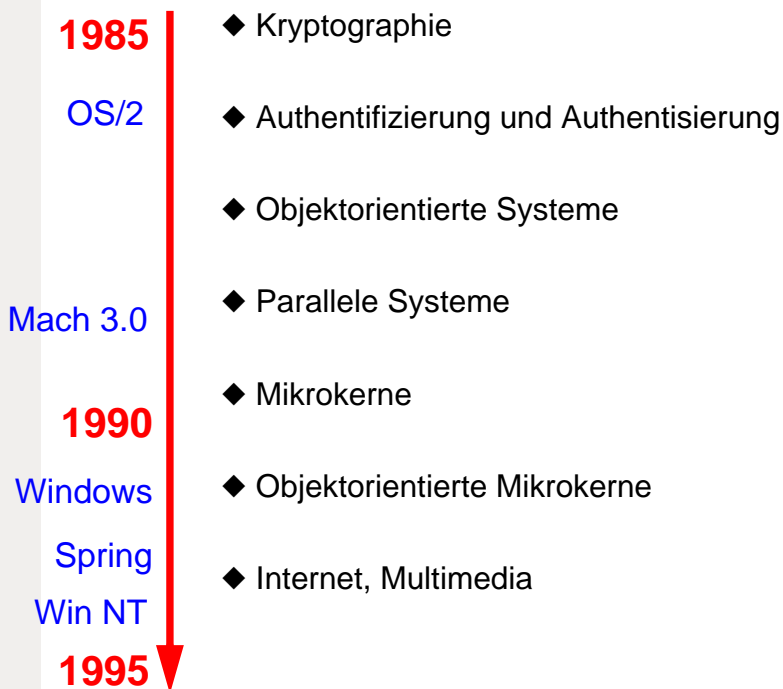
5.4 1970–1975

- 1970**
- ◆ Modularisierung:
Datenkapselung, Manipulation durch Funktionen (nach Parnas)
- VM**
- ◆ Virtuelle Maschinen: Koexistenz verschiedener Betriebssysteme im gleichen Rechner
- Hydra**
- MVS**
-
- ◆ Symmetrische Multiprozessoren: HYDRA
- Zugangskontrolle zu Instanzen durch Capabilities
 - Trennung von Strategie und Mechanismus
- 1975**

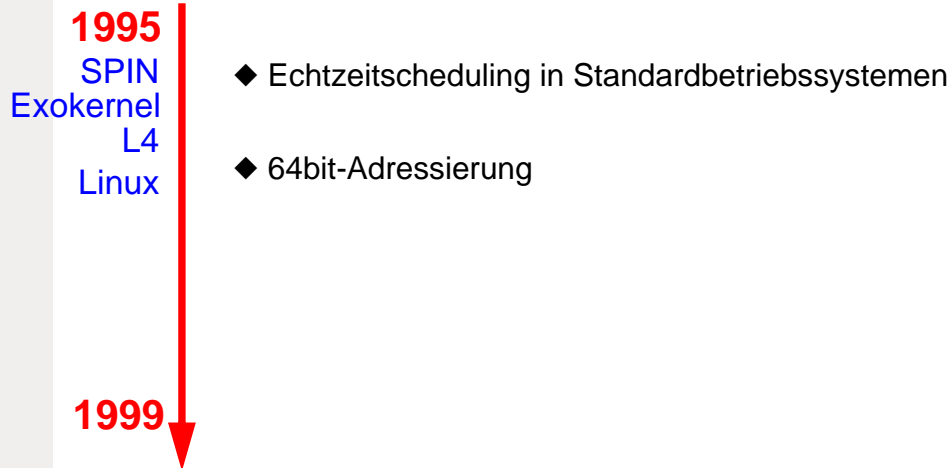
5.5 1975–1985



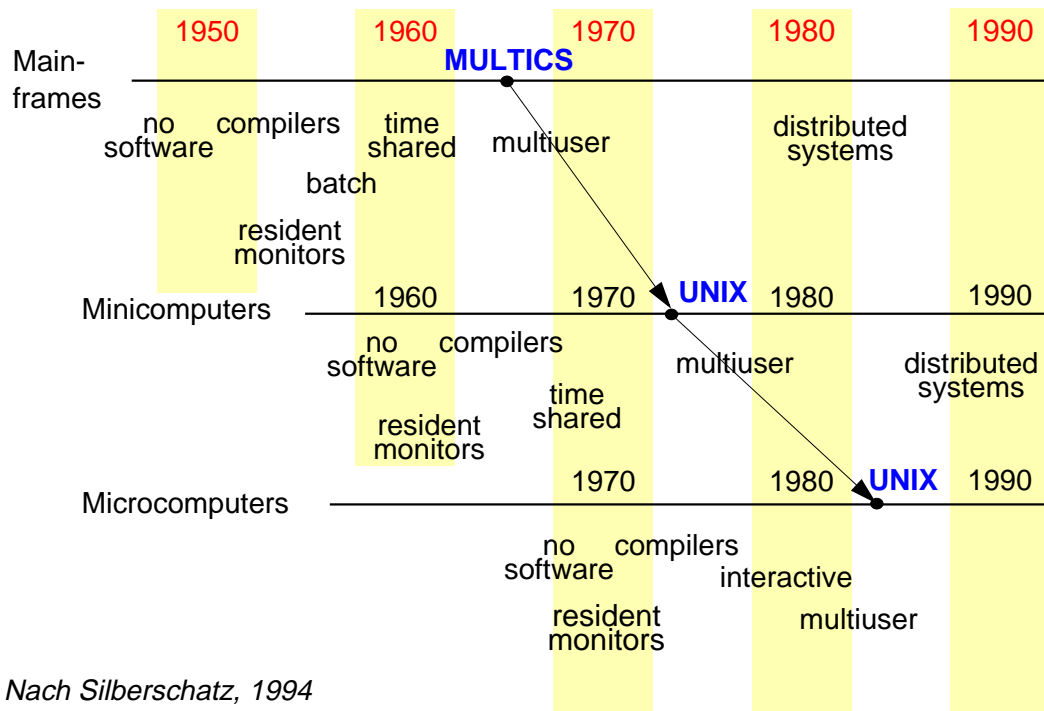
5.6 1985–1999



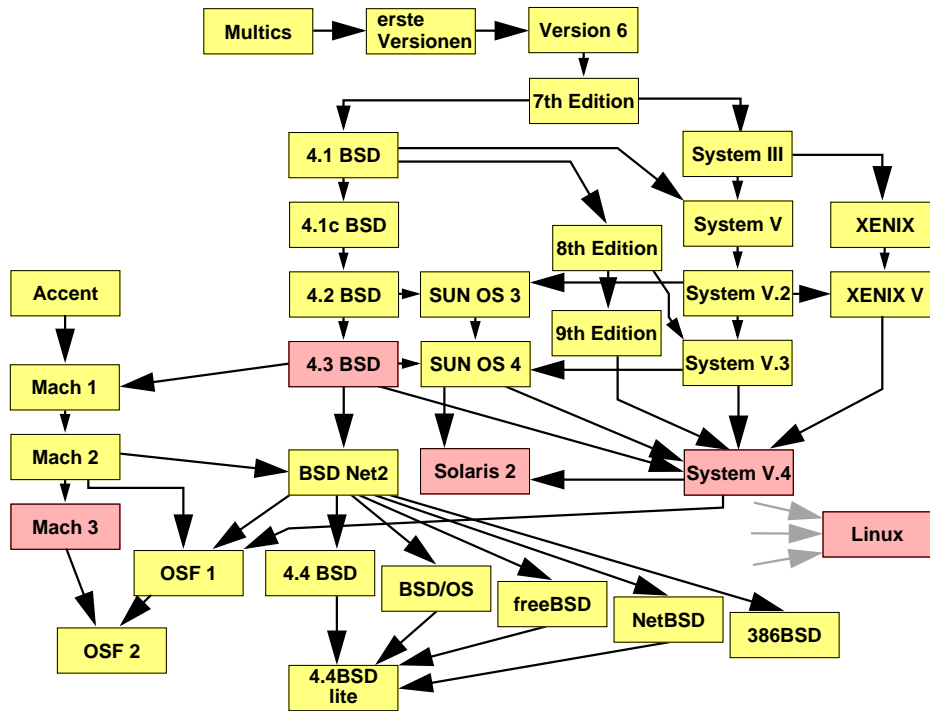
5.7 1995–1999



5.8 Migration von Konzepten



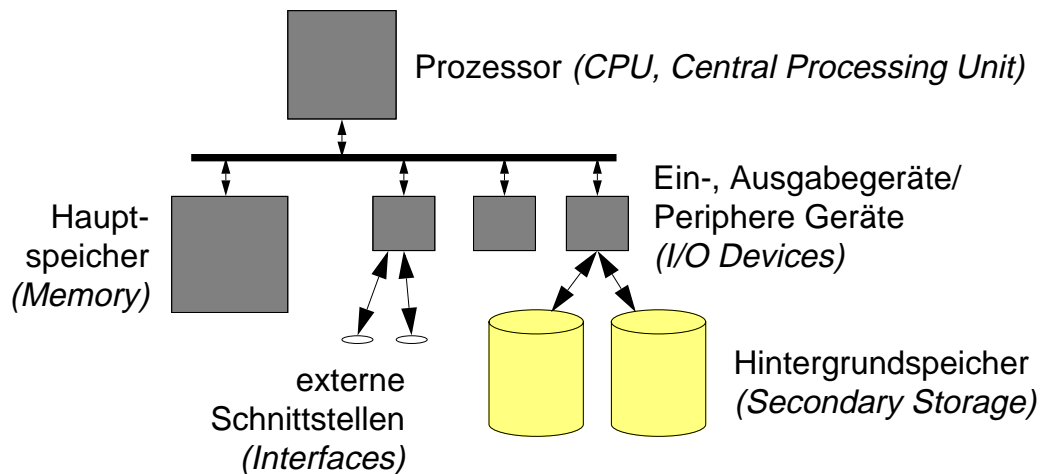
5.9 UNIX Entwicklung



C Dateisysteme

C Dateisysteme

■ Einordnung



C Dateisysteme (2)

■ Dateisysteme speichern Daten und Programme persistent in Dateien

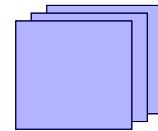
- ◆ Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Floppy Disk, Bandlaufwerke)
 - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
 - einheitliche Sicht auf den Sekundärspeicher

■ Dateisysteme bestehen aus

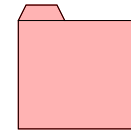
- ◆ Dateien (*Files*)
- ◆ Verzeichnissen, Katalogen (*Directories*)
- ◆ Partitionen (*Partitions*)

C Dateisysteme (3)

- Datei
 - ◆ speichert Daten oder Programme
- Verzeichnis
 - ◆ fasst Dateien (u. Verzeichnisse) zusammen
 - ◆ erlaubt Benennung der Dateien
 - ◆ enthält Zusatzinformationen zu Dateien
- Partitionen
 - ◆ eine Menge von Verzeichnissen und deren Dateien
 - ◆ Sie dienen zum physischen oder logischen Trennen von Dateimengen.
 - *physisch*: Festplatte, Diskette
 - *logisch*: Teilbereich auf Platte oder CD



Dateien



Verzeichnis



Partition

1 Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.

1.1 Dateiattribute

- *Name* — Symbolischer Name, vom Benutzer les- und interpretierbar
 - ◆ z.B. `AUTOEXEC.BAT`
- *Typ* — Für Dateisysteme, die verschiedene Dateitypen unterscheiden
 - ◆ z.B. sequenzielle Datei, zeichenorientierte Datei, satzorientierte Datei
- *Ortsinformation* — Wo werden die Daten physisch gespeichert?
 - ◆ Gerätenummer, Nummern der Plattenblocks

1.1 Dateiattribute (2)

- **Größe** — Länge der Datei in Größeneinheiten (z.B. Bytes, Blöcke, Sätze)
 - ◆ steht in engem Zusammenhang mit der Ortsinformation
 - ◆ wird zum Prüfen der Dateigrenzen z.B. beim Lesen benötigt
- **Zeitstempel** — z.B. Zeit und Datum der Erstellung, letzten Änderung
 - ◆ unterstützt Backup, Entwicklungswerkzeuge, Benutzerüberwachung etc.
- **Rechte** — Zugriffsrechte, z.B. Lese-, Schreibberechtigung
 - ◆ z.B. nur für den Eigentümer schreibbar, für alle anderen nur lesbar
- **Eigentümer** — Identifikation des Eigentümers
 - ◆ eventuell eng mit den Rechten verknüpft
 - ◆ Zuordnung beim Accounting (Abrechnung des Plattenplatzes)

1.2 Operationen auf Dateien

- **Erzeugen (*Create*)**
 - ◆ Nötiger Speicherplatz wird angefordert.
 - ◆ Verzeichniseintrag wird erstellt.
 - ◆ Initiale Attribute werden gespeichert.
- **Schreiben (*Write*)**
 - ◆ Identifikation der Datei
 - ◆ Daten werden auf Platte transferiert.
 - ◆ eventuelle Anpassung der Attribute, z.B. Länge
- **Lesen (*Read*)**
 - ◆ Identifikation der Datei
 - ◆ Daten werden von Platte gelesen.

1.2 Operationen auf Dateien (2)

- Positionieren des Schreib-/Lesezeigers (*Seek*)
 - ◆ Identifikation der Datei
 - ◆ In vielen Systemen wird dieser Zeiger implizit bei Schreib- und Leseoperationen positioniert.
 - ◆ Ermöglicht explizites Positionieren
- Verkürzen (*Truncate*)
 - ◆ Identifikation der Datei
 - ◆ Ab einer bestimmten Position wird der Inhalt entfernt (evtl. kann nur der Gesamthalt gelöscht werden).
 - ◆ Anpassung der betroffenen Attribute
- Löschen (*Delete*)
 - ◆ Identifikation der Datei
 - ◆ Entfernen der Datei aus dem Katalog und Freigabe der Plattenblöcke

2 Verzeichnisse / Kataloge

- Ein Verzeichnis gruppiert Dateien und evtl. andere Verzeichnisse
- Gruppierungsalternativen:
 - ◆ Verknüpfung mit der Benennung
 - Katalog enthält Namen und Verweise auf Dateien und andere Verzeichnisse, z.B. *UNIX*, *MS-DOS*
 - ◆ Gruppierung über Bedingung
 - Verzeichnis enthält Namen und Verweise auf Dateien, die einer bestimmten Bedingung gehorchen
z.B. gleiche Gruppennummer in *CP/M*
z.B. eigenschaftsorientierte und dynamische Gruppierung in *BeOS-BFS*
- Verzeichnis ermöglicht das Auffinden von Dateien
 - ◆ Vermittlung zwischen externer und interner Bezeichnung (Dateiname — Plattenblöcken)

2.1 Operationen auf Verzeichnissen

- Auslesen der Einträge (*Read, Read Directory*)
 - ◆ Daten des Verzeichnisinhalts werden gelesen und meist eintragsweise zurückgegeben
- Erzeugen und Löschen der Einträge erfolgt implizit mit der zugehörigen Dateioperation
- Erzeugen und Löschen von Verzeichnissen (*Create and Delete Directory*)

2.2 Attribute von Verzeichnissen

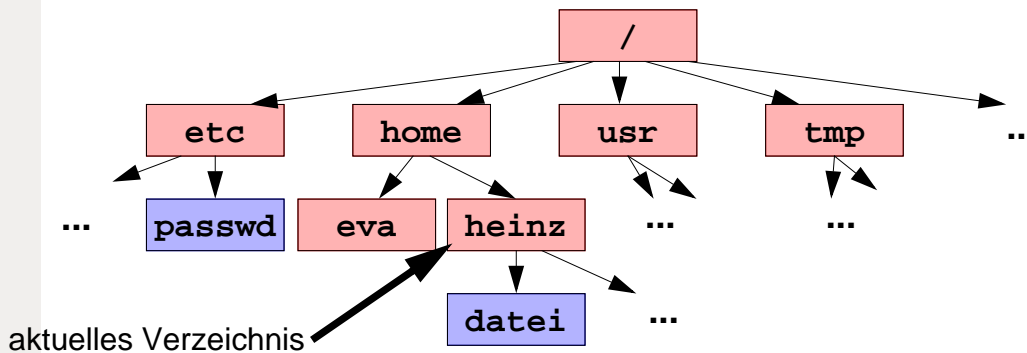
- Die meisten Dateiattribute treffen auch für Kataloge zu
 - ◆ Name, Ortsinformationen, Größe, Zeitstempel, Rechte, Eigentümer

3 Beispiel: UNIX (Sun-UFS)

- Datei
 - ◆ einfache, unstrukturierte Folge von Bytes
 - ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
 - ◆ dynamisch erweiterbar
 - ◆ Zugriffsrechte: lesbar, schreibbar, ausführbar
- Verzeichnis
 - ◆ baumförmig strukturiert
 - Knoten des Baums sind Verzeichnisse
 - Blätter des Baums sind Verweise auf Dateien (*Links*)
 - ◆ jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Verzeichnis (*Current Working Directory*) zugeordnet
 - ◆ Zugriffsrechte: lesbar, schreibbar, durchsuchbar, „nur“ erweiterbar

3.1 Pfadnamen

■ Baumstruktur

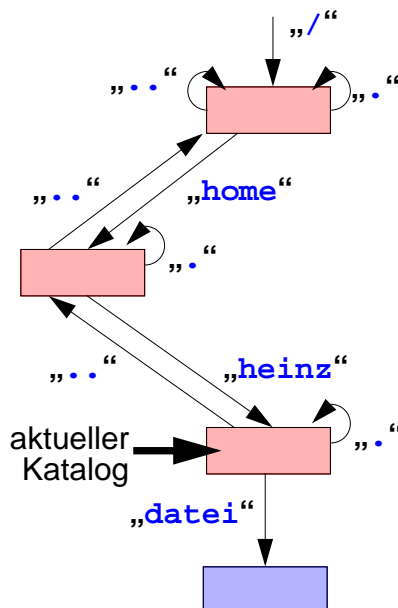


■ Pfade

- ◆ z.B. „/home/heinz/datei“, „/tmp“, „datei“
- ◆ „/“ ist Trennsymbol (*Slash*); beginnender „/“ bezeichnet Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellem Verzeichnis

3.1 Pfadnamen (2)

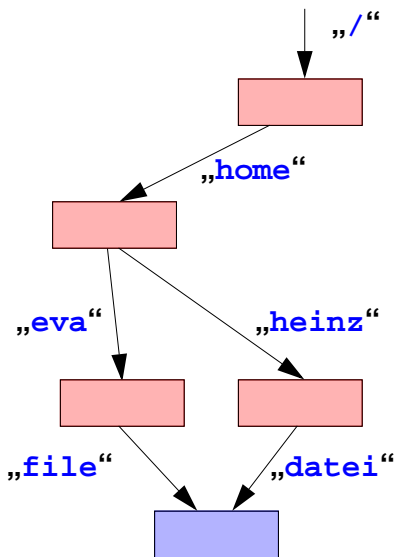
■ Eigentliche Baumstruktur



3.1 Pfadnamen (3)

■ Links (*Hard Links*)

- ◆ Dateien können mehrere auf sich zeigende Verweise besitzen, sogenannte Hard-Links (nicht jedoch Verzeichnisse)

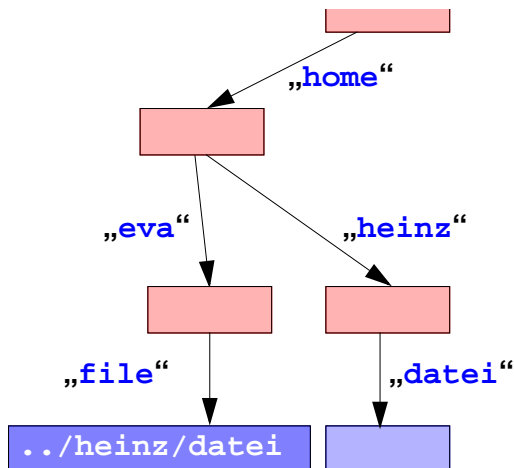


- ◆ Die Datei hat zwei Einträge in verschiedenen Verzeichnissen, die völlig gleichwertig sind:
`/home/eva/file`
`/home/heinz/datei`
- ◆ Datei wird erst gelöscht, wenn letzter Link gekappt wird.

3.1 Pfadnamen (4)

■ Symbolische Namen (*Symbolic Links*)

- ◆ Verweise auf einen anderen Pfadnamen (sowohl auf Dateien als auch Verzeichnisse)
- ◆ Symbolischer Name bleibt auch bestehen, wenn Datei oder Verzeichnis nicht mehr existiert



- ◆ Symbolischer Name enthält einen neuen Pfadnamen, der vom FS interpretiert wird.

3.2 Eigentümer und Rechte

- Eigentümer
 - ◆ Jeder Benutzer wird durch eindeutige Nummer (UID) repräsentiert
 - ◆ Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die durch eine eindeutige Nummer (GID) repräsentiert werden
 - ◆ Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
 - ◆ Lesen, Schreiben, Ausführen (nur vom Eigentümer veränderbar)
 - ◆ einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
 - ◆ Lesen, Schreiben (Löschen u. Anlegen von Dateien etc.), Durchgangsrecht
 - ◆ Schreibrecht ist einschränkbar auf eigene Dateien („nur erweiterbarer“)

3.3 Dateien

- Basisoperationen
 - ◆ Öffnen einer Datei

```
int open( const char *path, int oflag, [mode_t mode] );
```

 - Rückgabewert ist ein Filedescriptor, mit dem alle weiteren Dateioperationen durchgeführt werden müssen.
 - Filedescriptor ist nur prozesslokal gültig.
 - ◆ Sequentielles Lesen und Schreiben

```
ssize_t read( int fd, void *buf, size_t nbytes );
```

Gibt die Anzahl gelesener Zeichen zurück

```
ssize_t write( int fd, void *buf, size_t nbytes );
```

Gibt die Anzahl geschriebener Zeichen zurück

3.3 Dateien (2)

■ Basisoperationen (2)

◆ Schließen der Datei

```
int close( int fd );
```

■ Fehlermeldungen

◆ Anzeige durch Rückgabe von -1

◆ Variable `int errno` enthält Fehlercode

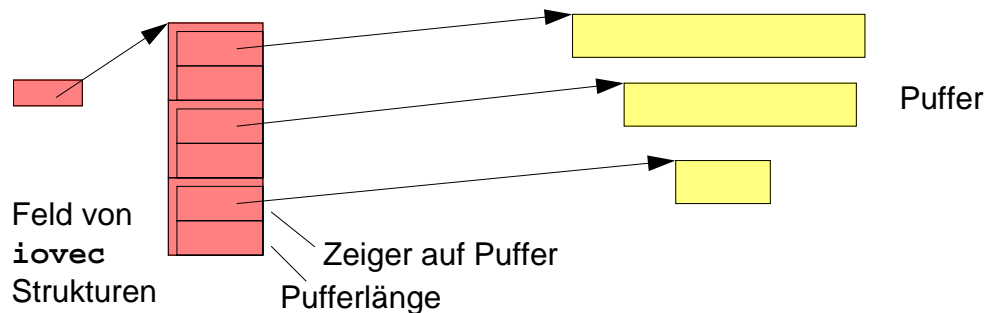
◆ Funktion `perror("")` druckt Fehlermeldung bzgl. `errno` auf die Standard-Ausgabe

3.3 Dateien (2)

■ Weitere Operationen

◆ Lesen und Schreiben in Pufferlisten

```
int readv( int fd, const struct iovec *iov, int iovcnt );  
int writev( int fd, const struct iovec *iov, int iovcnt );
```



◆ Positionieren des Schreib-, Lesezeigers

```
off_t lseek( int fd, off_t offset, int whence );
```

3.3 Dateien (3)

■ Attribute einstellen

◆ Länge

```
int truncate( const char *path, off_t length );  
int ftruncate( int fd, off_t length );
```

◆ Zugriffs- und Modifikationszeiten

```
int utimes( const char *path, const struct timeval *tvp );
```

◆ Implizite Maskierung von Rechten

```
mode_t umask( mode_t mask );
```

◆ Eigentümer und Gruppenzugehörigkeit

```
int chown( const char *path, uid_t owner, gid_t group );  
int lchown( const char *path, uid_t owner, gid_t group );  
int fchown( int fd, uid_t owner, gid_t group );
```

3.3 Dateien (4)

◆ Zugriffsrechte

```
int chmod( const char *path, mode_t mode );  
int fchmod( int fd, mode_t mode );
```

◆ Alle Attribute abfragen

```
int stat( const char *path, struct stat *buf );  
Alle Attribute von path ermitteln (folgt symbolischen Links)
```

```
int lstat( const char *path, struct stat *buf );  
Wie stat, folgt aber symbolischen Links nicht
```

```
int fstat( int fd, struct stat *buf );  
Wie stat, aber auf offene Datei
```

3.4 Kataloge

■ Kataloge verwalten

◆ Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

◆ Löschen

```
int rmdir( const char *path );
```

◆ Hard Link erzeugen

```
int link( const char *existing, const char *new );
```

◆ Symbolischen Namen erzeugen

```
int symlink( const char *path, const char *new );
```

◆ Verweis/Datei löschen

```
int unlink( const char *path );
```

3.4 Kataloge (2)

■ Kataloge auslesen

◆ Öffnen, Lesen und Schließen wie eine normale Datei

◆ Interpretation der gelesenen Zeichen ist jedoch systemabhängig, daher wurde eine systemunabhängige Schnittstelle zum Lesen definiert:

```
int getdents( int fildes, struct dirent *buf,  
             size_t nbyte );
```

◆ Zum [einfacheren](#) Umgang mit Katalogen gibt es Bibliotheksfunktionen:

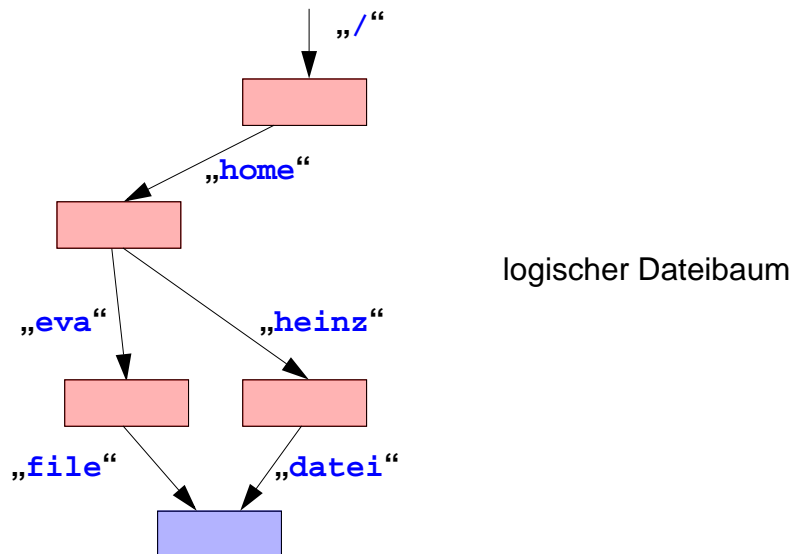
```
DIR *opendir( const char *path );  
struct dirent *readdir( DIR *dirp );  
int closedir( DIR *dirp );  
long telldir( DIR *dirp );  
void seekdir( DIR *dirp, long loc );
```

■ Symbolische Namen auslesen

```
int readlink( const char *path, void *buf, size_t bufsiz );
```

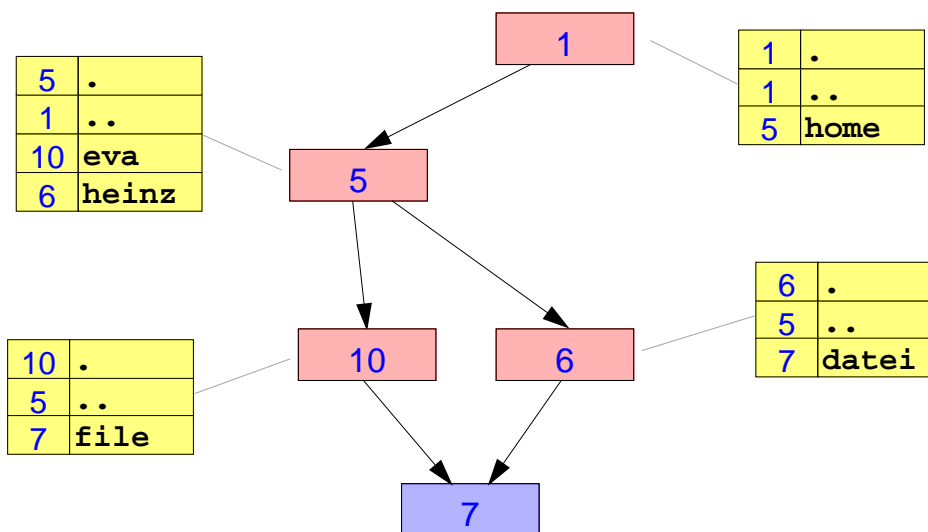
3.5 Inodes

- Attribute einer Datei und Ortsinformationen über ihren Inhalt werden in sogenannten Inodes gehalten
- ◆ Inodes werden pro Partition nummeriert (*Inode Number*)



3.5 Inodes (2)

- Kataloge enthalten lediglich Paare von Namen und Inode-Nummern

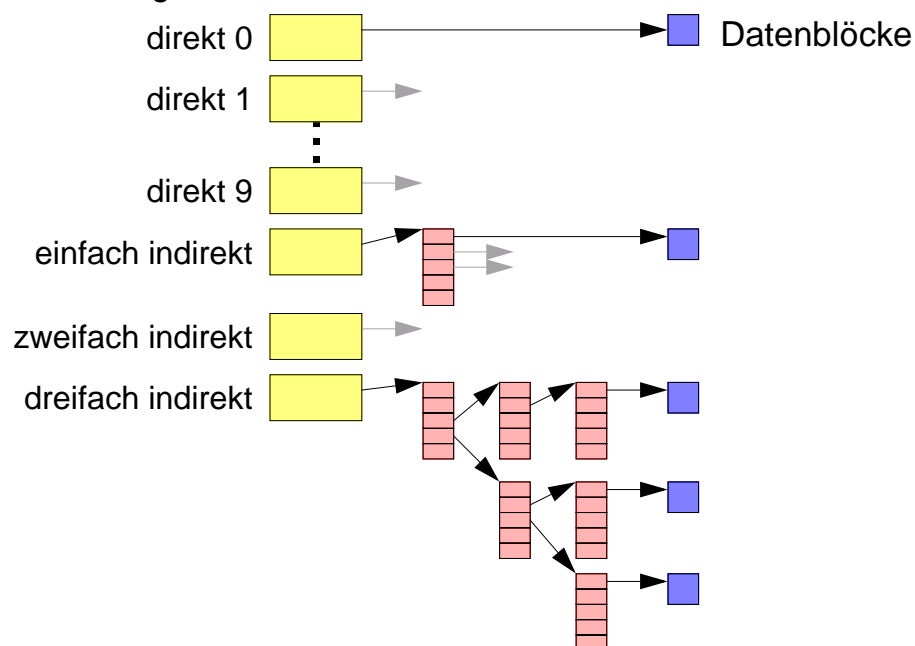


3.5 Inodes (3)

- Inhalt eines Inodes
 - ◆ Inodenummer
 - ◆ Dateityp: Katalog, normale Datei, Spezialdatei (z.B. Gerät)
 - ◆ Eigentümer und Gruppe
 - ◆ Zugriffsrechte
 - ◆ Zugriffszeiten: letzte Änderung (*mtime*), letzter Zugriff (*atime*), letzte Änderung des Inodes (*ctime*)
 - ◆ Anzahl der Hard links auf den Inode
 - ◆ Dateigröße (in Bytes)
 - ◆ Adressen der Datenblöcke des Datei- oder Kataloginhalts (zehn direkt Adressen und drei indirekte)

3.5 Inodes (4)

- Adressierung der Datenblöcke



3.6 Spezialdateien

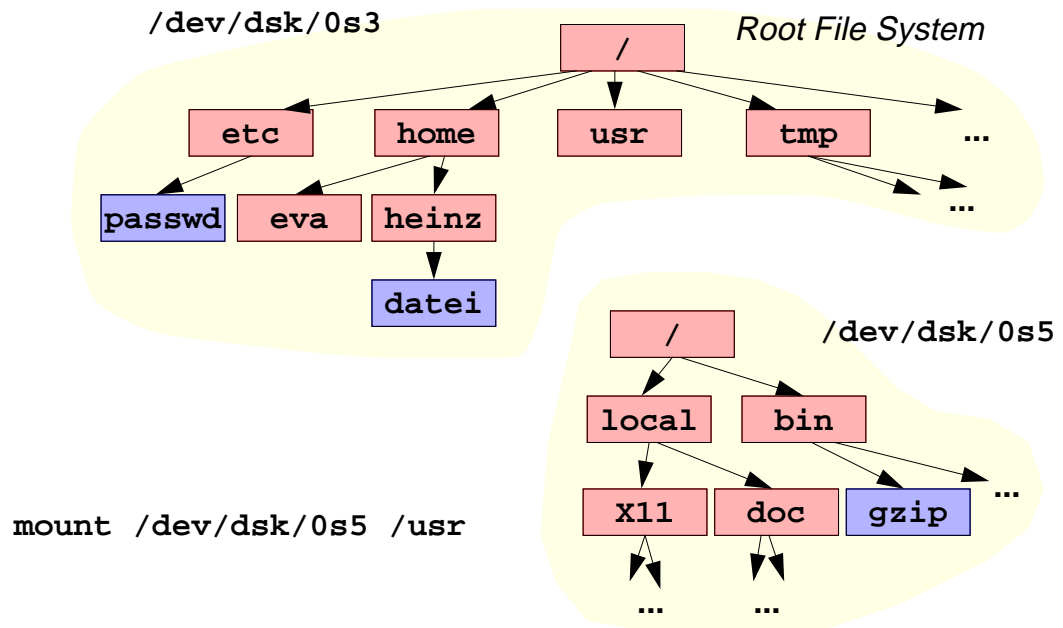
- Periphere Geräte werden als Spezialdateien repräsentiert
 - ◆ Geräte können wie Dateien mit Lese- und Schreiboperationen angesprochen werden
 - ◆ Öffnen der Spezialdateien schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch einen Treiber hergestellt wird
- Blockorientierte Spezialdateien
 - ◆ Plattenlaufwerke, Bandlaufwerke, Floppy Disks, CD-ROMs
- Zeichenorientierte Spezialdateien
 - ◆ Serielle Schnittstellen, Drucker, Audiokanäle etc.
 - ◆ blockorientierte Geräte haben meist auch eine zusätzliche zeichenorientierte Repräsentation

3.7 Montieren des Dateibaums

- Der UNIX-Dateibaum kann aus mehreren Partitionen zusammenmontiert werden
 - ◆ Partition wird Dateisystem genannt (*File system*)
 - ◆ wird durch blockorientierte Spezialdatei repräsentiert (z.B. `/dev/dsk/0s3`)
 - ◆ Das Montieren wird *Mounten* genannt
 - ◆ Ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig Wurzelverzeichnis des Gesamtsystems ist
 - ◆ Andere Dateisysteme können mit dem Befehl `mount` in das bestehende System hineinmontiert werden

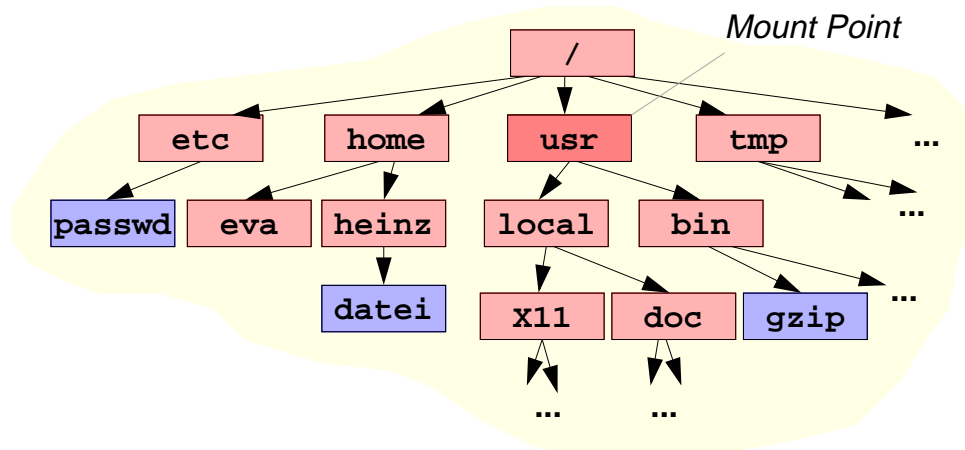
3.7 Montieren des Dateibaums (2)

■ Beispiel



3.7 Montieren des Dateibaums (3)

■ Beispiel nach Ausführung des Montierbefehls



4 Beispiel: Windows 95 (VFAT, FAT32)

- VFAT = Virtual (!) File Allocation Table (oder FAT32)
 - ◆ VFAT: MS-DOS-kompatibles Dateisystem mit Erweiterungen
- Datei
 - ◆ einfache, unstrukturierte Folge von Bytes
 - ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
 - ◆ dynamisch erweiterbar
 - ◆ Zugriffsrechte: „nur lesbar“, „schreib- und lesebar“

4 Beispiel: Windows 95 (VFAT, FAT32) (2)

- Partitionen heißen Laufwerke
 - ◆ Sie werden durch einen Buchstaben dargestellt (z.B. c:)
- Verzeichnis
 - ◆ baumförmig strukturiert
 - Knoten des Baums sind Verzeichnisse
 - Blätter des Baums sind Dateien
 - ◆ jedem Windows-Programm ist zu jeder Zeit ein aktuelles Laufwerk und ein aktuelles Verzeichnis pro Laufwerk zugeordnet
 - ◆ Zugriffsrechte: „nur lesbar“, „schreib- und lesbar“