Operating Systems Principles

File Systems: Semantics & Structure

Mark Kampe
(markk@cs.ucla.edu)

File Systems: Semantics & Structure

Sequential Byte Stream Access

```
int infd = open("abc", O_RDONLY);
int outfd = open("xyz", O_WRONLY+O_CREATE, 0666);
if (infd >= 0 && outfd >= 0) {
    int count = read(infd, buf, sizeof buf);
    while( count > 0 ) {
        write(outfd, buf, count);
        count = read(infd, inbuf, BUFSIZE);
    }
    close(infd);
    close(outfd);
}
```


Random Access

```
void *readSection(int fd, struct hdr *index, int section) {
    struct hdr *head = &hdr[section];
    off_t offset = head->section_offset;
    size_t len = head->section_length;
    void *buf = malloc(len);
    if (buf != NULL) {
        lseek(fd, offset, SEEK_SET);
        if (read(fd, buf, len) <= 0) {
            free(buf);
            buf = NULL;
        }
    }
    return(buf);
}
```


Consistency Model

- When do new readers see results of a write?
  - read-after-write
    - as soon as possible, data-base semantics
    - this commonly called "POSIX consistency"
  - read-after-close (or sync/commit)
    - only after writes are committed to storage
  - open-after-close (or sync/commit)
    - each open sees a consistent snapshot
  - explicitly versioned files
    - each open sees a named, consistent snapshot

File Attributes – basic properties

- thus far we have focused on a simple model
  - a file is a "named collection of data blocks"
- in most OS files have more state than this
  - file type (regular file, directory, device, IPC port, ...)
  - file length (may be excess space at end of last block))
  - ownership and protection information
  - system attributes (e.g. hidden, archive)
  - creation time, modification time, last accessed time
- typically stored in file descriptor structure

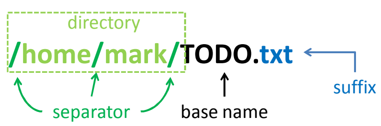
Extended File Types and Attributes

- extended protection information
  - e.g. access control lists
- resource forks
  - e.g. configuration data, fonts, related objects
- application defined types
  - e.g. load modules, HTML, e-mail, MPEG, ...
- application defined properties
  - e.g. compression scheme, encryption algorithm, ...

File Names and Name Binding

- file system knows files by their descriptors
- users know files by names
  - names more easily remembered than disk addresses
  - names can be structured to organize millions of files
- file system responsible for name-to-file mapping
  - associating names with new files
  - changing names associated with existing files
  - allowing users to search the name space
- there are many ways to structure a name space

What is in a Name?



- suffixes and file types
  - file-to-application binding often based on suffix
    - defined by system configuration registry
    - configured per user, or per directory
  - suffix may define the file type (e.g. Windows)
  - suffix may only be a hint (magic # defines type)

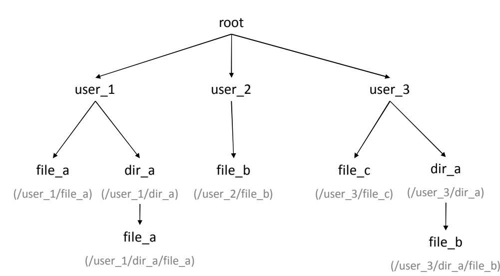
Flat Name Spaces

- there is one naming context per file system
  - all file names must be unique within that context
- all files have exactly one true name
  - these names are probably very long
- file names may have some structure
  - e.g. CAC101.CS111.SECTION1.SLIDES.LECTURE_13
  - this structure may be used to optimize searches
  - the structure is very useful to users
  - the structure has no meaning to the file system

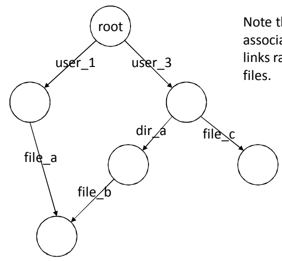
Hierarchical Namespaces

- directory
  - a file containing references to other files
  - it can be used as a naming context
    - each process has a *current working directory*
    - names are interpreted relative to directory
- nested directories can form a tree
  - file name is a path through that tree
  - directory tree expands from a *root* node
    - *fully qualified* names begin from the root
  - may actually form a directed graph

A rooted directory tree



Hard Links: example



Note that we now associate names with links rather than with files.

/user_1/file_a and

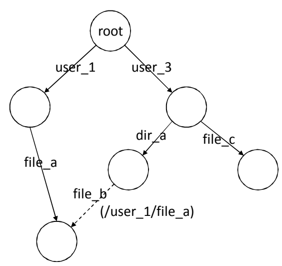
/user_3/dir_a/file_b

are both links to the same I-node

Unix-style Hard Links

- all protection information is stored in the file
  - file owner sets file protection (e.g. read-only)
  - all links provide the same access to the file
  - anyone with read access to file can create new link
  - but directories are protected files too
    - not everyone has read or search access to every directory
- all links are equal
  - there is nothing special about the owner‘s link
  - file is not deleted until no links remain to file
  - reference count keeps track of references

Symbolic Links: example



Symbolic Links

- another type of special file
  - an indirect reference to some other file
  - contents is a path name to another file
- Operating System recognizes symbolic links
  - automatically opens associated file instead
  - if file is inaccessible or non-existent, the open fails
- symbolic link is not a reference to the I-node
  - symbolic links will not prevent deletion
  - do not guarantee ability to follow the specified path
  - Internet URLs are similar to symbolic links

Databases

- a tool managing business critical data
- table is equivalent of a file system
- data organized in rows and columns
  - row indexed by unique key
  - columns are named fields within each row
- support a rich set of operations
  - multi-object, read/modify/write transactions
  - SQL searches return consistent snapshots
  - insert/delete row/column operations

Object Stores

- simplified file systems, cloud storage
  - optimized for large but infrequent transfers
- *bucket* is equivalent of a file system
  - a *bucket* contains named, versioned *objects*
- *objects* have long names in a flat name space
  - *object* names are unique within a *bucket*
- an *object* is a blob of *immutable* bytes
  - get … all or part of the *object*
  - put … new version, there is no *append/update*
  - delete

Key-Value Stores

- smaller and faster than an SQL database
  - optimized for frequent small transfers
- *table* is equivalent of a file system
  - a *table* is a collection of *key/value* pairs
- *keys* have long names in a flat name space
  - *key* names are unique within a *table*
- value is a (typically 64-64MB) string
  - get/put (entire value)
  - delete

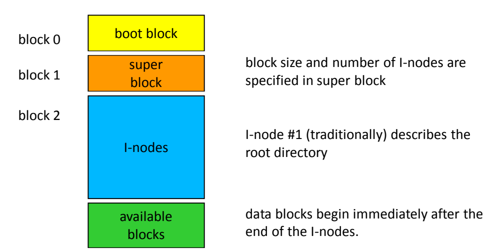
File System Goals

- ensure the privacy and integrity of all files
- efficiently implement name-to-file binding
  - find file associated with this name
  - list the file names in this part of the name space
- efficiently manage data associated w/each file
  - return data at offset X in file Y
  - write data Z at offset X in file Y
- manage attributes associated w/each file
  - what is the length of file Y
  - change owner/protection of file Y to be X

File System Structure

- disk volumes are divided into fixed-sized blocks
  - many sizes are used: 512, 1024, 2048, 4096, 8192 ...
- most of them will store user data
- some will store organizing "meta-data"
  - description of the file system (e.g. layout and state)
  - file control blocks to describe individual files
  - lists of free blocks (not yet allocated to any file)
- all operating systems have such data structures
  - different OS and FS often have very different goals
  - these result in very different implementations

Unix System 5 – Volume Structure



File Descriptor Structures

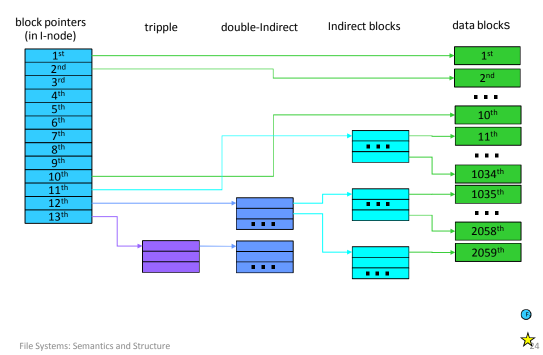
- all file systems have file descriptor structures
- contain all info about file
  - type (e.g. file, directory, pipe)
  - ownership and protection
  - size (in bytes)
  - other attributes
  - location of data blocks

UNIX I-node

type	protection
owner	group
# links	
file size	
last access time	
last written time	
last I-node update time	
data block pointers ...	

- descriptor location/# is file's *true name*

Unix I-nodes and block pointers



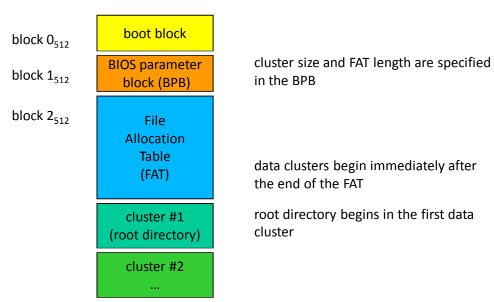
(Unix I-node block mapping)

- I-node contains 13 block pointers
  - first 10 point to first 10 blocks of file
  - 11th points to an indirect block (e.g. 4k bytes = 1k blocks)
  - 12th points to a double indirect block (w/1k indirect blocks)
  - 13th points to a triple indirect block (w/1k double indirs)
- assuming 4k bytes per block and 4-bytes per pointer
  - 10 direct blocks = 10 * 4K bytes = 40K bytes
  - indirect block = 1K * 4K = 4M bytes
  - double indirect = 1K * 4M = 4G bytes
  - triple indirect = 1K * 4G = 4T bytes (finite, but large)

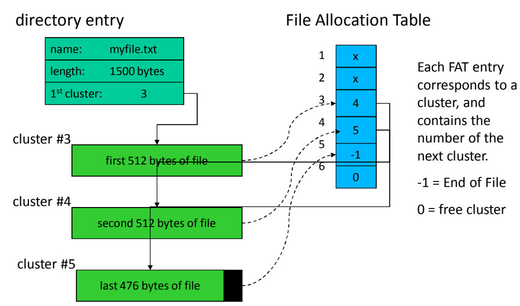
I-nodes – performance

- I-node is in memory whenever file is open
- first ten blocks can be found with no I/O
- after that, we must read indirect blocks
  - the real pointers are in the indirect blocks
  - sequential file processing will keep referencing it
  - block I/O will keep it in the buffer cache
- 1-3 extra I/O operations per thousand pages
  - any block can be found with 3 or fewer reads
- index blocks can support "sparse" files
- block # width determines max file system size

DOS FAT – Volume Structure



Clusters in a DOS FAT File



(DOS FAT File Systems – Overview)

- DOS file systems divide space into "clusters"
  - cluster size (multiple of 512) fixed for each file system
  - clusters are numbered 1 though N
- File control structure points to first cluster of file
- File Allocation Table (FAT), one entry per cluster
  - has number of next cluster in file
  - 0 -> cluster is not allocated
  - -1 -> end of file

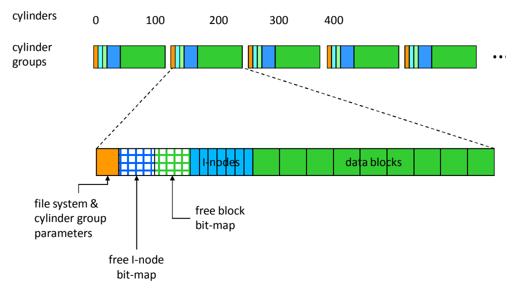
FAT – Performance/Capabilities

- to find a particular block of a file
  - get number of first cluster from directory entry
  - follow chain of pointers through FAT
- entire File Allocation Table is kept in memory
  - no disk I/O is required to find a cluster
  - for very large files the search can still be long
- no support for "sparse" files
  - if a file has a block n, it must have all blocks < n
- width of FAT determines max file system size

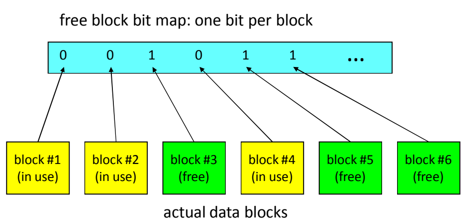
Free Space Maintenance

- file system manager manages the free space
- get/release chunk should be fast operations
  - they are extremely frequent
  - we'd like to avoid doing I/O as much as possible
- unlike memory, it matters what chunk we choose
  - best to allocate new space in same cylinder as file
  - user may ask for contiguous storage
- free-list organization must address both concerns
  - speed of allocation and de-allocation
  - ability to allocate contiguous or near-by space

FFS Cylinder Groups & Free Space



Bit Map Free Lists

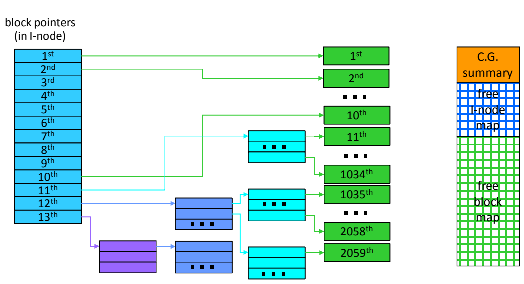


BSD file systems use bit-maps to keep track of both free blocks and free I-nodes in each cylinder group

(BSD UNIX free space bit-maps)

- file system divided into cylinder groups
  - each cylinder group has its own cyl group summary
  - active cyl group summaries are kept in memory
  - each cylinder group has its own i-nodes and blocks
  - free block list is a bit-map in cyl group summary
- enables significant reductions in head motion
  - data blocks in file can be allocated in same cylinder
  - I-node and its data blocks in same cylinder group
  - directories and their files in same cylinder group

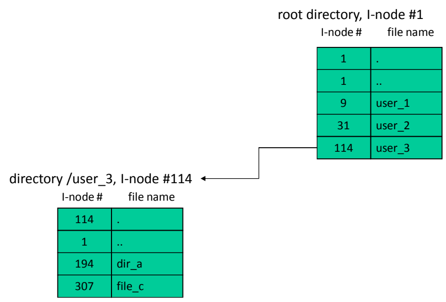
Unix File Extension



(Extending a BSD/UNIX file)

- note the cylinder group for the file's i-node
- note the cylinder for the previous block in the file
- find a free block in the desired cylinder
  - search the free-block bit-map for free block in right cyl
  - update bit-map to show the block has been allocated
- update the I-node to point to the new block
  - go to appropriate block pointer in I-node/indirect block
  - if new indirect block is needed, allocate/assign it first
  - update I-node/indirect to point to new block

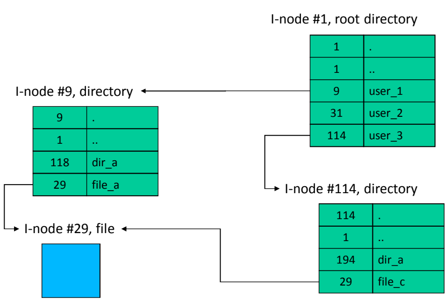
UNIX Directories



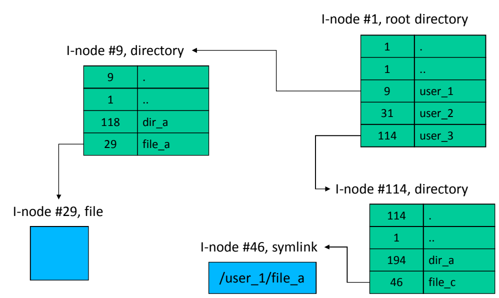
(Example: UNIX Directories)

- file names separated by slashes
  - e.g. /user_3/dir_a/file_b
- the actual file descriptors are the I-nodes
  - directory entries only point to I-nodes
  - association of a name with an I-node is called a "link"
  - multiple directory entries can point to the same I-node
- contents of a Unix directory entry
  - name (relative to this directory)
  - pointer to the I-node of the associated file

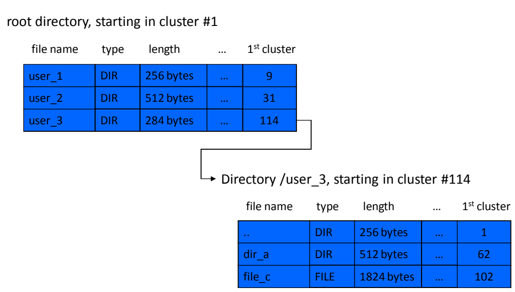
Hard Links - example



Symbolic Links - example



DOS Directories



(Example: DOS Directories)

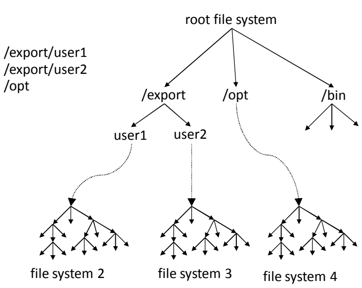
- File & directory names separated by back-slashes
  - e.g. \user_3\dir_a\file_b
- directory entries are the file descriptors
  - as such, only one entry can refer to a particular file
- contents of a DOS directory entry
  - name (relative to this directory)
  - type (ordinary file, directory, ...)
  - location of first cluster of file
  - length of file in bytes
  - other privacy and protection attributes

Unix File System Mounts

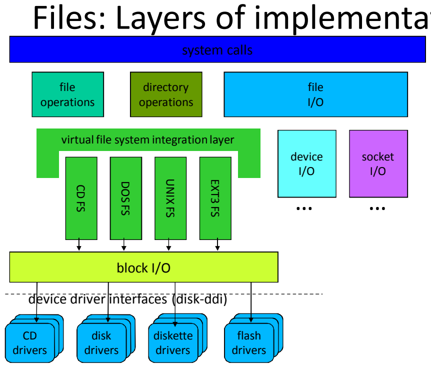
- goal
  - make many file systems appear to be one giant
  - users need not be aware of file system boundaries
- mechanism
  - mount <u>device</u> on <u>directory</u>
  - creates a warp from the named <u>directory</u> to the top of the file system on the specified <u>device</u>
  - any file name beneath that directory is interpreted relative to the root of the mounted file system

Unix - Mounted File Systems

mount filesystem2 on /export/user1
mount filesystem3 on /export/user2
mount filesystem4 on /opt



Files: Layers of implementation



Virtual File System (integration) Layer

- federation layer to generalize file systems
  - permits rest of OS to treat all file systems as the same
  - support dynamic addition of new file systems
- plug-in interface or file system implementations
  - DOS FAT, Unix, EXT3, ISO 9660, network, etc.
  - each file system implemented by a plug-in module
  - all implement same basic methods
    - create, delete, open, close, link, unlink,
    - get/put block, get/set attributes, read directory, etc
- implementation is hidden from higher level clients
  - all clients see are the standard methods and properties

Assignments

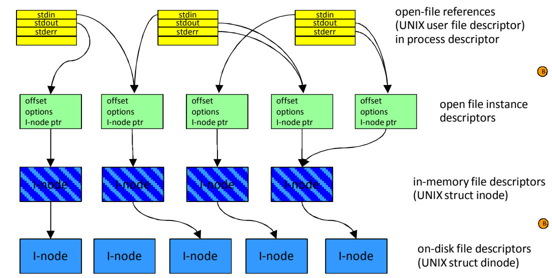
- for the next lecture:
  - Arpaci ch 41 … Fast File System (FFS)
  - Arpaci ch 42 … FSCK and Journaling
  - Arpaci ch 43 … Log Structured File Systems
  - Arpaci ch 44.1-4 … Data Integrity and Protection
  - Arpaci appx I6-10 … Flash based SSDs
  - Arpaci ch 45 … Summary

Supplementary Slides

Compaction and Defragmentation

- file I/O is efficient if file extents are contiguous
  – easy if free space is well distributed in large chunks
- with use the free space becomes fragmented
  – and file I/O involves more head motion
- periodic in-place compaction and defragmentation
  – move the most popular files to the inner-most cylinders
  – copy all files into contiguous extents
  – Leave the free-list with large contiguous extents
- has the potential to significantly speed up file I/O

Open Files – Levels of Indirection



(Open Files – Levels of Indirection)

- open file references (UNIX user file descriptors)
  – array to associate open file index numbers w/files
- open file descriptors (UNIX file structures)
  – describes an open instance (session) of a file
    - current offset, access (read/write), lock status
- in-memory file descriptors (UNIX I-nodes)
  – copy of on-disk file description
- on-disk file descriptors (UNIX dinodes)
  – file description (ownership, protection, etc)
  – location (on disk) of the file's data

Extending a File

- client requests new chunk be assigned to file
  – may be an explicit allocation/extension request
  – may be implicit (e.g. write to a non-existant block)
- find a free chunk of space
  – traverse the free list to find an appropriate chunk
  – remove the chosen chunk from the free list
- associate it with the appropriate address in the file
  – go to appropriate place in the file or extent descriptor
  – update it to point to the newly allocated chunk

Deleting a file

- release all the space that is allocated to the file
  – UNIX, return each block to the free block list
  – MVS, return each extent to the free chunk list
    (coalescing adjacent extents where possible)
  – DOS does not free space, it uses garbage collection
- deallocate the file control lock
  – UNIX, zero i-node and return it to free list
  – MVS, zero the format 1 DSCB in the VTOC
  – DOS, zero first byte of the name in the parent directory
    (indicating that the directory entry is no longer in use)

Block Device Drivers

- generalizing abstraction – make all disks look same
- implement standard operations on their devices
  – asynchronous read (physical block #, buffer, bytecount)
  – asynchronous write (physical block #, buffer, bytecount)
- map logical block numbers to device addresses
  – e.g. logical block number to <cylinder, head, sector>
- encapsulate all the particulars of device support
  – I/O scheduling, initiation, completion, error handlings
  – size and alignment limitations

Device Independent Block I/O

- simplifying abstraction – better than generic disks
- an LRU buffer cache for disk data
  - hold frequently used data until it is needed again
  - hold pre-fetched read-ahead data until it is requested
- buffers for data re-blocking
  - adapting file system block size to device block size
  - adapting file system block size to user request sizes
- automatic buffer management
  - allocation, deallocation
  - automatic write-back of changed buffers

File Systems

- file systems implemented on top of block I/O
  - should be independent of underlying devices
- all file systems perform same basic functions
  - map names to files
  - map <file, offset> into <device, block>
  - manage free space and allocate it to files
  - create and destroy files
  - get and set file attributes
  - manipulate the file name space
- different implementations and options