1 – File Systems Management

List of Slides

1. File Systems Management
2. File Systems Design Goals
3. A Data Type View of File Systems
4. Operations on Files
5. Operations on Parts of Files
6. File Characteristics
7. File System Characteristics
8. Directory Structure
9. Directory Structure Information
10. Single Level Directory
11. Two Level Directory
12. Tree Structured Directory
13. DAG Structured Directory
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>General Graph Structured Directory</td>
</tr>
<tr>
<td>19</td>
<td>Disk Vs. File System Formatting</td>
</tr>
<tr>
<td>20</td>
<td>File Organization and Access</td>
</tr>
<tr>
<td>21</td>
<td>Sequential Access Files</td>
</tr>
<tr>
<td>22</td>
<td>Direct or Hashed Access Files</td>
</tr>
<tr>
<td>23</td>
<td>Indexed Files</td>
</tr>
<tr>
<td>24</td>
<td>Indexed Sequential Files</td>
</tr>
<tr>
<td>25</td>
<td>File Sharing and Semantics</td>
</tr>
<tr>
<td>26</td>
<td>Session level Consistency Semantics</td>
</tr>
<tr>
<td>27</td>
<td>File Protection Mechanisms</td>
</tr>
<tr>
<td>28</td>
<td>Some Protection Mechanisms</td>
</tr>
<tr>
<td>29</td>
<td>Unix Protection Semantics</td>
</tr>
<tr>
<td>31</td>
<td>AFS Protection Semantics</td>
</tr>
<tr>
<td>32</td>
<td>AFS Fundamental Protection Semantics</td>
</tr>
<tr>
<td>33</td>
<td>AFS Compound Protection Semantics</td>
</tr>
<tr>
<td>34</td>
<td>Unix File I/O</td>
</tr>
<tr>
<td>37</td>
<td>Modern File Systems Approaches</td>
</tr>
</tbody>
</table>
38 Meta-Data and its Consistency
39 A Meta-Data Consistency problem
40 A Meta-Data Consistency problem
41 Meta-Data Consistency Approaches
43 Persistence of Non-File Objects
44 Name Space File Systems Issues
2 – File Systems Design Goals

Persistence refers to data which has a life time beyond the duration of the process which creates it. Files are persistent collections of data. File management focuses on the following issues [4]:

1. Meeting the data-management needs of individual users
2. Ensuring validity of data
3. Optimizing performance
4. Providing I/O support
5. Minimizing data loss
6. Provide standard I/O routines (logical vs. physical accesss)
7. Provide multi-user I/O support
8. Providing consistent distributed access
3 – A Data Type View of File Systems

Data types are often described according the data management [4]:

1. Field — A basic data element of homogeneous data type (ie. a scalar or an array).

2. Record — A collection of heterogeneous data types treated as a unit. Records may be heirarchical.

3. File — A persistent sequence of records (the records may differ in type).

4. Database — A persistent collection of related data.

Note that files and databases are typically assumed to be persistent.
Typical file operations include [2]:

1. *open* — Prepare for future access
2. *close* — Prevent further access
3. *destroy* — Remove the file from the system
4. *copy* — Duplicate to a new location
5. *rename* — Change the name
6. *print* — Print/Display contents
5 – Operations on Parts of Files

Files contain data, typically, files contain data, with the current point of access being the *seek position* or *file pointer*.

Operations on data items include [2]:

1. read
2. write
3. update
4. append
5. truncate
6. insert
7. delete
8. rewind/reset
9. seek
10. tell
6 – File Characteristics

Files may be characterized by [2]:

1. Volatility — Frequency of insertions/deletions
2. Activity — The percentage records across the file
3. Size — The amount of data in the file
7 – File System Characteristics

File systems can be described by [2]:

1. Access Methods — How data is stored/accessed
2. File Management — File access, storage, sharing, security.
3. File Integrity Mechanisms

Some more advanced/implementation issues include:

1. Kernel vs. User Level Implementation
2. Distributed file systems (DFS)
3. RAID — Redundant Array of Independent Disks
4. Checkpointing/Rollback — Saving/restoring from incremental changes.
8 – Directory Structure

A directory is a file containing information about the file system, used to navigate the physical device to find the files.

A device directory describes the contents of a single device.

A file system directory may in turn sit on top of the device directory.

Mounting allows multiple file systems to reside in the same directory structure, and requires a file system directory.
9 – Directory Structure Information

A directory structure tends to maintain the following information [3]:

1. **File Name** — Also enforces naming convention limitations.

2. **File type** — Record, block, byte stream, executable?

3. **Location** — Specifies device and where on the device

4. **Protection** — Who has permission to do what to this file?

5. **Size** — Current and (sometimes) Maximum.

6. **Current Position** — read/write pos. not often in directory

7. **Usage count** — current number of processes opening the file.

8. **Time, date, owner** — Stats. on creation, modification, etc...
10 – Single Level Directory

The simplest approach uses a single level directory. This encourages name space collisions.

Figure 1: A single level directory [3]
11 – Two Level Directory

The simplest approach uses a two-tiered directory. The second tier can be used to separate by user or by application for example. Finding a file (for deletion or opening for example) is more complex. Tends to be inflexible, but fast. Examples: RSX-11M and MS-DOS 1.0.

Figure 2: A single level directory [3]
The directory tree is a hierarchical directed acyclic graph. Finding a file requires navigating down the tree from the root. This can be expensive due to tree navigation, and due to replication.

Figure 3: A tree structured directory [3]
13 – DAG Structured Directory

A DAG is a Directed Acyclic Graph. Finding a file still requires navigating down the tree from the root. This can be expensive due to tree navigation, however replication can sometimes be avoided. Reference counts may be used. Examples, the Unix file ln and rm semantics.
Figure 4: A tree structured directory [3]
14 – General Graph Structured Directory

Typically the directory structure is expressed as a rooted graph. Traversing the structure and finding a file require that cycles be detected. *Garbage collection* may be needed to detect parts of the file system which have become unlinked from the root.
Figure 5: A tree structured directory [3]
15 – Disk Vs. File System Formatting

Frequently disks are used to store filesystems. Typically there are a variety of levels of formatting employed in disk management:

1. Low Level Format — The controller checks the disk’s sectors suitability for I/O.

2. Disk partitioning

3. Operating Systems level Formatting — The disk is has an empty file system placed on it (i.e. Unix mkfs).
Some common file structures include [2, 4]:

1. Sequential
2. Direct or Hashed File
3. Indexed
4. Indexed sequential
5. Partitioned
This is also called a flat file. The data is stored in order of arrival, and is supported for devices such as tapes.

The physical storage of flat files need not be contiguous.

Stallings calls a flat file where the type of each data item may differ a pile.

Piles require a complete scan to find any data element. Storing an item is fast $O(1)$ I/O operations. They are useful for real time data acquisition and data communication and backups.

Stallings reserves the term sequential file for a file of uniform record type, where each record has a key AND the file is sorted in order of the key values of the records.
18 – Direct or Hashed Access Files

Direct files have records of uniform size, and the records can be looked up by their position in the sequence. At the lowest level, many of the remaining file schemes are implemented using this structure.

Often people will hash the addresses of the file (using hash tables) for fast keyed lookup (good performance requires knowledge of the key structure).

Unix at the logical level supports a direct access/sequential file system, where the record size is the byte.

Limitation: the position indicator imposes a limit on file size.
Indexed files are comprised of uniformly typed records, each record having a key field. The keys impose a partial ordering on the records, with the primary key of each record being unique. Indexing is frequently used in database applications.

Indexing on multiple keys may be used.

Typically these are implemented using tree of indices, i.e. B-trees [1] via direct access methods on block oriented files.
20 – Indexed Sequential Files

Indexed Sequential Files are a sort of merge of indexed sequential files and sorted sequential files. The records in an indexed sequential file are stored in sorted order, and can be traversed rapidly for file level processing, while having quick lookup for indexed access.

These are frequently implemented on top of hashed/direct access methods.

Figure 6: A $B^+$-tree [?]
21 – File Sharing and Semantics

Consistency semantics refers to the view that individual user processes in a shared file system see.

Unix specifies the following [3]:

1. Writes to an open file by a user are immediately visible to other users that simultaneously have this file open.

2. Users may share the file pointer (i.e. they read/write from a common location).
22 – Session level Consistency Semantics

A session is the duration during which a user has access to a resource (such as having a file open).

Session level consistency semantics refers to consistency being defined at the based on when the user opened the file.

The Andrew File System (AFS) and Coda have the following semantics:

1. Writes to an open file by a user are NOT immediately visible to another user that simultaneously has this file open.

2. Upon a user closing the file, all changes will be reflected in sessions starting later.
23 – File Protection Mechanisms

The file system may keep information to prevent file corruption and improper access.

1. Read
2. Write
3. Execute
4. Append
5. Delete
24 – Some Protection Mechanisms

The following are commonly used protection schemes.


2. Naming — Require users to guess the file name (poor).

3. Encryption — Scramble the contents, force the users to guess the key.

4. Passwords — Require users to guess a password.

5. Access Groups — Require users to be have membership.

Unix uses access groups.

Unix has 3 levels of groups:

1. Owner — Who created/owns the file.

2. Group — A subset of the users on the machine to which groups you belong to,

3. World — All users on the machine try the groups command

The permissions are encoded as a 3 digit octal number, the high order digit being the owner permissions, the middle digit being the group permissions, and the low order digit being the user permissions.

The digits are constructed by adding together the following permissions:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Execute (directory search)</td>
</tr>
<tr>
<td>2</td>
<td>Write/delete</td>
</tr>
<tr>
<td>4</td>
<td>Read</td>
</tr>
</tbody>
</table>

Table 1: Unix File Permissions

To see the (symbolic) access groups for a file do:

```
ls -la filename
```
AFS Protection Semantics

AFS uses access lists (acl).

Access lists give a list of groups and users who can access a file, and describe the sorts of access they have.
27 – AFS Fundamental Protection Semantics

AFS maintains access lists via the fs command, the access rights are:

<table>
<thead>
<tr>
<th>Permission</th>
<th>Letters</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup</td>
<td>l</td>
<td>Permission to examine the ACL and the directory.</td>
</tr>
<tr>
<td>read</td>
<td>r</td>
<td>View file contents</td>
</tr>
<tr>
<td>insert</td>
<td>i</td>
<td>Add files or sub-directories</td>
</tr>
<tr>
<td>write</td>
<td>w</td>
<td>Modify file contents and use &quot;chmod&quot;</td>
</tr>
<tr>
<td>delete</td>
<td>d</td>
<td>Remove file(s) in directory</td>
</tr>
<tr>
<td>lock</td>
<td>k</td>
<td>&quot;flock&quot; Permission</td>
</tr>
<tr>
<td>administer</td>
<td>a</td>
<td>Ability to change the ACL</td>
</tr>
</tbody>
</table>
AFS also supports a sort of notational shorthand for compound protection semantics.

<table>
<thead>
<tr>
<th>Permission</th>
<th>Letters</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>rl</td>
<td>read and lookup all rights</td>
</tr>
<tr>
<td>write</td>
<td>rlidwk</td>
<td>except administer every right</td>
</tr>
<tr>
<td>all</td>
<td>rlidwka</td>
<td>removes all rights</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For more details on Unix system calls do 
man 2 name.

Unix provides the following system calls for I/O:

1. int open(const char *path, int flags, mode_t
mode)
   (a) path — the file name
   (b) flags (oflags in some Unix versions) —
       select the sort of access the user wants.
   (c) mode — The protection level for file
       creation.
   (d) result — The file descriptor (integer) or a
       negative value on failure. Reserved
       descriptors are 0 (stdin), 1 (stdout), and 2
       (stderr)

2. int close(int d) — Closes descriptor d.
3. `ssize_t write(int d, const void *buf, size_t nbytes)`
   
   (a) d — the file descriptor
   (b) buf — the data to write
   (c) nbytes — how much data to write
   (d) result — On success returns the number of bytes written, on failure returns a negative value.

4. `ssize_t read(int d, const void *buf, size_t nbytes)`
   
   (a) d — the file descriptor
   (b) buf — the data to write
   (c) nbytes — how much data to write
   (d) result — On success returns the number of bytes read, on failure returns a negative value.
5. \texttt{off_t lseek(int fd, off_t offset, int whence)}

   (a) \texttt{fd} — the open file descriptor
   
   (b) \texttt{offset} — relative displacement in file
   
   (c) \texttt{whence} — Base address, \texttt{SEEK_SET} is start of file, \texttt{SEEK_CUR} is current file position, and \texttt{SEEK_END} is EOF.

   (d) \texttt{result} — Returns new relative file position.
Some file systems challenges include:

- Ensuring metadata consistency
- Persistence of non-file objects
- Where should data management operations go
- How name spaces are handled
- Managing files in a parallel environment (later in semester):
  - Managing large numbers of small files (DFS)
  - Managing a few large files (Parallel File Systems)
Recall that persistent data can fall into one of two categories:

- User defined data
- Meta Data — Needed to organize the user defined data

Note that if the meta data is lost or corrupted, the user data is no longer identifiable by the operating system.
Consider the following scenario:

- User 1 truncates file $A$ making sector $x$ available for reuse
- User 2 appends to file $B$ using sector $x$

What would happen if the file system:

1. the directory entry for files $A$ and $B$ are cached in memory
2. the *in memory* copy of $A$’s directory entry is marked so that it no longer includes $x$
3. the *in memory* copy of $B$’s directory entry is updated so that it includes $B$
4. Sector $x$ is written to disk
5. The directory entry for file $B$ is updated
6. The System crashes before any further writes occur

Can an automated file checker (e.g. fsck) fix this on restart without human intervention?
33 – A Meta-Data Consistency problem

No! This is because the block $x$ appears to belong in both file $A$ and file $B$. 
There are a number of approaches adopted:

- **Ostrich** — ext2fs - asynchronous and unordered metadata updates, but that the problem happens rarely enough to warrant higher performance.

- **Files are expendable** — tmpfs — shares virtual memory space with the /tmp directory in many Unix Systems.

- **Totally Synchronous** — Supported by FreeBSD, has a high performance penalty, but is very safe.

- **Synchronous Metadata Updates** — Supported by FreeBSD FFS, ensures at most one transaction is incomplete at crash, so recovery can occur.

- **McKusick’s Soft Updates** — Apply all metadata updates in memory first and then write FINAL inode values to disk (good for actions beating on individual inodes, e.g. `rm *`).
• Journaling — Ousterhouts LFS, SGI XFS — Write meta data updates are appended to a cyclic region on the disk, and applied during low file system activity. If the system crashes, the unapplied transactions in the log are applied. Small updates require two disk updates (slow).
35 – Persistence of Non-File Objects

Keykos and its successor the *Extremely Reliable Operating System* (EROS) [?] focus on:

- Ability for quick restart after abnormal termination
- Ensuring metadata consistency at all times

This is achieved by checkpointing every object in the system to prevent inconsistency.
36 – Name Space File Systems Issues

This area is receiving some attention (e.g. AVS by Terry Jones, and Reiser FS). Changes in this area are visible to the user, which brings additional difficulties:

- **Inertia** — Don’t break our code base please!
- **Difficulty** — If we can’t fix it, it ain’t broke.
- **Uncertainty** — How could we tell if it was fixed?
References


