File Systems

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s
Copyright notice: care has been taken to use only those web images deemed by the
instructor to be in the public domain. If you see a copyrighted image on any slide and are
the copyright owner, please contact the instructor. It will be removed.
Outline

• File system concepts
  – What is a file?
  – What operations can be performed on files?
  – What is a directory and how is it organized?

• File implementation
  – How to allocate disk space to files?
What is a file

• User view
  – Named byte array
    • Types defined by user
  – Persistent across reboots and power failures

• OS view
  – Map bytes as collection of blocks on physical storage
  – Stored on nonvolatile storage device
    • Magnetic Disks
Role of file system

• Naming
  – How to “name” files
  – Translate “name” + offset → logical block #

• Reliability
  – Must not lose file data

• Protection
  – Must mediate file access from different users

• Disk management
  – Fair, efficient use of disk space
  – Fast access to files
File metadata

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system (inode number in UNIX)
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

- How is metadata stored? (inode in UNIX)
File Access Methods

• Sequential Access
  – Maintain file pointer

• Direct Access
  – Relative block number
  – Relative block numbers: allow OS to decide where file should be placed (like paging virtual memory addresses)

• Indexed Access (e.g., ISAM)
  – File records kept sorted on a specified index-key
  – Index block tracks beginning record in each data block
UNIX File operations

• int creat(const char* pathname, mode_t mode)
• int unlink(const char* pathname)
• int rename(const char* oldpath, const char* newpath)
• int open(const char* pathname, int flags, mode_t mode)
• int read(int fd, void* buf, size_t count);
• int write(int fd, const void* buf, size_t count)
• int lseek(int fd, offset_t offset, int whence)
• int truncate(const char* pathname, offset_t len)
• ...

4/10/13  COMS W4118. Spring 2013, Columbia University. Instructor: Dr. Kaustubh Joshi, AT&T Labs.
Everything as a file

• A core UNIX tenet from the early days
  – Block devices (disks, graphics cards in /dev)
  – Character devices (USB devices, network cards in /dev)
  – IPC: Pipes, Network sockets
  – Accessing kernel data structures (/proc, /sys)
  – Setting kernel configuration
  – Volatile filesystems in RAM (e.g., tmpfs)
  – Shared memory (based on tmpfs/shmfs)
  – Remote files (NFS, SMB, AFP, …)
  – Even normal local files

• Implications
  – Everything accessed using common API (open, read, write)
  – Implementation may be totally different
  – OS must support some measure of object orientedness
Open files

- Problem: **expensive** to resolve name to identifier on each access
- Solution: open file before access
  - **Name resolution**: search directories for file name and check permission
  - Read relevant file metadata into **open file table** in memory
  - Return index in open file table (**file descriptor**)
  - Application pass index to OS for subsequent access

- **System-wide open file table** shared across processes
- **Per-process open file table** stores current pointer position and index to system-wide open file table
Directories

• Organization technique
  – Map file name to location on disk
  – Also stored on disk

• Single-Level directory
  – Single directory for entire disk
    • Each file must have unique name
  – Not very usable

• Two-level directory
  – Directory for each user
  – Still not very usable
Tree-structured directory

• Directory stored on disk just like files
  – Data consists of <name, index> pairs
    • Index points to file identifier (inode)
    • Name can be another directory
  – Designated by special bit in meta-data
  – Reference by separating names with slashes
  – Operations
    • User programs can read (readdir())
    • Only special system calls can write

• Special directories
  – Root (/): fixed index for metadata
  – . : this directory
  – .. : parent directory
Acyclic-graph directories

- Directories can share files
- Create links from one file
- Two types of links
  - **Symbolic link**
    - Special file, designated by bit in meta-data
    - File data is name to another file
  - **Hard link**
    - Multiple directory entries point to same file
    - All hard-links are equal: no primary
    - Store reference count in file metadata
    - Cannot refer to directories; why?
Cycles cause problems with reference counts.
E.g., a cycle that isn’t accessible through root.
Need garbage collection.
Path names

• Absolute path name (full path name)
  – Start at root directory
    • E.g. /home/html

• Relative path name
  – Full path is lengthy and inflexible
  – Give each process current working directory
  – Assume file in current directory
Directories as files

• Direction as special files that store pointers to the contained files
  – File data is interpreted by FS code

• Separate functionality in two levels
  – Lowest: storage management
  – Highest: naming, directory

• Advantage: simplifies design and implementation
In-Memory File System Structures

Principle: heavy caching to reduce impact of slow disk I/O
Unified Buffer Cache

- We’ve seen the Linux page cache
  - Example of unified memory-disk subsystems

Separate I/O and paging systems (double caching)

Unified disk and paging cache
Example: Linux In-memory Data Structures

• struct super_block
  – Contains FS type, size, free space, pointer to root dir

• struct inode
  – One per physical file
  – Unique inode number
  – Contains file size, permissions, attributes, timestamps

• struct dentry
  – A directory entry (to file or another directory)
  – Contains name used to access file, inode number

• struct file
  – File opened by process
  – Contains file pointer, mode user opened the file in
File System Mounting

- Start off with root filesystem
- New file systems can be **mounted** into an existing directory (**mount point**)
- E.g., `mount -o opts -t ext2 /dev/hda3 /users`

(a) (b)
Protection

• Type of access
  – Read, write, execute, append, delete, list ...

• Access control list
  – Associate lists of users with access rights for every file
  – Advantage: complete control
  – Disadvantage
    • Tedious to construct list (may not know in advance for all users)
    • Require variable-size information

• Classify users
  – Assign an owner and group to each file
  – Different permissions based on who is accessing: owner, group, other
  – Advantage: easier to implement
  – Disadvantage: no fine grained control
Outline

• File system concepts
  – What is a file?
  – What operations can be performed on files?
  – What is a directory and how is it organized?

• File implementation
  – How to allocate disk space to files?
Typical file access patterns

• **Sequential Access**
  - Data read or written in order
    - Most common access pattern
      - E.g., copy files, compiler read and write files,
    - Can be made very fast (peak transfer rate from disk)

• **Random Access**
  - Randomly address any block
    - E.g., update records in a database file
  - Difficult to make fast (*seek time and rotational delay*)
Disk management

• Need to track where file data is on disk
  – How should we map logical sector # to surface #, track #, and sector #?
    • Order disk sectors to minimize seek time for sequential access

• Need to track where file metadata is on disk

• Need to track free versus allocated areas of disk
  – E.g., block allocation bitmap (Unix)
    • Array of bits, one per block
    • Usually keep entire bitmap in memory
Allocation strategies

• Various approaches (similar to memory allocation)
  – Contiguous
  – Extent-based
  – Linked
  – FAT tables
  – Indexed
  – Multi-Level Indexed

• Key metrics
  – Fragmentation (internal & external)?
  – Grow file over time after initial creation?
  – Fast to find data for sequential and random access?
  – Easy to implement?
  – Storage overhead?
Contiguous allocation

• Allocate files like **continuous memory allocation** (base & limit)
  – User specifies length, file system allocates space all at once
  – Can find disk space by examining bitmap
  – Metadata: contains starting location and size of file
Contiguous allocation example

```
count
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
```

```
directory

<table>
<thead>
<tr>
<th>file</th>
<th>start</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>tr</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>mail</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td>list</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Pros and cons

• Pros
  – Easy to implement
  – Low storage overhead (two variables to specify disk area for file)
  – Fast sequential access since data stored in continuous blocks
  – Fast to compute data location for random addresses. Just an array index

• Cons
  – Large external fragmentation
  – Difficult to grow file
Extent-based allocation

- Multiple contiguous regions per file (like segmentation)
  - Each region is an extent
  - Metadata: contains small array of entries designating extents
    - Each entry: start and size of extent
Pros and cons

• Pros
  – Easy to implement
  – Low storage overhead (a few entries to specify file blocks)
  – File can grow overtime (until run out of extents)
  – Fast sequential access
  – Simple to calculate random addresses

• Cons
  – Help with external fragmentation, but still a problem
Linked allocation

- All blocks (fixed-size) of a file on linked list
  - Each block has a pointer to next
  - Metadata: pointer to the first block
Linked allocation example
Pros and cons

• Pros
  – No external fragmentation
  – Files can be easily grown with no limit
  – Also easy to implement, though awkward to spare space for disk pointer per block

• Cons
  – Large storage overhead (one pointer per block)
  – Potentially slow sequential access
  – Difficult to compute random addresses
Variation: FAT table

- Store linked-list pointers outside block in File-Allocation Table
  - One entry for each block
  - Linked-list of entries for each file

- Used in MSDOS and Windows operating systems
FAT example

directory entry

```
test  •••  217
```

name  start block

no. of disk blocks  -1

FAT

0  217  618

339

618  339
Pros and cons

• Pros
  – Fast random access. Only search cached FAT

• Cons
  – Large storage overhead for FAT table
  – Potentially slow sequential access
Indexed allocation

- File has array of pointers (index) to block
  - Allocate block pointers contiguously in metadata
    - Must set max length when file created
    - Allocate pointers at creation, allocate blocks on demand
    - Cons: fixed size, same overhead as linked allocation
  - Maintain multiple lists of block pointers
    - Last entry points to next block of pointers
    - Cons: may need to access a large number of pointer blocks
Indexed allocation example

```
0  1  2  3
4  5  6  7
8  9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
```

Directory

```
file
  jeep
index block
19
```

```
  9
 16
  1
 10
 25
  1
  1
  1
```
Pros and cons

• Pros
  – Easy to implement
  – No external fragmentation
  – Files can be easily grown with the limit of the array size
  – Fast random access. Use index

• Cons
  – Large storage overhead for the index
  – Sequential access may be slow.
    • Must allocate contiguous block for fast access
Multi-level indexed files

- Block index has multiple levels
Multi-level indexed allocation (UNIX FFS, and Linux ext2/ext3)

- Inode
  - Direct blocks
  - Indirect Blocks
    - (BLKSIZE/4)
    - Double Indirect
      - (BLKSIZE/4)^2
      - Triple Indirect
        - (BLKSIZE/4)^3
  - Data Block
    - BLKSIZE/4
    - Data Block
      - Indirect Blocks
        - Indirect Blocks
          - Indirect Blocks
            - Data Block

Pros and cons

• Pros
  – No external fragmentation
  – Files can be easily grown with much larger limit compared to one-level index
  – Fast random access. Use index

• Cons
  – Large space overhead (index)
  – Sequential access may be slow.
    • Must allocate contiguous block for fast access
  – Implementation can be complex
Advanced Data Structures

- Combine Indexes with extents/multiple cluster sizes
- More sophisticated data structures
- B+ Trees
  - Used by many high perf filesystems for directories and/or data
  - E.g., XFS, ReiserFS, ext4, MSFT NTFS and ReFS, IBM JFS, btrfs
  - Can support very large files (including sparse files)
  - Can give very good performance (minimize disk seeks to find block)
Free Space Management

- File system maintains **free-space list** to track available blocks/clusters
- **Free bitmap** stored in the superblock
  
  \[
  \begin{array}{cccc}
  0 & 1 & 2 & \cdots & n-1 \\
  \end{array}
  \]

  
  \[
  \text{bit}[i] = \begin{cases} 
  1 & \text{block}[i] \text{ free} \\
  0 & \text{block}[i] \text{ occupied} 
  \end{cases}
  \]

- Linked free list in free blocks
  - **Pros**: space efficient
  - **Cons**: requires many disk reads to find free cluster of right size

- **Grouping**
  - Use a free index-block containing n-1 pointers to free blocks and a pointer to the next free index-block

- **Counting**
  - Free list of variable sized contiguous clusters instead of blocks
  - Reduces number of free list entries