Kapitel 15: Dateien und Dateisystem

- Motivation und Einführung
- Dateimanagement
- Dateitypen
- Dateiorganisation und -zugriffsmethoden
- Verzeichnisse (*directories or folders*)
- Implementierung von Dateisystemen
    - Dateien
    - Verzeichnisse
    - Gemeinsam benutzte Dateien
    - Freispeicherverwaltung
    - Spezialfälle
- Beispiele

15.1 Motivation und Einführung

Obwohl wir nicht in jedem System Dateien benötigen, wird das Dateikonzept doch als eine der drei wichtigsten Abstraktionen der Systemarchitektur angesehen (neben Thread und Adressraum). Mittels Dateien können wir dafür sorgen, dass Information persistent abgelegt wird und jederzeit wieder abrufbar ist. Der Begriff Persistenz beinhaltet eine Art Langlebigkeit. Wie im täglichen Leben, in dem bestimmte Produkte ihren "Produzenten" überleben können, so überleben auch in einem System Dateien i.d.R. ihren Erzeugerprozess (temporäre Dateien bilden da eine Ausnahme).

15.2 Dateimanagement

Die Verwaltung der Dateien sowie die Organisation des Dateizugriffs gehören zum Betriebssystem, und werden in einer modernen Systemarchitektur auf Anwenderebene gelöst, bei den meisten heutigen Desktopsystemen jedoch nach wie vor im Systemkern, obwohl für die Mehrzahl der Operationen kein privilegierter Prozessormodus benötigt wird.

Manche Desktopsysteme wollen mittels des virtuellen Dateisystemkonzepts mehrere reale Dateisysteme unterstützen, die u.U. auch von einem Konkurrenzunternehmen angefertigt werden. Wie es sich schon beim E/A-Subsystem herausgestellt hat, birgt die Integration fremder Subsysteme in den Systemkern immer die Gefahr einer zusätzlichen Bedrohung.

15.2.1 Ziele des Dateimanagements

- Bequemes Namensschema für Dateien
    - Textuelle Namen sind vorzuziehen
    - Pfadnamen mit Konventionen bei Namenserweiterungen
- Einheitliche E/A-Unterstützung für beliebige Datenträger
- Standardisierte E/A-Schnittstelle
- Garantie, dass Dateiinhalte gültig sind
- Optimierung der Leistung
- Minimaler Datenverlust
- Zugriffskontrolle für mehrere Benutzer
- Unterstützung der Systemadministration (z.B. Backup System)

15.2.2 Interne Dateistrukturen

Dateien können wie folgt strukturiert sein:

- Byte Sequenzen
- Dateisatz (*record*) Sequenzen
- Bäume

15.3 Dateitypen

Dateitypen können zum einen wegen ihres spezifischen Inhalts oder auch wegen ihrer speziellen Rollen im Rahmen des Dateisystems insgesamt unterschieden werden, u.a. gibt es folgende Dateitypen:

Reguläre Dateien:

- ASCII Textdateien
- Archivdateien
- Binärdateien (ausführbare bzw. ladbare Dateien)

Verzeichnisse (*directories*)
Gerätedateien (*special files*):

- Zeichenorientiertes Gerät
- Blockgerät

15.4 Dateiorganisation und -zugriffsmethoden

Die richtige Dateiorganisation mit den entsprechenden Dateizugriffsmethoden ergibt sich aus den am häufigsten verwendeten Dateizugriffen pro Datei bzw. pro Dateisystem. Manche Systeme bieten die volle Palette an Dateizugriffsmethoden an, andere (wie z.B. Unix, Linux) überlassen die Implementierung der nicht-sequentiellen Dateizugriffsmethoden dem Anwender und bieten nur eine möglichst effiziente sequentielle Dateiorganisationsform an. Folgende Dateiorganisationsformen haben sich etabliert:

- Pile (unstrukturiert)
- Sequentielle Datei
- Direkte Datei oder Hashdatei
- Index-sequentielle Datei

15.4.1 Sequentielle Dateien

Sequentielle Dateien können sowohl als Bytesequenzen (wie in Unix) organisiert sein oder als Record-sequenzen (wobei man sowohl konstant lange als auch dynamisch variabel lange Records unterstützen könnte). Typische Zugriffsmethoden von sequentiellen Dateien sind:

- Read(next item, file)
- Read(file)
- Append(item, file)
- Update(item xyz, file, new data)
- Position(item pointer xyz) ...

Man kann unschwer erkennen, dass das Modifizieren eines bisherigen Records u.U. kompliziert werden kann, wenn variable Dateisatzlängen zugelassen sind, da die endgültige Position auf der Platte nicht einmal

bei zusammenhängender Abspeicherung der Records möglich ist. Mit Hilfe von Metadaten (analog Unix) kann man aber immerhin die Position eines Records schneller ermitteln, als wenn man die gesamte sequentielle Datei von Anfang an durchsuchen müsste. Dieser typische Nachteil von sequentiellen Dateien wird dann jedoch wettgemacht, wenn man an die sequentiellen typischen Zugriffe denkt, in denen eine Datei z.B. komplett gelesen wird.

15.4.2 Direkte Dateien

Bei dieser Dateiorganisation geht man davon aus, dass man auf jeden Dateisatz über einen Schlüssel möglichst schnell zugreifen möchte. Bei dieser Dateiform will man so gut wie nie einen kompletten Abzug der Datei.

Da man sich als Erzeuger der Datei am Anfang einen bestimmten Plattenbereich reservieren muss, kann es leicht vorkommen, dass man den späteren Umfang der Datei unterschätzt, d.h. ab einem gewissen Füllgrad kann man nicht mehr davon ausgehen, dass man den gewünschten Record in O(1) auf der Platte findet, da er bereits auf den Überlaufbereich abgebildet werden musste. 1980 erfand man das so genannte "Erweiterbare Hashing", das zur Laufzeit gestattet, die direkte Datei nur an den benötigten Stellen zu erweitern. Über die so genannten Basisvektoren, die als indirekte Metadaten fungieren, kann man nun -von gelegentlichen meist lokal beschränkten Reorganisationen abgesehen- jeden Datensatz in O(2) + O(Containerlänge/Anzahl gespeicherter Datensätze) erreichen.

15.4.3 Indexsequentielle Dateien

Heutzutage werden statt indexsequentieller Dateien nur noch baumsequentielle Dateien verwendet, meist so genannte B*- oder B+-Dateien. Bei einer B*-orientierten Dateiorganisation enthalten nur die Blätter Datensätze, während die internen Konten des B*-Baums nur Metadaten beinhalten.

Die Klasse der indexsequentiellen Dateien stellen ein Art Kompromiss zwischen den rein sequentiellen Dateien und den direkten Dateien dar. Wenn also von Zeit zu Zeit auch mal der gesamte Dateiinhalt sequentiell ausgegeben oder modifiziert werden muss (z.B. Gehaltsaufbesserung aller Universitätsbeschäftigten), aber häufig auch nur der Datensatz einer ganz bestimmten Person aus allen Uni-Beschäftigten benötigt wird (Geburt eines Kindes und damit höherer Lohn), dann wählt man besser doch eine indexsequentielle Dateiorganisation.

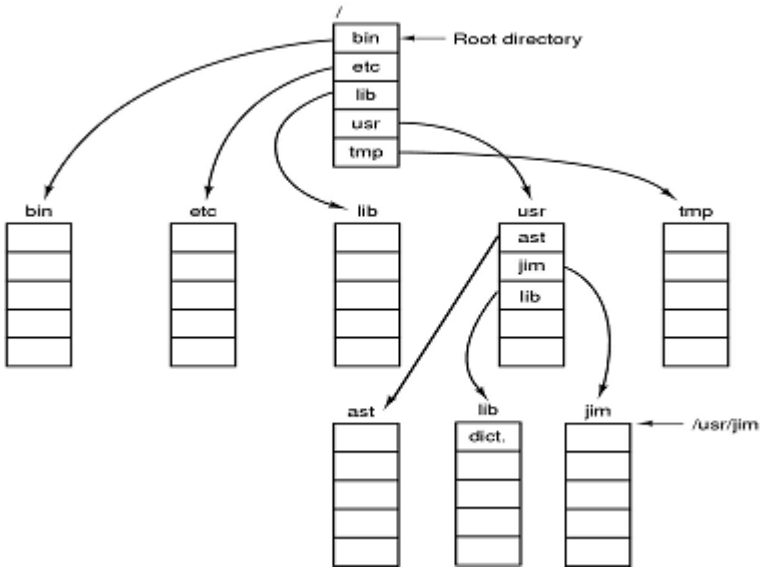
Ähnlich wie bei erweiterbaren Hashdateien bleiben bei den B*-organisierten Dateien beim Löschen oder Hinzufügen von mehreren Datensätzen die benötigten Strukturorganisationen eher lokal beschränkt, nur in Ausnahmefällen muss zumindest ein Teil des B*-Baums komplett reorganisiert werden.

15.5 Verzeichnisse

Verzeichnisse sind besondere Dateien, die dem Betriebssystem gehören und eine Abbildung von Dateinamen auf interne Dateideskriptoren (FIDs) (z.B. Anfangsadresse des Dateikopfes (inode)) beinhalten. Sie können u.a. auch noch Informationen über reguläre- und Gerätedateien besitzen, zu diesen Dateiinformationen könnten gehören u.a.

- Dateiattribute, wie z.B.
    - Textdatei, Binärdatei
    - Dateierstellungsdatum
- Anfangsadresse der Datei
- Besitzer der Datei und weitere Schutzattribute wie z.B.
    - Zugriffsrechte des Besitzers
    - Zugriffsrechte anderer Benutzer etc.

Die Organisation von Verzeichnissen kann entweder flach oder baumartig gestaltet werden. Die erste Form kann nur bedingt und nur in sehr kleinen Systemen nutzbringend verwendet werden, größere Dateisysteme erfordern, dass man über einen hierarchisch angeordneten Baum von Verzeichnissen die gewünschten Dateien möglichst effizient finden kann (s.u.).



Mit einer Reihe von zusätzlichen Metadaten (siehe Unix), kann ein Benutzer sich relativ bequem innerhalb eines größeren Dateibaumes orientieren und zu den entsprechenden Verzeichnissen navigieren. Ein solches Metadatum ist:

Working directory = aktuelles Verzeichnis

Statt sich jedes Mal die absoluten Pfadnamen der Dateien merken zu müssen, kann man Dateien auch relativ zum aktuellen Verzeichnis ansprechen.

Welche typischen Schnittstellenoperationen bieten moderne Verzeichnisse an?

- `Create_directory`
- `Delete_directory`
- `Open_directory`
- `Close_directory`
- `Read_directory`
- `Rename_directory`
- `Link_directory`
- `Unlink_directory`

In Unix können zwei Formen von zusätzlichen Links auf eine Datei unterschieden werden:

- Hardlink (ein neuer Name für die gleiche Datei)
  - Nur innerhalb eines Dateisystems anwendbar
  - Nur Superuser darf Hardlinks auf Verzeichnisse setzen
- Symbolic Link (eine neue Datei, die den absoluten Pfadnamen der Zieldatei enthält)
  - Kann auch über Dateisystemgrenzen hinweg eingesetzt werden
  - Beim Löschen der Zieldatei zeigt dieser Link ins Nirwana

15.6 Implementierung von Dateisystemen

Das Implementieren von Dateisystemen muss einige Besonderheiten berücksichtigen, zum einen sollte es möglich sein, bestimmte Subdateisysteme zur Laufzeit zu entfernen (unmount) bzw. hinzuzufügen (mount), ob diese nun auf der Festplatte oder auf einem entfernbaren Datenträger realisiert sein mögen. Wie auch immer, in jedem lokalen System wird es ein ausgezeichnetes Dateisystem, das so genannte Wurzeldateisystem (oder root file system) geben, dessen Startadresse im Betriebssystem (oder im BIOS) "hart codiert" sein muss.

Bei einem Unix ähnlichen Dateisystem wird es neben dem Bootblock, der für das Urladen des Systems verantwortlich zeichnet, den so genannten Superblock geben, der im Prinzip alle Dateisystem relevanten Organisationsdaten beinhaltet, u.a. die Standardblockgröße, die Startadresse der Inodetabelle, die Grenze zwischen Inodetabelle und den Daten- bzw. Metadatenblöcken des Dateisystems und natürlich dessen Gesamtkapazität in Anzahl von Plattenblöcken.

15.6.1 Umgang mit Dateien

Will nun ein Nutzer eine neue Datei erzeugen, dann gibt es ähnlich wie beim Laden einer neuen Task prinzipiell zwei Vorgehensweisen:

1. Man reserviert sofort den maximalen Dateibedarf an Daten und Metadatenblöcken,
2. man reserviert nur auf Bedarf Daten- und Metadatenblöcke.

Im ersten Fall besteht bei entsprechender Freispeicherverwaltung die Möglichkeit, dass man die Datei in konsekutiven Plattenblöcken implementieren kann, andererseits besteht die Gefahr, dass man u.U. viel zuviel Plattenplatz verschwendet, wenn diese Datei ihre maximale Ausdehnung niemals erreicht. Eine konsekutive Abspeicherung einer sequentiellen Datei könnte sich sehr positiv auf den sequentiellen Zugriff auf diese Datei auswirken.

15.6.2 Implementierung von sequentiellen Dateien

Prinzipiell gibt es drei verschiedene Arten, sequentielle Dateien auf der Platte abzuspeichern:

- Zusammenhängend
- verkettet
- indiziert
    - konstante Blockgröße
    - variable Blockgröße

Eine zusammenhängende Abspeicherung einer sequentiellen Datei erfordert, dass man vorab den benötigten voraussichtlich maximalen Speicherplatz beim Erzeugen der Datei reserviert. Bei dieser Implementierungsart, könnte man mit Abstrichen sogar eine Art von indiziertem Zugriff per Rechnung unterstützen.

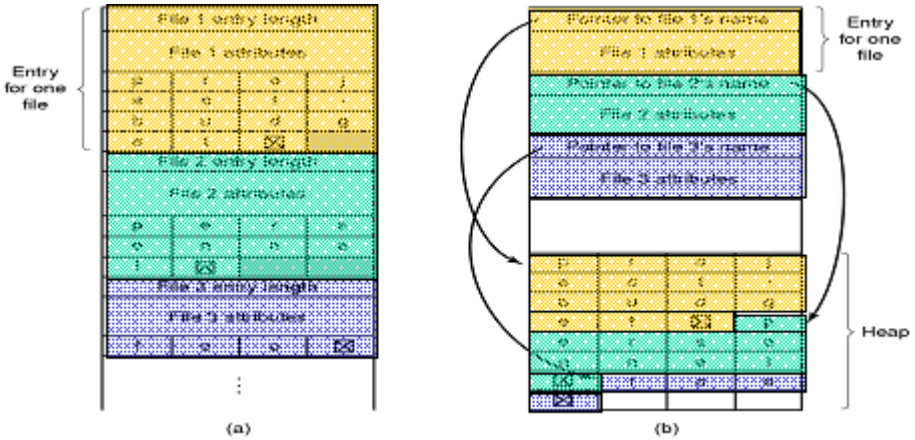
Die verkettete Lösung hingegen lässt nur den rein sequentiellen Zugriff zu, da im Verzeichnis oder im Dateikopf nur ein Verweis (Plattenadresse) auf den ersten Datenblock enthalten ist. Man kommt bei dieser Lösung mit einer sehr primitiven Freispeicherverwaltung aus, allerdings wird sich im Laufe der Zeit eine völlig zersplitterte Abbildung der sequentiellen Datenblöcke ergeben. Mittels eines Defragmentierungsalgorithmus könnte man dem wiederum entgegen wirken.

Eine indizierte Implementierung einer sequentiellen Datei entspricht der Vorgehensweise seit Unix, und ist sowohl für sehr kleine als auch für sehr große sequentielle Dateien gleichermaßen geeignet.

15.6.3 Verzeichnisse

Verzeichnisse haben die primäre Aufgabe, Dateinamen auf "Anfangsadressen von Dateien" oder so genannte *inodes* abzubilden. MS-DOS hatte die unangenehme Eigenschaft, dass die Dateinamen auf nur 8 Zeichen

beschränkt waren, Unix war da schon viel mutiger und unterstützte von Anfang an Dateinamen von bis zu 255 Zeichen. Will man auch lange Dateinamen neben den gebräuchlichen kürzeren Dateinamen unterstützen, dann bieten sich im Prinzip für die Einträge in Verzeichnissen zwei verschiedene Lösungen an:



In der ersten Lösung wird als erstes Feld eines Verzeichniseintrags dessen Länge abgespeichert, dann kommen sonstige Dateiattribute, und im Anschluss daran erst der Dateiname, der durch ein Zeichenketten-endezeichen abgeschlossen wird. In der zweiten Lösung werden konsekutiv am Anfang des Verzeichnisses nur die konstant langen Dateiattribute aufgeführt, während der Dateiname in einem Heap am Ende des Verzeichnisses über einen Zeiger erreicht werden kann.

15.6.4 Gemeinsam genutzte Dateien

Die gemeinsame Nutzung der gleichen Datei kann zum einen dadurch unterstützt werden, dass man einer Datei zwei verschieden Namensverweise (*links*) zuordnet und sie somit aus zwei unterschiedlichen Arbeitsverzeichnissen (*working directories*) aus ansprechen kann. Bei der Namensgebung unterscheidet man hierbei in Unixsystemen zwei verschiedene Verweisarten:

- Hardlinks
- Symbolische Verweise

Im ersten Fall enthalten die beiden Verweise jeweils einen Dateinamen (kann auch der gleiche sein), der aber auf die gleiche Inode verweist. In der Inode wird dabei der so genannte Verweiszähler (*link count*) hierbei auf zwei gesetzt. Dies wird deshalb gemacht, weil man erreichen will, dass erst wenn der letzte Verweis auf eine Datei gelöscht wird, dies auch das Zeichen dafür ist, die Datei selbst zu löschen. Die Verwendung von Hardlinks ist in soweit beschränkt, dass Hardlinks auf Verzeichnisse nur dem Systemadministrator erlaubt sein sollen, da dies ansonsten zu möglichen Zyklen im Dateisystem führen könnte, so dass beispielsweise das ls-Kommando nicht mehr terminieren würde.

Symbolische Verweise sind dagegen fast wie reguläre Dateien, deren Inhalt eben nur ein absoluter oder relativer Pfadname ist. Bei der Verwendung von Symbolischen Verweisen gibt es keine Einschränkungen, sie können sogar auf Dateien verweisen, die es noch nicht gibt bzw. auch nie geben wird, oder die zur Zeit auf einem entfernten Dateiträger realisiert sind.

Wollen zwei Anwenderprozesse die gleiche Datei benutzen, dann sollten sich beide per Konvention daran halten, entsprechende Lock-Protokolle einzuhalten. Man beachte, dass die so genannten READ-/WRITE-Locks genau zu diesem Zweck eingeführt worden sind.

15.6.5 Freispeicherverwaltung

Die Freispeicherverwaltung auf den Platten kann wie auch schon bei der Hauptspeicherverwaltung mittels Bitmaps oder mittels verketteter Datenstrukturen realisiert werden. Pro Dateisystem gibt es nur eine einzige

Portionsgröße, die verwaltet werden muss, d.h. bei den heute üblichen Plattensystemen können beide Freispeicherverwaltungen problemlos eingesetzt werden.

15.6.6 Spezialfälle

Unabhängig von der Zuverlässigkeit der Hardware und der verwendeten Freispeicherverwaltung und des Dateimanagements ist es unerlässlich, dass man in regelmäßigen Abständen den Inhalt des Dateisystems inkrementell oder komplett auf einen Archivspeicher abbildet, um so im Fehlerfall zumindest auf einen früheren Stand wieder aufsetzen zu können.

15.6.6.1 Logstructered Dateisystem

Bei dieser Idee geht man davon aus, dass die meisten Dateizugriffe auf die Platte Schreibzugriffe sind, denn bei genügend großen Plattenpufferspeicher (*disk cache*) sind die aktuell benötigten Lesezugriffe bereits im Plattenpufferspeicher. Statt den gewöhnlichen Dateizusammenhang abzuspeichern, wird man nun nur noch in Tagessegmenten die jeweils modifizierten Dateiinhalte auf Platte schreiben und dies in den Inodes festhalten. Hierbei wird die Platte als riesiger Ringpuffer aufgefasst.

15.6.6.2 Memory Mapped Dateien

Man kann ganze Dateien in den virtuellen Adressraum einer Task abbilden, wobei der tatsächliche Transfer der Datei in den Hauptspeicher analog wie beim Paging anderer Adressraumregionen realisiert wird, als entweder sofort (*eager*) oder verzögert (*lazy*). Im ersten Fall wird im Zuge des `mmap` Systemaufrufs die ganze Datei in den Hauptspeicher abgebildet, egal ob wir nur kleine Ausschnitte der Datei benötigen oder die ganze Datei. Wenn sowieso die ganze Datei abgebildet werden muss, weil beispielsweise jeder Dateisatz aktualisiert werden soll, dann könnte die "EAGER MAPPING-Variante" vorteilhaft sein, im Allgemeinen wird hierdurch jedoch unnötig Hauptspeicher verschwendet.

15.7 Beispieldateisysteme

Bei den Beispieldateisystemen reicht es, wenn Sie die Dateisysteme von Unix und Linux, sowie das Reiserdateisystem beherrschen.

Potentielle Fragen:

1. *Worin besteht der wesentliche Unterschied zwischen dem Standard-Unixdateisystem und dem Berkeley FFS?*
2. *Durch welche Metadaten wird der sequentielle Dateizugriff in Unix unterstützt?*
3. *Sieht man von Kapazitätsfragen der Platte ab, welche Größen bestimmen die maximale Dateigröße in Unix?*