[09] STORAGE

OUTLINE

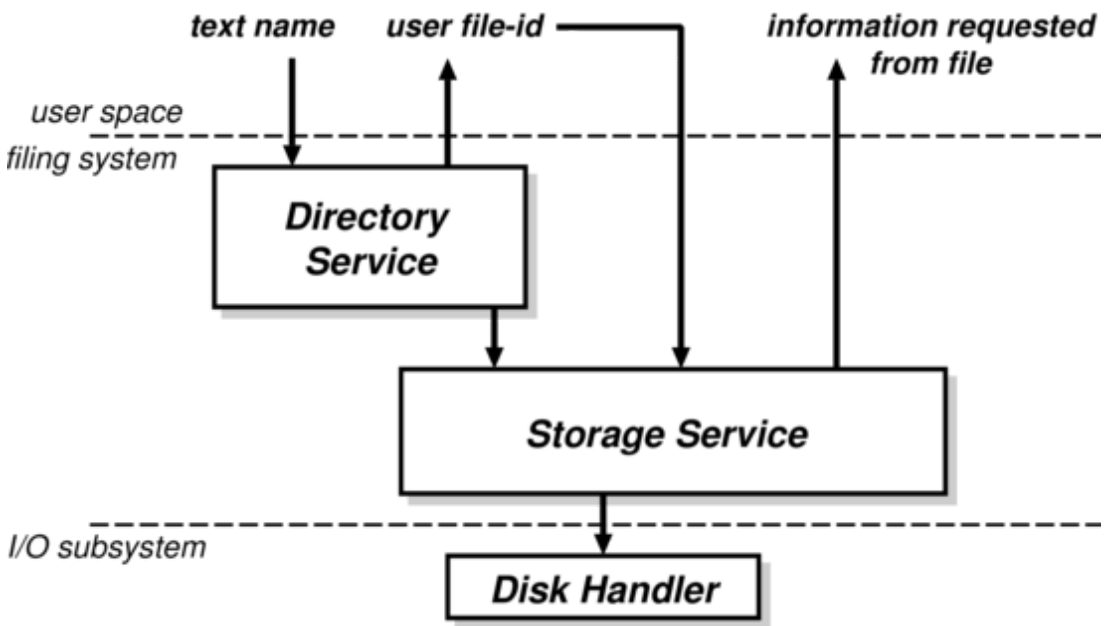
- File Concepts
  - Filesystems
  - Naming Files
  - File Metadata
- Directories
  - Name Space Requirements
  - Structure
  - Implementation
- Files
  - Operations
  - Implementation
  - Access Control, Existence Control, Concurrency Control

FILE CONCEPTS

- **File Concepts**
  - **Filesystems**
  - **Naming Files**
  - **File Metadata**
- Directories
- Files

FILESYSTEM

We will look only at very simple filesystems here, having two main components:



1. **Directory Service**, mapping names to file identifiers, and handling access and existence control
2. **Storage Service**, providing mechanism to store data on disk, and including means to implement directory service

WHAT IS A FILE?

The basic abstraction for non-volatile storage:

- User abstraction — compare/contrast with segments for memory
- Many different types:
  - Data: numeric, character, binary
  - Program: source, object, executable
  - "Documents"
- Typically comprises a single contiguous logical address space

Can have varied internal structure:

- None: a simple sequence of words or bytes
- Simple record structures: lines, fixed length, variable length
- Complex internal structure: formatted document, relocatable object file

WHAT IS A FILE?

OS split between *text* and *binary* is quite common where text files are treated as

- A sequence of lines each terminated by a special character, and
- With an explicit EOF character (often)

Can map everything to a byte sequence by inserting appropriate control characters, and interpretation in code. Question is, who decides:

- OS: may be easier for programmer but will lack flexibility
- Programmer: has to do more work but can evolve and develop format

NAMING FILES

Files usually have at least two kinds of "name":

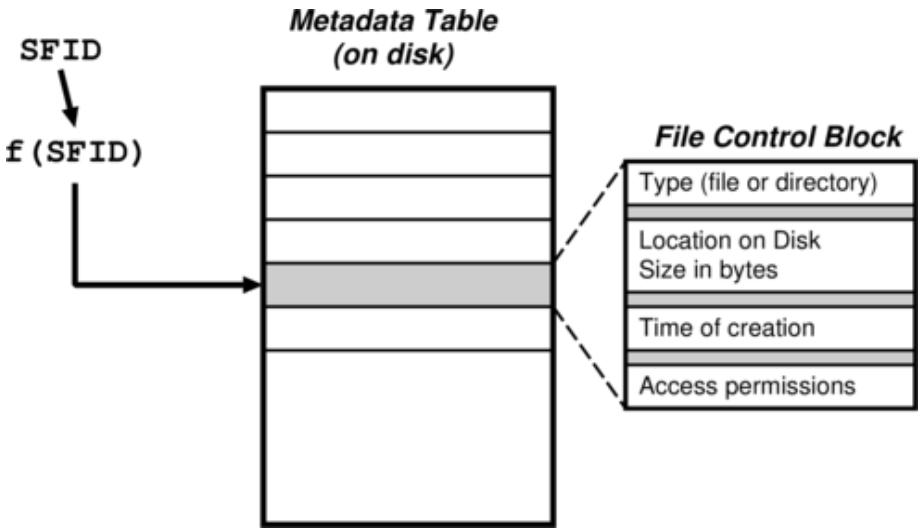
- **System file identifier** (SFID): (typically) a unique integer value associated with a given file, used within the filesystem itself
- **Human name**, e.g. `hello.java`: what users like to use
- May have a third, **User File Identifier** (UFID) used to identify open files in applications

Mapping from human name to SFID is held in a directory, e.g.,

Name	SFID
hello.java	12353
Makefile	23812
README	9742

Note that directories are *also* non-volatile so they must be stored on disk along with files — which explains why the storage system sits "below" the directory service

FILE METADATA



NB. Having resolved the name to an SFID, the actual mapping from SFID to **File Control Block** (FCB) is OS and filesystem specific

In addition to their contents and their name(s), files typically have a number of other attributes or **metadata**, e.g.,

- **Location**: pointer to file location on device
- **Size**: current file size
- **Type**: needed if system supports different types
- **Protection**: controls who can read, write, etc.
- **Time**, **date**, and **user identification**: data for protection, security and usage monitoring

DIRECTORIES

- File Concepts
- **Directories**
  - **Name Space Requirements**
  - **Structure**
  - **Implementation**
- Files

REQUIREMENTS

A **directory** provides the means to translate a (user) name to the location of the file on-disk. What are the requirements?

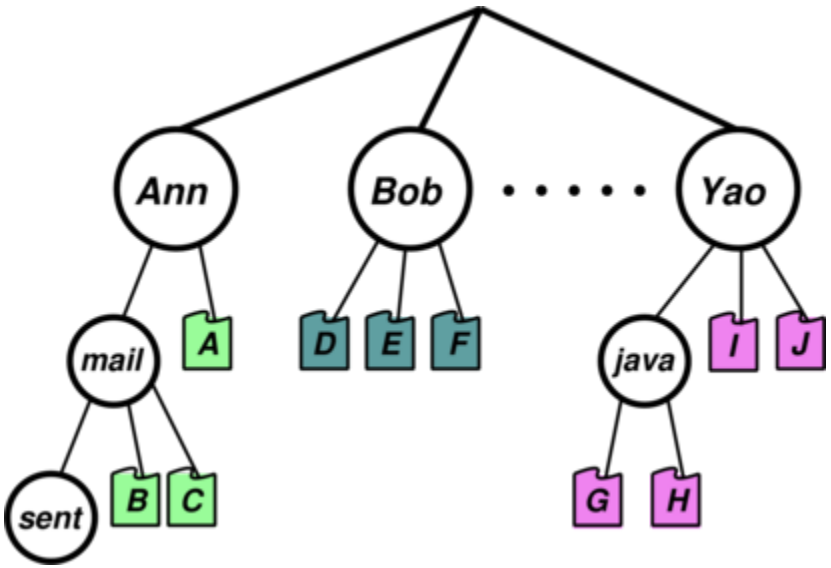
- **Efficiency**: locating a file quickly.
- **Naming**: user convenience
  - allow two (or, more generally, *M*) users to have the same name for different files
  - allow one file have several different names
- **Grouping**: logical grouping of files by properties, e.g., "all Java programs", "all games"

EARLY ATTEMPTS

- **Single-level**: one directory shared between all users
  - naming problem
  - grouping problem
- **Two-level directory**: one directory per user
  - access via pathname (e.g., `bob:hello.java`)
  - can have same filename for different user
  - ... but still no grouping capability.

Add a general hierarchy for more flexibility

STRUCTURE: TREE



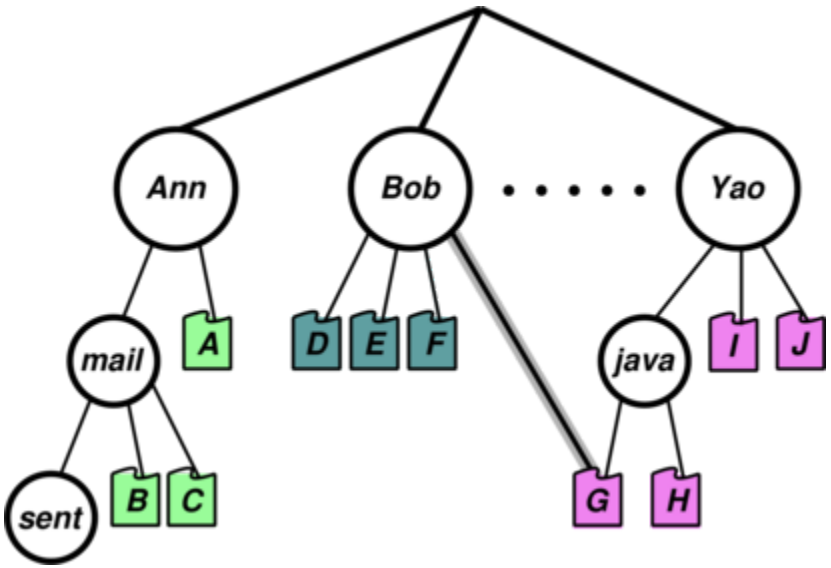
Directories hold files or [further] directories, reflecting structure of organisation, users' files, etc

Create/delete files relative to a given directory

Efficient searching and arbitrary grouping capability

The human name is then the full path name, though these can get unwiedly, e.g., `/usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c`. Resolve with **relative naming**, **login directory**, **current working directory**. Sub-directory deletion either by requiring directory empty, or by recursively deleting

STRUCTURE: DAG

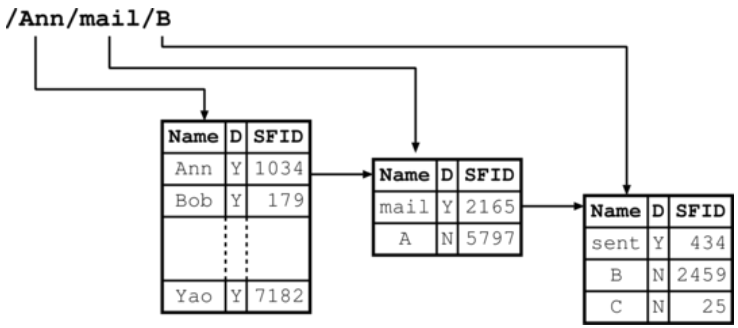


Hierarchy useful but only allows one name per file. Extend to **directed acyclic graph** (DAG) structure: allow shared subdirectories and files, and multiple aliases for same thing

Manage dangling references: use back-references or reference counts

Other issues include: **deletion** (more generally, permissions) and knowing when ok to free disk blocks; **accounting** and who gets "charged" for disk usage; and **cycles**, and how we prevent them

DIRECTORY IMPLEMENTATION



Directories are non-volatile so store as "files" on disk, each with own SFID

- Must be different types of file, for traversal
- Operations must also be explicit as info in directory used for access control, or could (eg) create cycles
- Explicit directory operations include:
  - Create/delete directory
  - List contents
  - Select current working directory
  - Insert an entry for a file (a "link")

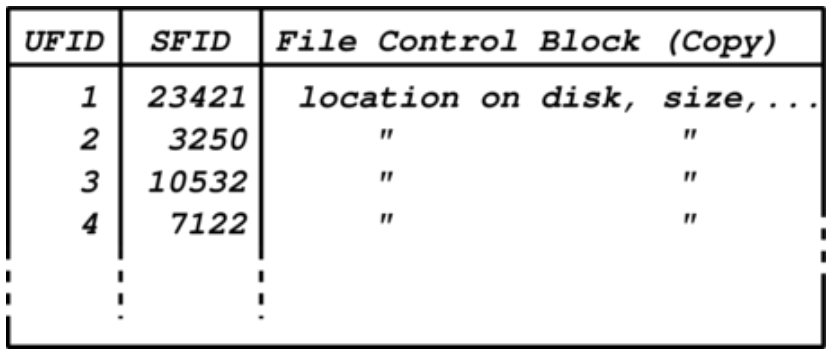
FILES

- File Concepts
- Directories
- **Files**
  - **Operations**
  - **Implementation**
  - **Access Control, Existence Control, Concurrency Control**

OPERATIONS

Basic paradigm of use is: open, use, close

Opening or creating a file:
`UFID = open(<pathname>)` or
`UFID = create(<pathname>)`



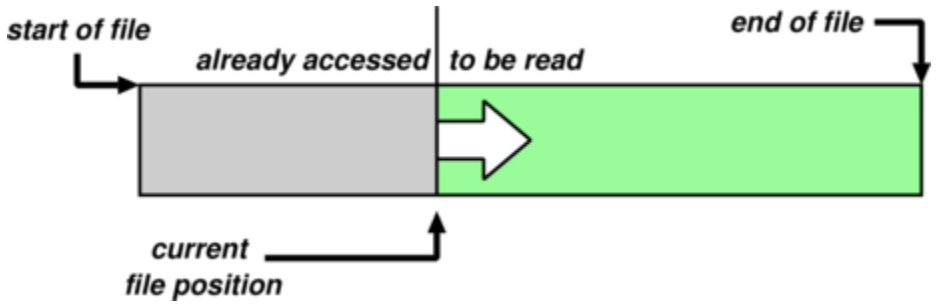
UFID	SFID	File Control Block (Copy)
1	23421	location on disk, size,...
2	3250	" "
3	10532	" "
4	7122	" "
⋮	⋮	

- Directory service recursively searching directories for components of `<pathname>`
- Eventually get `SFID` for file, from which `UFID` created and returned
- Various modes can be used

Closing a file: `status = close(UFID)`

- Copy [new] file control block back to disk and invalidate `UFID`

IMPLEMENTATION



Associate a cursor or file position with each open file (viz. UFID), initialised to start of file

- Basic operations: read next or write next, e.g., `read(UFID, buf, nbytes)`, or `read(UFID, buf, nrecords)`

Access pattern:

- **Sequential**: adds `rewind(UFID)` to above
- **Direct Access**: `read(N)` or `write(N)` using `seek(UFID, pos)`
- Maybe others, e.g., append-only, indexed sequential access mode (ISAM)

ACCESS CONTROL

File owner/creator should be able to control what can be done, by whom

- File usually only accessible if user has both directory and file access rights
- Former to do with lookup process — can't look it up, can't open it
- Assuming a DAG structure, do we use the presented or the absolute path

Access control normally a function of directory service so checks done at file open time

- E.g., read, write, execute, (append?), delete, list, rename
- More advanced schemes possible (see later)

EXISTENCE CONTROL

What if a user deletes a file?

- Probably want to keep file in existence while there is a valid pathname referencing it
- Plus check entire FS periodically for garbage
- Existence control can also be a factor when a file is renamed/moved.

CONCURRENCY CONTROL

Need some form of locking to handle simultaneous access

- Can be mandatory or advisory
- Locks may be shared or exclusive
- Granularity may be file or subset

SUMMARY

- File Concepts
  - Filesystems
  - Naming Files
  - File Metadata
- Directories
  - Name Space Requirements
  - Structure
  - Implementation
- Files
  - Operations
  - Implementation
  - Access Control, Existence Control, Concurrency Control