

Files and File Systems

- 11A File Semantics
- 11B File Representation
- 11C Free-Space Representation

What is a File

- a file is a named collection of information
- primary roles of file system:
 - to store and retrieve data
 - to manage the media/space where data is stored
- typical operations:
 - where is the first block of this file
 - where is the next block of this file
 - where is block 35 of this file
 - allocate a new block to the end of this file
 - free all blocks associated with this file

File Systems: Semantics and Structure

2

Data and Metadata

- File systems deal with two kinds of information
- *Data* – the contents of the file
 - e.g. instructions of the program, words in the letter
- *Metadata* – Information about the file
 - e.g. how many bytes are there, when was it created
 - sometimes called *attributes*
- both must be persisted and protected
 - stored and connected by the file system

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);
}
```

File Systems: Semantics and Structure

4

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);
}
```

File Systems: Semantics and Structure

5

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 Systems: Semantics and Structure

6

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, ...

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

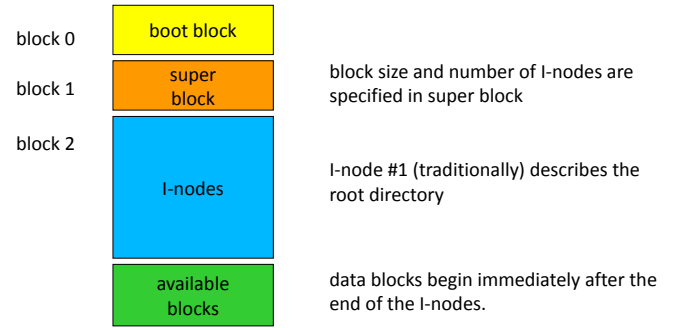
File System Goals

- 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
- descriptor location/# is file's *triple*

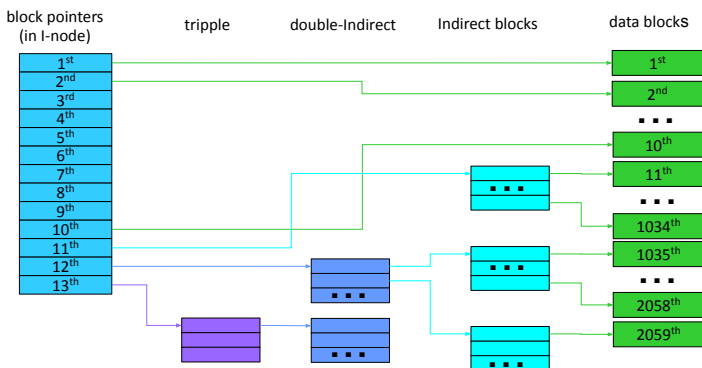
UNIX I-node

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

Ways to Manage Allocated Space

- a single large, contiguous *extent*
 - one pointer per file, very efficient I/O
 - hard to extend, external fragmentation, coalescing
- a linked lists of blocks
 - one pointer per file, one per extent
 - potentially long searches
- N block pointers per file
 - limits maximum file size to N blocks
 - but maybe some blocks contain pointers

Unix I-nodes and block pointers



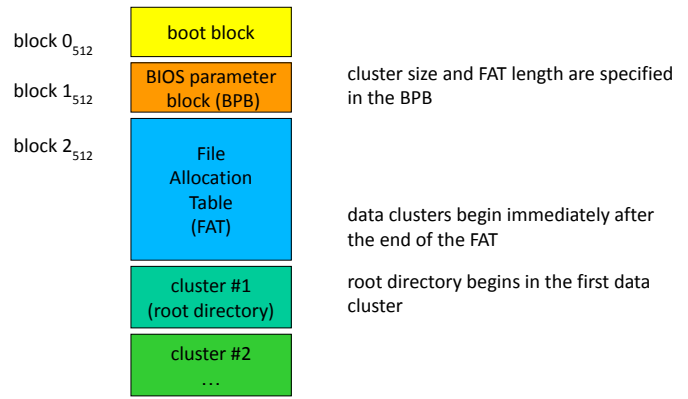
(Unix I-node block mapping)

- 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 indire)
- 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)

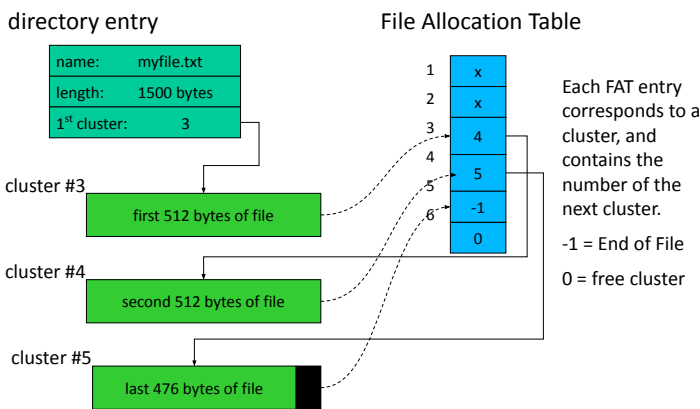
I-nodes – performance

- 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 through 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

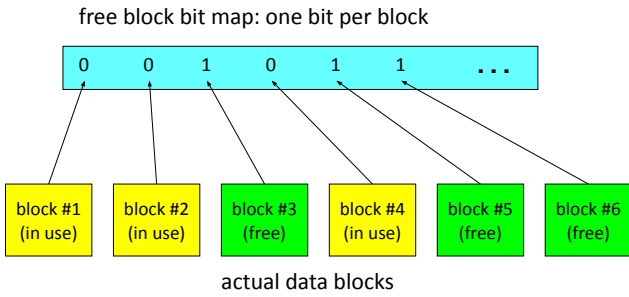
FAT – Performance/Capabilities

- 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
- file system free-list organization must address
 - speed of allocation and de-allocation
 - ability to allocate contiguous or near-by space
 - ability to coalesce and de-fragment

Bit Map Free Lists



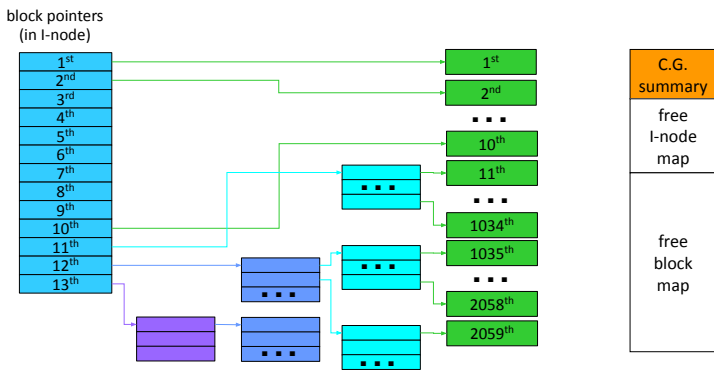
BSD file systems use bit-maps to keep track of both free blocks and free I-nodes in each cylinder group



(free space bit-maps)

- fixed sized blocks simplify free-lists
 - equal sized blocks do not require size information
 - all blocks are fungible (modulo performance)
- bit maps are a very efficient representation
 - minimal space to store the map
 - very code and cache efficient to search
- bit maps enable efficient allocation
 - easy to find chunks in a desired area
 - easy to coalesce adjacent chunks

FFS I-nodes and Free Lists



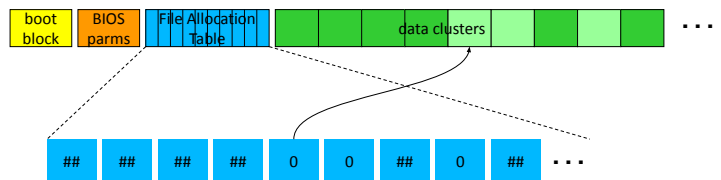
FFS create(/foo/bar)

	data bitmap	I-node bitmap	root I-node	foo I-node	bar I-node	root data	foo data	bar data[0]	bar data[1]	bar data[2]
search /			read							
search /						read				
search foo				read						
search foo							read			
new I-node		read								
new I-node		write								
update foo								write		
new I-node					read					
new I-node					write					
update foo				write						

FFS 3 x write(bar, one block)

	data bitmap	I-node bitmap	root I-node	foo I-node	bar I-node	root data	foo data	bar data[0]	bar data[1]	bar data[2]
find bar[0]					read					
new block	read									
new block		write								
write bar[1]								write		
update I-node					write					
find bar[1]					read					
new block	read									
new block		write								
write data									write	
update I-node					write					
find bar[2]					read					
new block	read									
new block		write								
write bar[2]										write
update I-node					write					

FAT Free Space



Each FAT entry corresponds to a cluster, and contains the number of the next cluster.

A value of zero indicates a cluster that is not allocated to any file, and is therefore free.



FAT Free Space

- can search for free clusters in desired cylinder
 - we can map clusters to cylinders
 - the BIOS Parameter Block describes the device geometry
 - look at first-cluster of file to choose desired cylinder
 - start search at first cluster of desired cylinder
 - examine each FAT entry until we find a free one
- if no free clusters, we must garbage collect
 - recursively search all directories for existing files
 - enumerate all of the clusters in each file
 - any clusters not found in search can be marked as free

(Extending a DOS/FAT file)

- note cluster number of current last cluster in file
- search FAT to find a free cluster
 - free clusters are indicated by a FAT entry of zero
 - look for a cluster in same cylinder as previous cluster
 - put -1 in FAT entry to indicate that this is the new EOF
 - this has side effect of marking new cluster as not free
- chain new cluster on to end of the file
 - put number of new cluster into FAT entry for last cluster

Reading and Assignments

Reading:

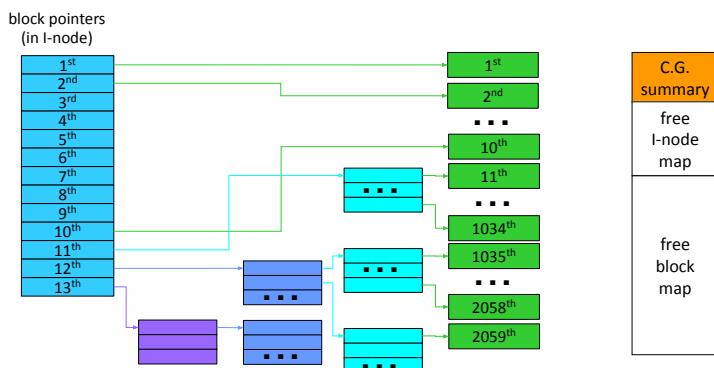
- Arpaci C40.4 ... file systems - directories

Projects:

- no lab this week

Supplementary Slides

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