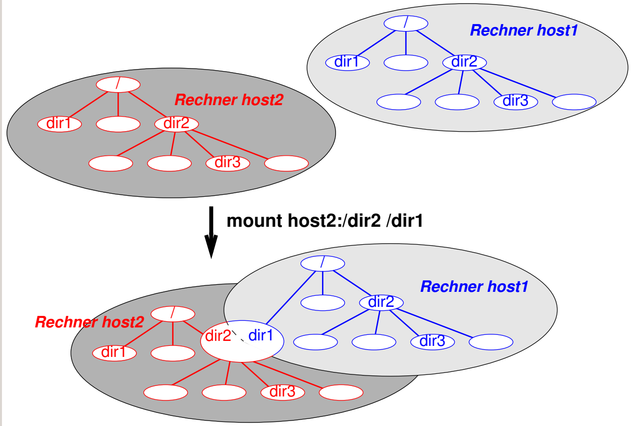
1 Überblick (2)

- Abweichungen von der UNIX-Dateisemantik:
  - ➤ ***special files*** (Geräte-Dateien) sind nicht erreichbar
  - ➤ es werden nicht alle Systemdienste unterstützt (z. B. *File-locks*)
  - ➤ ***append***-Modus beim Öffnen einer Datei nicht möglich
  - ➤ wenn ein Dateisystem voll ist, wird dies nicht unmittelbar mitgeteilt
- auf der *Client*-Seite sind die NFS-Funktionen in den UNIX-Systemkern integriert **(virtuelles Dateisystem - *VFS*)**
- NFS-*Server* können, müssen aber nicht im Betriebssystem-Kern ablaufen
- Kommunikation zwischen *NFS-Client* und *-Server* erfolgt über ***SUN-RPC*** *(Remote Procedure Calls),* aufbauend auf ***UDP/IP*** oder (Solaris 2.5, 4.4bsd) ***TCP/IP***

1 Überblick (3)

- *NFS-Server* sind zustandslos, d. h. sie halten keine Informationen über frühere Aufträge eines *Clients* (z. B. welche Dateien geöffnet, welche Position in einer Datei)
  → problemloser Neustart nach Systemabstürzen
- standardisierte Datendarstellung für Datenaustausch zwischen unterschiedlichen Rechnerarchitekturen:
  XDR *(e**X**ternal **D**ata **R**epresentation)*
- die Spezifikationen für NFS, XDR und RPC ist *public domain*, d. h. frei erhältlich - nicht jedoch die Implementierungen
- Programme werden immer auf dem *aufrufenden* Rechner ausgeführt
  - ➤ Betrieb von *diskless workstations* möglich
- Probleme bei unterschiedlichen Rechnertypen - unterschiedlich übersetzte Programme *(Binaries)* notwendig

2 Beispiel



3 Sicherheitsaspekte

- Dateisysteme müssen explizit exportiert werden um von anderen Rechnern *gemountet* werden zu können
  - ➤ der Systemverwalter des exportierenden Rechners kann festlegen, welche Rechner welches Dateisystem von ihm importieren dürfen
  - ➤ bei ***secure NFS*** kann darüber hinaus auch spezifiziert werden, welche Benutzer des anderen Rechners zugreifen dürfen
- die Benutzerabbildung erfolgt in UNIX über die *User-Id*
  - ➤ Beispiel: der Benutzer mit der *User-Id* 129 auf Rechner *host1* hat in einem Dateisystem, das von Rechner *host2* importiert ist, genau die Rechte, die der Benutzer mit *User-Id* 129 auf dem Rechner *host2* hat
- Der *Super-User* (*User-Id* 0) wird auf den Benutzer *nobody* abgebildet (die *Super-User*-Abbildung kann pro exportiertes Dateisystem/ Zielrechner auch explizit festgelegt werden)

4 Implementierung

- NFS-Server erzeugen beim ***nfs_open()*** ein **file-handle** (Kombination aus Filesystem/Device-Id und Inode) und übergeben dies dem NFS-Client
  - ◆ es werden auf Server-Seite keine Informationen gehalten!
  - ◆ bei jedem Lese-/Schreib-Zugriff wird die Datei auf Basis der file-handle-Informationen neu geöffnet
- NFS-Client benutzt file-handle für alle weiteren Zugriffe auf die geöffnete Datei (Block lesen/schreiben)
  - ◆ file-handle wird im **rnode**-Teil des vnode gespeichert
  - ◆ Offset-Informationen sind in der file-Struktur vorhanden
  - ◆ vfs-Struktur enthält die Adresse des NFS-Servers

J.8 Block-Puffer Cache

1 Verwaltungsstrategien

- Pufferverwaltungssystem mit Verwaltungsalgorithmen, ähnlich denen in einem Paging System (nicht demand paging!) mit folgenden Eigenschaften:
  - ◆ **read ahead**
    wird ein Block gelesen, so werden auch der Transfer des folgenden Blocks angestoßen (wenn sequentiell gelesen wird)
  - ◆ **write after**
    wird ein Block geschrieben, bedeutet dies nicht den sofortigen physikalischen Transfer auf die Platte
  - ◆ **geschrieben** wird u. a. bei folgenden Ereignissen:
    - ➤ Datei wird geschlossen
    - ➤ keine freien Puffer mehr vorhanden
    - ➤ vom System *(clock)* wird ein ***update*** angestoßen
    - ➤ Systemaufruf ***sync(2)*** oder ***fsync(2)*** (BSD)

4 Implementierung (2)

- Besonderheiten:
  - ◆ wird eine Datei auf Client-Seite gelöscht, obwohl noch Prozesse gültige Dateideskriptoren darauf halten:
    - ➤ wird sie auf der Serverseite nur umbenannt (nfs_...)
    - ➤ wird sie erst endgültig gelöscht, wenn die letzte Referenz auf den vnode auf der Client-Seite geschlossen wird

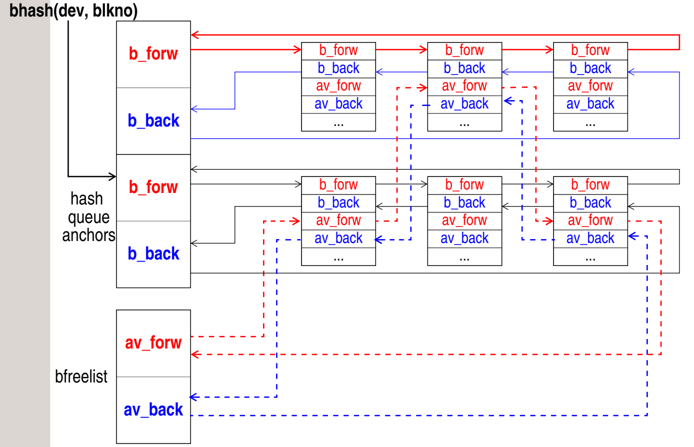
1 Verwaltungsstrategien (2)

- ◆ werden ausgelagerte Blöcke, deren Puffer noch nicht anderweitig benutzt wurde, erneut angesprochen, so werden sie reaktiviert ***(reclaim)***
- ◆ wenn zur Gewinnung von freien Blöcken eine Auslagerung angestoßen wird, werden die Kandidaten für die Freiliste nach dem LRU-Algorithmus ausgewählt

2 Integration von Buffer Cache und Speicherverwaltung

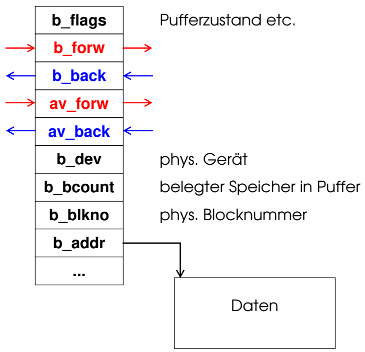
- Ab SystemV R4 werden die Puffer des Dateisystems mit dem normalen Paging-System verwaltet
  - ◆ Dateiblöcke werden in virt. Adreßraum des Kerns abgebildet
  - ◆ Daten werden dann einfach aus Kern in User-level Adreßraum kopiert
    - ➤ Page-fault bewirkt Einlagerung der Seite aus der Datei
      (über vnode-Operationen des jeweiligen Dateisystems)

3 Kontrollstrukturen



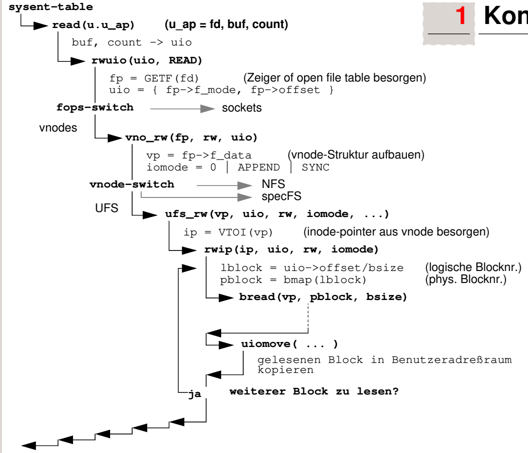
3 Kontrollstrukturen

- in BSD4.4 sind die Puffer zusätzlich an den *vnodes* verankert
- Buffer-Header (struct buf, siehe *<sys/buf.h>)*



J.9 Dateisystem - Buffer Cache

1 Kontrollfluß



J.10 Treiber

1 Überblick

- Jede Gerätedatei entspricht einem physischen oder virtuellen Gerät
- bei den physischen Geräten werden zwei Ein-Ausgabesysteme unterschieden:
  - ◆ zeichenorientierte E/A (Terminals, Drucker)
  - ◆ blockorientierte E/A (vor allem Platten)
- für jedes Ein-Ausgabesystem existiert ein:
  - ◆ Puffersystem
  - ◆ Treiberfunktionsvektor, *cdevsw* und *bdevsw*

1 Überblick (2)

- Für jeden Gerätetyp ist in mindestens einem dieser Vektoren eine Zeile mit den gerätespezifischen Operationen reserviert
- Jedes Gerät ist mit dem Tripel (E/A-Typ, *major number*, *minor number*) eindeutig beschrieben
  - E/A-Typ wählt Funktionsvektor aus
  - *major number* wählt die zugehörige Zeile im Funktionsvektor aus
  - *minor number* dient im gerätespezifischen Kontrollblock des Treibers als Index zur Identifikation des Geräts

2 Treiber-Funktionsvektoren

- *block-device-switch*:

```
struct bdevsw {
        int         (*d_open)();
        int         (*d_close)();
        int         (*d_strategy)();
        int         (*d_dump)();
        int         (*d_psize)();
        int         d_flags;
};
```

  - *read*- und *write*-Aufrufe werden über den Block-Puffer abgewickelt
  - die *strategy*-Funktion regelt die E/A-Operation zwischen Block-Puffer und Gerät
  - *major*- und *minor number* werden im Inode gespeichert

2 Treiber-Funktionsvektoren (2)

- *character -device-switch*:

```
struct cdevsw {
        int      (*d_open)();
        int      (*d_close)();
        int      (*d_read)();
        int      (*d_write)();
        int      (*d_ioctl)();
        int      (*d_reset)();
        int      (*d_select)();
        int      (*d_mmap)();
        struct   streamtab *d_str;
};
```

  - im Falle von *Streams*-Treibern sind alle Komponenten leer - nur die *streamtab*-Komponente enthält einen Verweis auf die Struktur der *Streams*-Verwaltung
- Block-orientierte Geräte (z. B. Platten) haben in der Regel einen zweiten Eintrag als zeichen-orientiertes Gerät um direkte Zugriffe unter Umgehung der Block-Puffer zu ermöglichen

2 Treiber-Funktionsvektoren (3)

- der Systemaufruf *mknod* legt im Dateisystem Knoten für die Gerätedateien an
- Beispiel aus /dev

```
% ls -l /dev
crw--w--w-  1 root      0,   0 Feb  5 12:38 console
crw-r-----  1 root      3,   1 Dec  2 11:48 kmem
crw-rw-rw-  1 root     13,   0 Sep 15  1988 mouse
crwx-w----  1 eckert   20,   5 Feb  5 12:07 ttyp5
crw-rw-rw-  1 root     21,   5 Feb  5 12:06 ptyp5
brw-r-----  1 root      7,   1 Sep 15  1988 sd0b
crw-r-----  1 root     17,   1 Sep 15  1988 rsd0b
brw-r-----  1 root      3,   0 Dec  2  1988 xy0a
crw-r-----  1 root      9,   0 Jan 10 10:34 rxy0a
crw--w--w-  1 root     12,   0 Feb  5 12:11 ttya
crw--w--w-  1 root     12,   1 Jan 11 09:58 ttyb
```

2 Treiber-Funktionsvektoren (4)

- Auszug aus der Konfigurationsdatei des Systemkerns

```
struct bdevsw   bdevsw[] =
{
  { nodev,      nodev,     nodev,       nodev,      /*0*/
    0,          0},                           /* was ip */
  { tmopen,     tmclose,   tmstrategy,  tmdump,     /*1*/
    0,          B_TAPE },
  { nodev,      nodev,     nodev,       nodev,      /*2*/
    0,          B_TAPE},                      /* was ar */
  { xyopen,     nulldev,   xystrategy,  xydump,     /*3*/
    xysize,     0 },
  ...
  { sdopen,     nulldev,   sdstrategy,  sddump,     /*7*/
    sdsize,     0 },
  ...
}
```


J.11 Hintergrundspeicher

1 Organisation einer Disk - Laufwerk

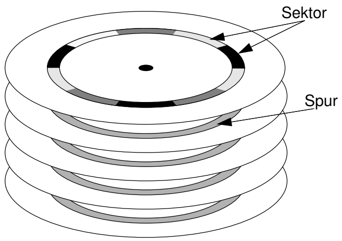
- typische Winchester-Festplatten bestehen aus mehreren, übereinander angeordneten Magnetplatten und einem Schreib-/Lese-Kopf pro Plattenoberfläche

- Begriffe:

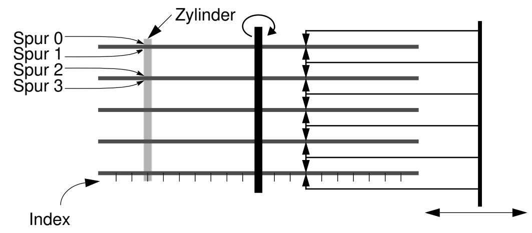
Spur *(Track)* Umlaufbahn auf einer Diskoberfläche

Sektor Teil einer Spur

Zylinder Menge der Spuren mit gleichem Radius über alle Oberflächen



1 Organisation einer Disk - Laufwerk (2)



1 Organisation einer Disk - Laufwerk (3)

- Beispiele realer Disks:

	Fujitsu			*Seagate*	
	M2344K	*M2652*	*Elite9*	*Elite47*	*Cheetah18*
	1987	1990	1994	1998	1998
	9"	9"	5,25"	5,25"	3,5"
Bussystem:	SMD	IPI	SCSI-2	Ultra SCSI	Fibre
Kapazität:	690 MB	2,0GB	10,8GB	47 GB	18GB
Zylinder:	624	1893	4925	9996	6962
Spuren/Zyl.:	27	20	27	28	24
Bytes/Spur:	40960	52864	variabel		
Sektoren/Spur:	1 - 128 (konfigurierbar)				
Positionierzeit:					
(Spur-Spur)	4 ms	2 ms		1,1 ms	0,6 ms
(mittel)	16 ms	11 ms	11 ms	13 ms	5,7 ms
(max)	33 ms	22 ms		28 ms	12 ms
Transfer MB/s:	2,458	4,75	6 - 7	40	200
Umdr./Min.:	3600	5400	5400	5357	10025

➥ was bedeutet das konkret (z.B. bei Elite9)?
- eine Umdrehung = 11 ms
- bei DEC Alpha (300 MHz, 2 Instr./Takt = 600 Mio/Sek.) ca. 6,6 Mio. Befehle ausführbar

2 Organisation einer Disk (SCSI)

- Platte = Sequenz von Datenblöcken (perfect media)
- lokaler Pufferspeicher (z. B. 1-4 MB) für Optimierungsstrategien (da Bus schneller als Medium-Transfer)

3 Organisation einer Disk — Betriebssystem

- UNIX kann eine Disk in **Partitionen** unterteilen
  - ➥ auf jeder Partition kann ein UNIX-Dateisystem angelegt werden
- jedes UNIX-Dateisystem enthält eigene, von anderen Partitionen unabhängige Inodes
- aus diesen Inodes wird auf jedem UNIX-Dateisystem über Directories ein eigener, hierarchisch aufgebauter Dateibaum erzeugt
- mehrere solche Bäume können durch ***mounten*** zu einem Dateibaum zusammengefaßt werden

3 Organisation einer Disk — BS (2)

- die Position der einzelnen Partitionen wird festgelegt:
  - in älteren UNIX-Systemen statisch im Kern
  - in neueren UNIX-Systemen in Tabelle (*VTOC - Volume Table of Contents*) am Anfang der Platte (z. B. Zyl. 0, Spur 0, Sekt. 0)
- Partitionierung am Beispiel Solaris

```
Current partition table (original):
Total disk cylinders available: 2733 + 2 (reserved cylinders)

Part      Tag  Flag     Cylinders      Size          Blocks
 0       root   wm       0 -  101     75.70MB   (102/0/0)    155040
 1       swap   wu     102 -  446    256.05MB   (345/0/0)    524400
 2     backup   wm       0 - 2732      1.98GB   (2733/0/0)  4154160
 3 unassigned   wm       0             0        (0/0/0)           0
 4 unassigned   wm       0             0        (0/0/0)           0
 5 unassigned   wm       0             0        (0/0/0)           0
 6        usr   wm     447 - 2732      1.66GB   (2286/0/0)  3474720
 7 unassigned   wm       0             0        (0/0/0)           0
```


3 Organisation einer Disk — BS (3)

- den einzelnen Partitionen sind jeweils *block special files* **(/dev/dsk)** und *character special files* **(/dev/rdsk)** zugeordnet
  - ➥ über diese Dateien sind *major-number* (=Disk-Typ) und *minor-number* (=Partition) zur Interaktion mit dem Treiber feststellbar
  - ➥ die *block special files* sind in *mount*-Kommandos verwendbar
  - ➥ über die *character special files* ist ungepufferter Zugriff auf eine Partition möglich (z. B. für file system check)
  - ➥ Beispiel Solaris:

```
% ls -lL /dev/dsk
brw-r-----   1 root      32,   0 Oct 23 17:21 c0t0d0s0
brw-r-----   1 root      32,   1 Oct 23 17:21 c0t0d0s1
brw-r-----   1 root      32,   2 Oct 23 17:21 c0t0d0s2
...
brw-r-----   1 root      32,   6 Oct 23 17:21 c0t0d0s6
brw-r-----   1 root      32,   7 Oct 23 17:21 c0t0d0s7

% fsck /dev/rdsk/c0t0d0s6
% mount /dev/dsk/c0t0d0s6 /usr
```


4 Disk-Zugriffsstrategien

- die Verwaltungsstrategien des *Buffer Cache* haben u. a. zwei Auswirkungen auf den Platten-Treiber
  - ◆ weniger Plattenzugriffe bei häufigen Operationen auf einer Datei
  - ◆ Anforderungen an den Plattentreiber kommen nicht einzeln, sondern immer in Gruppen (z. B. wenn Auslagerung angestoßen wurde)
- der Plattendurchsatz kann gesteigert werden, wenn die Anforderungen geeignet sortiert werden
- Problem bei modernen Disks:
  - ➤ Disk hat internen Pufferspeicher und fährt eine eigene Strategie für Zugriffe auf das Medium

4 Disk-Zugriffsstrategien (2)

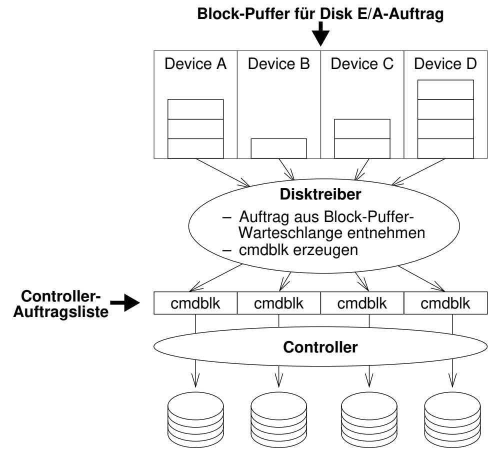
- mögliche Strategien:
  - ◆ **FIFO**
    - \+ Aufträge werden fair behandelt
    - – Plattenzugriffe nicht optimal, da viele Positionierungen erforderlich
  - ◆ **SATF** *(shortest access time first)*
    - \+ optimiert die Positionierungen des Plattenarms
    - – bei vielen Aufträgen für einen begrenzten Bereich können aber Aufträge für entferntere Positionen ausgehungert werden
  - ◆ **SCAN**
    - \+ Aufträge werden nach Zylinderpositionen sortiert, so daß der Plattenarm der Reihe nach über alle Positionen bewegt wird
    - – viele Aufträge für einen kleinen Bereich benachteiligen entfernte Aufträge
  - ◆ gemischte Strategien
    z. B. SATF über die ersten 8 Aufträge → Mischung aus SATF und FIFO

J.12 Disk-Treiber: Datenstrukturen und Kontrollfluß

1 Überblick

- Jedem Gerät (Platte) ist eine Warteschlange für Puffer des Buffer-Caches zugeordnet
- Puffer für Aufträge an den Disktreiber werden in die Warteschlange des zugehörigen Geräts einsortiert (*disksort( )*)
- Disktreiber entnimmt für freie Geräte Aufträge und startet den Controller
- bearbeitete Aufträge werden durch Interrupt gemeldet
  - ➥ Interrupt-Bearbeitung
    - – eigener Aktivitätsträger
    - – unabhängig von den beauftragenden Prozessen!
      - • weckt wartende Prozesse
      - • startet nächsten Auftrag aus Warteschlange

2 Datenstrukturen



3 Kontrollfluß