(1) Dateiorganisation (MSDOS, UNIX)

Bei der Ausgabe auf eine Datei in einem Programm wird z.B. in PASCAL programmiert:

```
Assign(f1,'D1.TXT');
Reset(f1);
Readln(f1,s);
```

In JAVA analog:

```
FileInputStream f1=new FileInputStream('D1.TXT');
bytes_read=f1.read(data,0,80);
```

Hier mit soll die erste Zeile einer Datei eingelesen werden. Dieser logische Zugriff muss vom Programmier- und Betriebssystem auf einen physikalischen Zugriff auf z.B. eine Festplatte umgewandelt werden.

Die Festplatte ist aufgeteilt in
Nb Oberflächen
Na Spuren
Nc Sektoren/Spur

Jeder Sektor enthält eine Anzahl von Bytes z.B. ns=512.
Die Hardwarebefehle lesen bzw. Schreiben einen Sektor c in Spur a auf Oberfläche b.

Die physikalische Platte wird als dargestellt durch:

```
Platte = array[0..na-1,0..nb-1,0..nc-1]of Sektor
Sektor = array[0..ns-1]of bytes.
```

Die logische Platte ist dagegen

```
Log-Platte:array[0..(na*nb*nc)-1]of Sektor;
```

Die Abbildung von der physikalischen Sektoren in die logischen Sektoren ist:

```
Lognr(a,b,c)=c+nc*(b+nb*a);
```

D.h. bei fortlaufenden Sektoren wird zunächst eine Spur beschrieben, dann alle Oberflächen auf der gleichen Spur und erst dann werden die Köpfe zur nächsten Spur bewegt. Dadurch werden die Kopfbewegungen minimiert.

Soll die logische Sektornummer i nach a,b,c transformiert werden:

```
c:=i mod nc;
b:=(i div nc)mod nb;
a:=(i div nc)div nb;
```

Um die Verwaltung zu vereinfachen werden Blöcke aus d Sektoren eingeführt (Cluster).
Um eine unbekannte Platte lesen zu können, ist es notwendig die Werte für na, nb und nc an einer definierten Stelle zu finden.

Aus Sicht des Betriebssystems werden die Platten jedoch nicht als ein Feld von Sektoren aufgefasst, sondern auf eine Platte werden Dateien geschrieben, die über ihren Namen angesprochen werden. Die Dateien sollen verlängert, verkürzt oder gelöscht werden können. Daher ist es nicht möglich für eine Datei einen festen Bereich auf der Platte zu reservieren. Aufgabe des **Dateisystem** ist es diese Verwaltung vorzunehmen.

Für eine Datei stellt das Dateisystem folgende Funktionen zur Verfügung:

- Erzeugen der Datei
- Löschen der Datei
- Umbenennen einer Datei
- Öffnen der Datei
- Positionieren in der Datei
- Lesen ab der eingestellten Position
- Schreiben ab der eingestellten Position
- Löschen des Inhalts hinter der eingestellten Position
- Schließen der Datei

Besondere Zugriffe, wie das Satz weise Lesen einer Textdatei oder ein Indexsequentieller Zugriff wird nicht durch das Dateisystem geleistet sondern von den Anwendungsprogrammen, z.B. dem Laufzeitsystem eines Compilers bzw. einem Datenbanksystem.

Unter DOS und Windows benutzt man das FAT-Dateisystem. FAT steht für File-Allocation-Table.

```
FAT:array[0..ncluster-1]of (d)word;
```

Für jeden Cluster der Platte gibt es einen Eintrag inder FAT.
Dieser kennzeichnet normal die Nummer des Clusters bei dem die Datei fortgesetzt wird.
Besondere Nummern sind reserviert für:

- Letztes Cluster einer Datei
- Bad-Cluster
- Leeres Cluster

Die FAT liefert für jede Datei eine verkettete Liste der Cluster. Die Nummer des ersten Cluster befindet sich im Eintrag der Datei im Dateiverzeichnis.

Dateiverzeichnis

Jede Platte(Partition) hat ein Root(Wurzel)-Verzeichnis. Die Position des Rootverzeichnis liegt fest, ebenso die Zahl der Einträge.

Ein Dateieintrag im Verzeichnis ist ein record, der folgende Angaben enthält:

- Dateiname
- Attribute
- Zeit und Datum der letzten Änderung
- Anzahl der Bytes
- Nummer des 1. Cluster

Wird eine Datei angelegt, wird ein neuer Dateieintrag erzeugt und ein leeres Cluster gesucht. Diese Nummer wird im Dateieintrag eingetragen, in dem FAT-Eintrag wird Letztes Cluster eingetragen. Die Dateilänge wird auf 0 gesetzt.
Reicht beim Schreiben dieses Cluster nicht mehr aus, wird ein neues Cluster gesucht und dessen Nummer in den Eintrag des ersten Cluster geschrieben.

Um die Dateien besser zu organisieren benötigt man Unterverzeichnisse. Ein Unterverzeichnis ist ein normaler Dateieintrag im Vaterverzeichnis mit dem Attribut Directory. Jedes Verzeichnis hat 2 besondere Einträge:

`..` Clusternummer des Vaterverzeichnisses
`.` Clusternummer des aktuellen Verzeichnisses

Lesen der Byte `i..i+n-1` einer Datei:

```
iCluster:=i div clustersize;
ClusterNr:=DirectoryEintrag.Nummer des 1. Cluster;
for i:=1 to iCluster do ClusterNr:=FAT[ClusterNr];
```

Über die Anzahl der Sektoren/Cluster kann die Nummer des ersten Sektors des Clusters ermittelt werden. Über `i mod Clustersize` erhält man die Position von i im Cluster. Daraus kann die Nummer des Sektors im Clusters ermittelt werden, die zur Nummer des ersten Sektors addiert werden muss. Mit `i mod Sektorsize` erhält man die Position von i im Sektor.

Beim Löschen einer Datei wird dies zunächst nur im Verzeichniseintrag vermerkt. Dadurch ist es möglich das Löschen wieder rückgängig zu machen. Erst wenn keine freien Cluster mehr gefunden werden, werden die Cluster der gelöschten Dateien benutzt.

Um das Lesen bzw. Schreiben kleiner Mengen zu beschleunigen benutzen die Programmiersysteme zum Teil Puffer, insbesondere bei Textdateien.

Es wäre sehr ineffektiv, wenn bei jedem Lesen bzw. Schreiben jeweils über den Verzeichniseintrag der benutzte Sektor ermittelt würde. Daher arbeitet das Dateisystem mit Dateihandle. Beim Öffnen einer Datei wird ein Handle zurückgeliefert. Über diesen Handle werden alle Lese- und Schreiboperationen abgewickelt. Der Handle ist ein Index in den Job-File-Table (JFT). Der Eintrag im JFT ist ein Index in den System-File-Table (SFT).

Ein Eintrag im SFT ist eine Datenstruktur, die u.a. folgende Angaben enthält:

- StartCluster
- Aktuelle Position
- Nummer des aktuellen Clusters
- Anzahl der Prozesse, die Datei bearbeiten (handlecount)

Damit kann bei den Lese bzw. Schreiboperationen schnell der aktuelle Sektor gefunden werden.

Bei der Open-Funktion in DOS bzw. Windows finden daher folgende Schritte statt:

1. Freien Eintrag im JFT suchen und den Index als Handle abliefern
2. Freien Eintrag im SFT suchen (handlecount=0) und den Index im JFT eintragen
3. Über den Pfad die Verzeichnisse durchlaufen bis der Dateieintrag gefunden bzw. neuen Eintrag einrichten.
4. handlecount im SFT auf 1 setzen
5. Informationen aus dem Verzeichniseintrag in den SFT Eintrag kopieren. Aktuelle Position auf 0 setzen.

Bei der Bearbeitung wird zur Beschleunigung nicht jeweils jeder Sektor jeweils neu gelesen, sondern die Sektoren werden in einen Diskbuffer geschrieben. Die Anzahl der Puffer wird in config.sys mit z.B. BUFFERS=128 gesetzt.

Beim Lesen bzw. Schreiben wird jeweils der Handle benutzt.

```
HFILE OpenFile(
  LPCSTR lpFileName,        // file name
  LPOFSTRUCT lpReOpenBuff,  // file information
  UINT uStyle               // action and attributes
);
```

Die geöffnete Datei kann positioniert werden.

```
DWORD SetFilePointer(
  HANDLE hFile,                // handle to file
  LONG lDistanceToMove,        // bytes to move pointer
  PLONG lpDistanceToMoveHigh,  // bytes to move pointer
  DWORD dwMoveMethod           // starting point
);
```

Lesen und Schreiben ab der aktuellen Position:

```
BOOL ReadFile(
  HANDLE hFile,                // handle to file
  LPVOID lpBuffer,             // data buffer
  DWORD nNumberOfBytesToRead,  // number of bytes to read
  LPDWORD lpNumberOfBytesRead, // number of bytes read
  LPOVERLAPPED lpOverlapped    // overlapped buffer
);

BOOL WriteFile(
  HANDLE hFile,                  // handle to file
  LPCVOID lpBuffer,              // data buffer
  DWORD nNumberOfBytesToWrite,   // number of bytes to write
  LPDWORD lpNumberOfBytesWritten,  // number of bytes written
  LPOVERLAPPED lpOverlapped      // overlapped buffer
);
```

Schließen der Bearbeitung:

```
BOOL CloseHandle(
  HANDLE hObject   // handle to object
);
```

Nachteile des FAT-Dateisystems.

1. Die FAT Tabelle muss im Speicher gehalten werden oder ständig geladen/gesichert werden.
2. Bei großen Platten werden entweder die Cluster sehr groß (viel Verschnitt) oder die FAT Tabelle wird sehr groß.
   Annahme: 32 GByte Platte (2^{35} Byte)
   Alternativen:
   64k Cluster mit je 512 k Byte ($2^{16} * 2^{19}$ Bytes)
   1024k Cluster mit je 32 k Byte ($2^{20} * 2^{15}$ Bytes)
   Hier ist die FAT 4 MB groß
3. Beim Positionieren muss evtl. eine lange Kette sequentiell durchlaufen werden.

Daher benutzen Unix und NTFS hierarchische Listen bzw. Bayer Bäume.

Unix Dateisystem

Verzeichniseintrag: Name der Datei und Nummer des I-Node

I-Node:

Ein I-Node enthält folgende Informationen:
- Typ und Zugriffsrechte
- Anzahl der Hardlinks
- Benutzernummer (UID)
- Gruppennummer (GID)
- Größe der Datei in Bytes
- Datum der letzten Veränderung (mtime)
- Datum der letzten Statusänderung (ctime)
- Datum des letzten Zugriffs (atime)
- Adresse von Datenblock 0 ... Adresse von Datenblock 9
- Adresse des ersten Indirektionsblocks
- Adresse des Zweifach-Indirektionsblocks
- Adresse des Dreifach Indirektionsblocks

Die Adressen der ersten 10 Datenblöcke sind im I-Node selbst gespeichert. Die Adressen der nächsten 128 Datenblöcke befinden sich im ersten Indirektionsblock. Dieser Block enthält 128 Zeiger auf die Datenblöcke 10-137. Der Zweifach-Indirektionsblocks enthält 128 Zeiger die jeweils auf einen Block mit 128 Zeigern zeigen. Beim Dreifach Indirektionsblocks ist dies dann noch eine Stufe mehr.

Eine Datei kann so maximal $10+128+128^2+128^3 > 2^{21}$ Blöcke enthalten. Bei 512 Byte / Block kann die Datei so mehr als 2^{30} Byte = 1 Gbyte enthalten.
Bei einer kleinen Datei besteht der Overhead jedoch nur aus den maximal 12 unnötigen Zeigern.
Soll in der Datei positioniert werden, kann der Zeiger auf den Block direkt berechnet werden.

Datei: `array[0..  ]of byte;`
Gesucht: Byte an Position i

```
lbn:=i div 512;  // logische Nummer des Blockes

if lbn<10 then b:=p[lbn]
else begin
 lbn:=lbn-10;
 if lbn<128 then b:=p[10]^.[lbn]
 else begin
  lbn:=lbn-128;
  if lbn<128*128 then b:=p[11]^.[lbn div 128]^.[lbn mod 128]
  else begin
   lbn:=lbn-128*128;
   b:=p[12]^.[lbn div (128*128)]^.[lbn div 128 mod 128]^.[lbn mod 128];
  end;
 end;
end;
```

`b^[i mod 512]` liefert das gesuchte Byte im Datenblock.

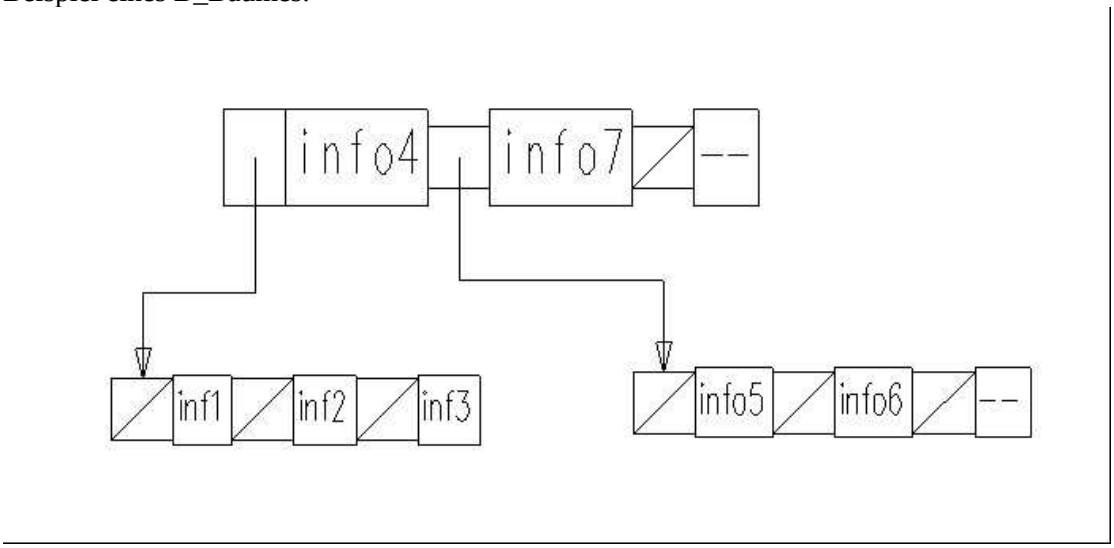
NTFS-Dateisystem

Für jede Datei gibt es einen Eintrag im Master-File-Table (MFT)
Dieser Eintrag ist 1024 Byte groß und enthält:

Standard Information (immer in der MFT)
• enthält Länge, MS-DOS Attribute, Zeitstempel, Anzahl der Hard links
Dateiname (immer in der MFT)
• kann mehrfach vorkommen (Hard links, MS-DOS Name)
Zugriffsrechte
Security descriptor
Daten
• die eigentlichen Daten

Bei größeren Dateien sind die Daten und andere Informationen nicht MFT-Eintrag sondern dieser enthält die Wurzel eines Bayer-Baumes. Bei NTFS ein Element des B-Baumes genau soviel Zeiger wie Informationsblöcke, bei einem Standard B-Baum ist es ein Zeiger mehr.

Beispiel eines B_Baumes:



Auch bei NTFS werden die Platte und die Dateien in Cluster eingeteilt. VCN ist die virtuelle Clusternummer in einer Datei, LCN die fortlaufende logische Clusternummer einer Platte.

Für jede Datei wird bei den Dateiattributen folgende Tabelle gehalten:

```
Starting VCN    Starting LCN    #Cluster
0               1355            4
16              1588            4

Dies hat folgende Belegung zur Folge:
VCN   0    1    2    3   16   17   18   19
LCN 1355 1356 1357 1358 1588 1589 1590 1591
```

Die Daten einer Datei werden in Einheiten von 32kB gehalten, die aus 16 Clustern von je 2kB bestehen.
Eine Einheit kann sein:

- unkomprimiert d.h. sie benötigt 16 Cluster
- komprimiert d.h. sie benötigt <16 Cluster

- sparse d.h. sie enthält lauter 0; es werden überhaupt keine Cluster angelegt und #Cluster=0

Da die Dateilänge bis 2^{64} Byte sein kann, kann so eine Schlange realisiert werden, ohne dass Daten verschoben werden.

Der Server kann vorne die Werte auf 0 setzen. Wird eine Einheit 0 wird #Cluster auf 0 gesetzt und die Cluster werden frei gegeben. Am Ende werden die Daten normal eingefügt. Die Anzahl der benötigten Cluster hängt dann nur von der Länge der Schlange ab.
Wenn der Client 100kByte / Sekunde einfügt braucht man 5 Mio. Jahre um die max. Dateilänge zu erreichen.

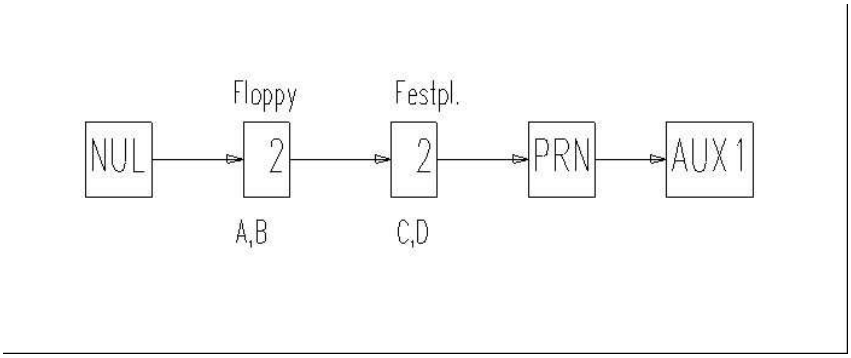
Beispiel: Komprimierte Datei mit sparse Elementen

```
VCN   LCN    #Cluster
 0     19    16          unkomprimiert
16    131     4          komprimiert
32      -     0          sparse
48    200    10          unkomprimiert <32 kByte
```

Zugriff auf Geräte

Bildschirm, Drucker, Serielle oder Parallele Schnittstellen lassen sich mit den gleichen Aufrufen (Open, Read, Write, Close) ansteuern wie Plattendateien.

Open: Es wird in einer Kette der Treiber nachgesehen, ob ein Treiber für diesen Dateinamen (z.B. prn) existiert.
Wenn ja: Aufruf über den Treiber.
Wenn nein: Es wird der Blockorientierte Treiber gewählt, dem der Laufwerksname zugeordnet ist. Die Blockorientierten Treiber belegen fortlaufende Laufwerksbuchstaben von A..Z.



Aktionen bei der Ein / Ausgabe:

- Ermitteln welcher Treiber
- Aufrufkopf (Requestheader) erzeugen.
  Aus den Dateiangaben über Medienbeschreibung und evtl. Pfad ermitteln.
- Strategieroutine mit Requestheader aufrufen.
- **Treiber**: Sichern des Request
- Interruptroutine aufrufen
- **Treiber:** Bearbeitet den Request
  Blockorientiert: log. Sektornummer -> Kopf, Spur, Sektor
  Positionieren, Datenübertragen

Zeichenorientiert: Zeichen Ein / Ausgabe
Statuswert im Requestheader setzen

- Requestheader prüfen

Durch die Aufteilung in Strategieroutine und Interruptroutine ist dies multitaskfähig. Die Requests müssen nicht in der Reihenfolge der Aufrufe abgearbeitet werden.

Eine Operation im Anwendungsprogramm führt evtl. zu mehreren Rquests.
Beispiel: Öffnen einer Datei in einem Unterverzeichnis das noch nicht bearbeitet wurde.

- Lesen Urladesektor für BIOS-Parameterblock
- Lesen Wurzelverzeichnis um Eintrag des Unterverzeichnis zu finden.
- Lesen Unterverzeichnis um Datei zu finden. Dabei evtl. FAT lesen.

Literatur:

Schulman et.al.: Undocumented DOS, Addison-Wesley 1990