# File Systems: Semantics & Structure

11A. File Semantics
11B. Namespace Semantics
11C. File Representation
11D. Free Space Representation
11E. Namespace Representation
11L. Disk Partitioning
11F. File System Integration



# 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



# 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);
}
```



# 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, ...



## 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

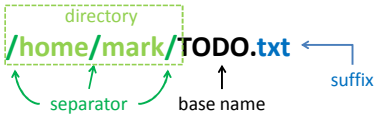
- smaller and faster than an SQL database
  - optimized for frequent small transfers
- *table* is equivalent of a file system
  - a *table* is a collection of *key/value* pairs
- *keys* have long names in a flat name space
  - *key* names are unique within a *table*
- value is a (typically 64-64MB) string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



## File Names and Name Binding

- file system knows files by internal 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

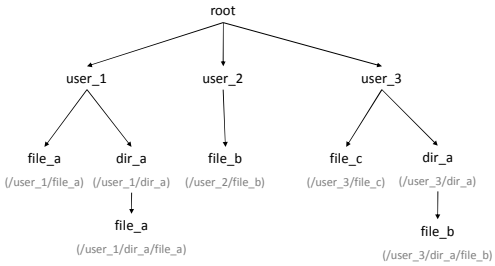


## Hierarchical Namespaces

- 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



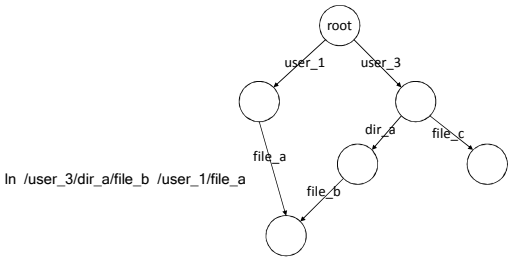



## True Names vs. Path Names

- Some file systems have “true names”
- DOS and ISO9660 have a single “path name”
  - files are described by directory entries
  - data is referred to by exactly one directory entry
  - each file has only one (character string) name
- Unix (and Linux) … have named links
  - files are described by I-nodes (w/unique I#)
  - directories associate names with I-node numbers
  - many directory entries can refer to same I-node



## Hard Links: example




Both names now refer to the same I-node

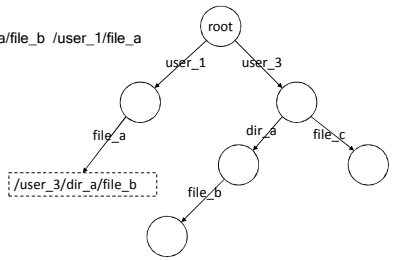
## Unix-style Hard Links

- 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

ln -s /user_3/dir_a/file_b /user_1/file_a




/user_1/file_a is now a macro for /user_3/dir_a/file_b



## 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



## Generalized Directories: Issues

- Can there be multiple links to a single file?
  - who can create new references to a file?
  - if one reference is deleted, does the file go away?
  - can the file's owner cancel other peoples' access?
- Can clients work their way back up the tree?
- Is the namespace truly a tree
  - or is it an a-cyclic directed graph
  - or an arbitrary directed graph
- Does namespace span multiple file systems?



## File System Goals

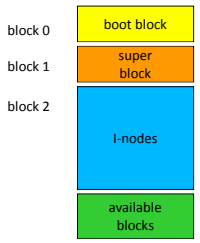
- ensure the privacy and integrity of all files
- efficiently implement name-to-file binding
  - find file associated with this name
  - list the file names in this part of the name space
- efficiently manage data associated w/each file
  - return data at offset X in file Y
  - write data Z at offset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




block size and number of I-nodes are specified in super block

I-node #1 (traditionally) describes the root directory

data blocks begin immediately after the end of the I-nodes.



## 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*

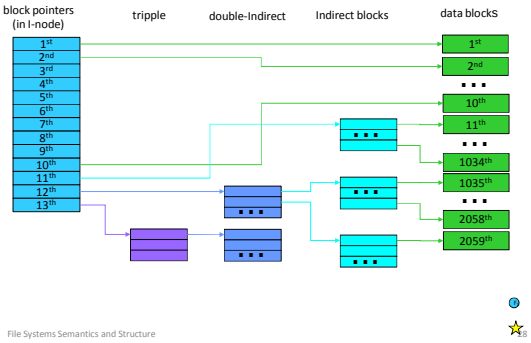


## Ways to Manage Allocated Space

- 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)

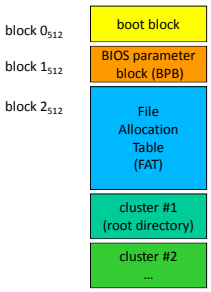
- I-node contains 13 block pointers
  - first 10 point to first 10 blocks of file
  - 11th points to an indirect block (e.g. 4k bytes = 1k blocks)
  - 12th points to a double indirect block (w/1k indirect blocks)
  - 13th points to a triple indirect block (w/1k double indirs)
- assuming 4k bytes per 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



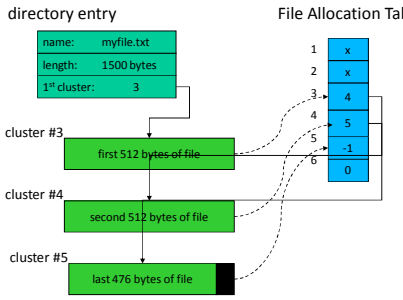

cluster size and FAT length are specified in the BPB

data clusters begin immediately after the end of the FAT

root directory begins in the first data cluster



## Clusters in a DOS FAT File




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

-1 = End of File

0 = free cluster



## (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

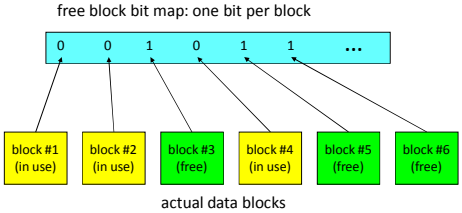


## Free Space Maintenance

- 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
- 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



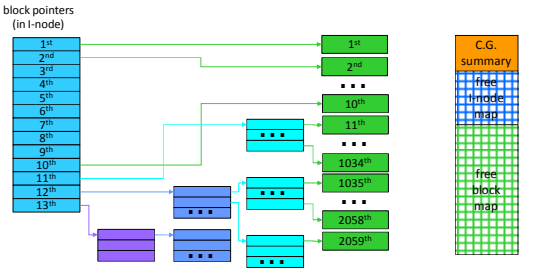

BSD file systems use bit-maps to keep track of both free blocks and free I-nodes in each cylinder group

## (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 | | | | | | |

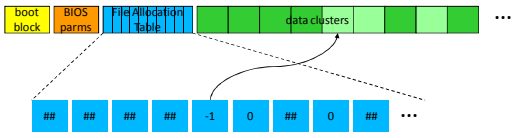


## FFS 3 x write(bar, one block)

| | 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.

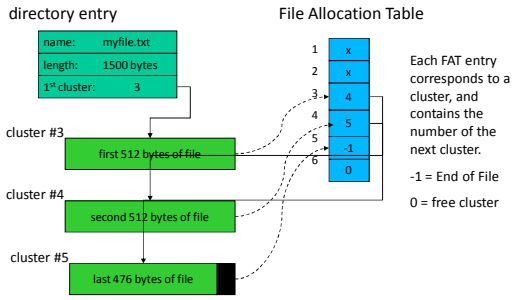
A value of zero indicates a cluster that is not allocated to any file, and is therefore free.



## 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

## Clusters in a DOS FAT File






## (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

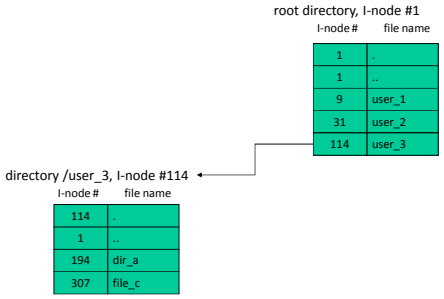


## Directories are usually files

- directories are a special type of file
  - used by OS to map file names into the associated files
- a directory contains multiple directory entries
  - each directory entry describes one file and its name
- user applications are allowed to read directories
  - to get information about each file
  - to find out what files exist
- only Operating System is allowed to write them
  - the file system depends on the integrity of directories



## UNIX Directories



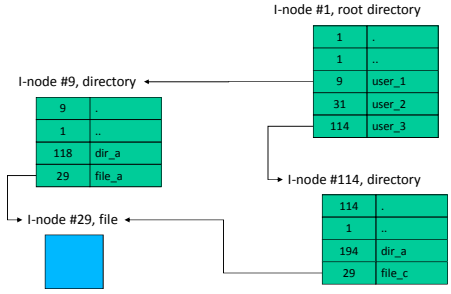



## (Example: UNIX Directories)

- 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

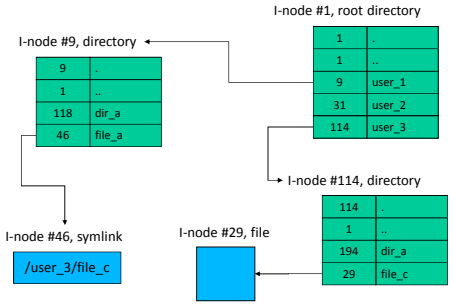


## Hard Links and De-allocation

- there can be many links to a single I-node
  - all links are equivalent
  - no link enjoys a preferential (e.g. master) status
- file exists as long as at least one link exists
- most easily implemented w/reference counts
  - increment with every link operation
  - decrement with every unlink operation
  - delete file when reference count goes to zero
- could be implemented with garbage collection



## Symbolic Links - example






## DOS Directories

root directory, starting in cluster #1

| file name | type | length | ... | 1st cluster |
|---|---|---|---|---|
| user_1 | DIR | 256 bytes | ... | 9 |
| user_2 | DIR | 512 bytes | ... | 31 |
| user_3 | DIR | 284 bytes | ... | 114 |

Directory /user_3, starting in cluster #114

| file name | type | length | ... | 1st cluster |
|---|---|---|---|---|
| .. | DIR | 256 bytes | ... | 1 |
| dir_a | DIR | 512 bytes | ... | 62 |
| file_c | FILE | 1824 bytes | ... | 102 |



## (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



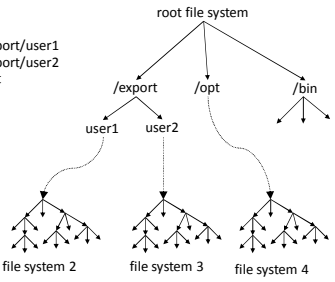
## Unix File System Mounts

- goal
  - make many file systems appear to be one giant
  - users need not be aware of file system boundaries
- mechanism
  - mount device on directory
  - creates a warp from the named directory to the top of the file system on the specified device
  - 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



## File Systems and Disks

- independence of multiple disks
  - they can be turned on and off independently
  - they can be backed-up and restored independently
  - some are physically removable (diskette, CD, Flash)
- file system spanning multiple (non RAID) disks is risky
  - losing one disk could lose parts of many files
  - better to lose all of some files and none of others
  - disks can be checked, repaired, restored independently
- people do put multiple file systems on a single disk
  - partitioning a physical disk into multiple logical disks



## Logical Disk Partitioning

- divide physical disk into multiple logical disks
  - perhaps in disk driver, perhaps through meta-driver
  - rest of system sees partitions as separate devices
- typical motivations
  - permit multiple OS to coexist on a single disk
    - e.g. a notebook that can boot either Windows or Linux
  - fire-walls for installation, back-up and recovery
    - e.g. separate personal files from the installed OS file system
  - fire-walls for free-space
    - running out of space on one file system doesn't affect others

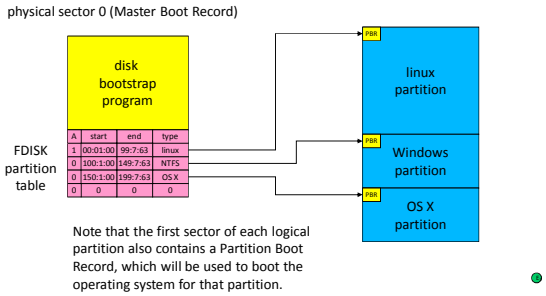


## Disk Partitioning Mechanisms

- some designed for use by a particular OS
  - e.g. Linux LVM partitions, understood by GRUB
- some designed to support multiple OS
  - e.g. DOS FDISK partitions, understood by BIOS
- there may be hierarchical partitioning
  - e.g. logical volumes within an FDISK partition
- should be possible to boot from any partition
  - direct from BIOS, or w/help from L2 bootstrap



## example: FDISK Disk Partitioning

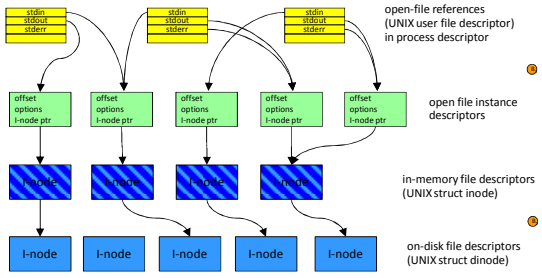





## A Simple Bootstrap Procedure

1. BIOS tries designated devices in a configurable order
2. Read MBR (Block 0) from chosen device and check for validity, and branch to it.
3. Scan FDISK table to find ACTIVE partition, read PBR (block 0) from chosen partition, and branch to it.
4. PBR loader finds and loads 2nd level bootstrap (e.g. GRUB), and branches to it.
5. 2nd level bootstrap (often a very sophisticated program) finds and loads operating system.
6. all early-stage loading is done by calls to BIOS

## 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

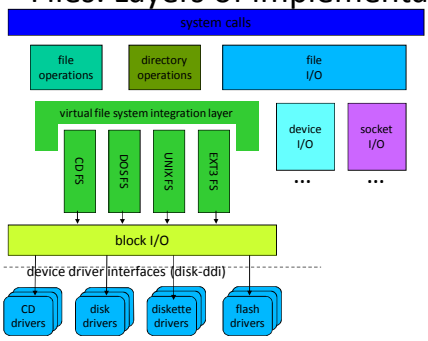


## File Systems

- 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



## 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



## 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



## Assignments

- Lab
  - Get started on Project 4B
- Reading (73pp, last big reading assignment)
  - A/D 41 FFS implementation
  - A/D 42 Crash-consistency
  - A/D 43 Logging File Systems
  - A/D 44 Data Integrity
  - A/D Appx I (6-10) SSD
  - Defragmentation

# Supplementary Slides



## Indexed Sequential Files

- record structured files for database like use
  - records may be fixed or variable length
- all records have one or more keys
  - records can be accessed by their key
  - sample key: student ID number
- new file update operations
  - insert record, delete record
- performance challenges
  - efficient insertion, key searches

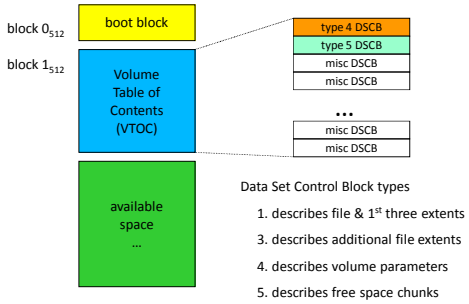


## What is an Index

- a table of contents for another file
  - lists keys, and pointers to associated records
  - built by examining the real data file
- enables much faster access to desired records
  - search for records in index rather than file
  - index is much shorter, and sorted by key(s)
- index is sometimes called an inverted file
  - files are accessed by record location, to find the data
  - index is accessed by data key, to find record location

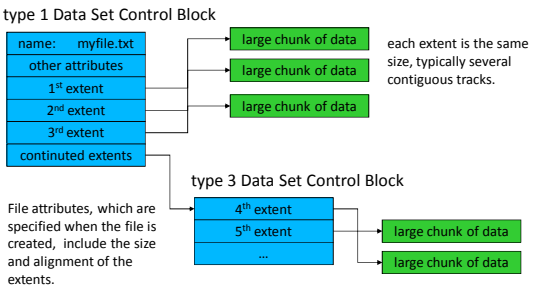


## MVS – Volume Structure






## MVS Files and Data Extents






## (MVS Volumes – Overview)

- Data Set Control Blocks at start of volume
  - describe volume parameters
  - describe allocated files and allocated space
  - describe unallocated space
- file space
  - allocated to files in variable length "extents"
    (extent size is determined when file is created)
- file index blocks – format 1 DSCBs
  - describe file attributes (and unit of space allocation)
  - points at first three extents, and extra extent DSCB

## MVS – Performance/Capabilities

- extent based allocation
  - user is required to specify extent size and alignment
  - enables large, efficient, contiguous, allocation
  - potentially very efficient file reads and writes
  - large extents may mean large internal fragmentation
  - variable size extents means external fragmentation
- finding the n'th extent involves following DSCBs
  - if VTOC is in memory, this needn't involve I/O
- no support for sparse files



## Indexed Sequential Files

- record structured files for database like use
  - records may be fixed or variable length
- all records have one or more keys
  - records can be accessed by their key
  - sample key: student ID number
- new file update operations
  - insert record, delete record
- performance challenges
  - efficient insertion, key searches



## What is an Index

- a table of contents for another file
  - lists keys, and pointers to associated records
  - built by examining the real data file
- enables much faster access to desired records
  - search for records in index rather than file
  - index is much shorter, and sorted by key(s)
- index is sometimes called an inverted file
  - files are accessed by record location, to find the data
  - index is accessed by data key, to find record location

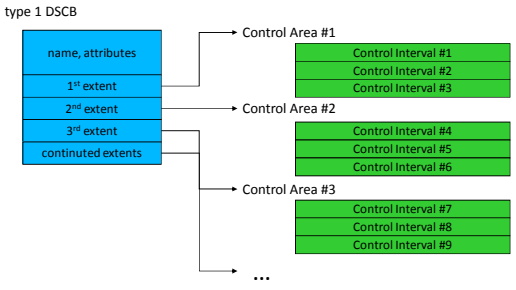


## Virtual Sequential Access Method (VSAM)

- flexible and very efficient data base access
  - designed for large commercial applications
- multiple access modes
  - entry sequenced, relative record number
  - key sequenced (primary and secondary keys)
  - linear (byte stream)
- multiple record types
  - fixed length, variable length
  - compressed, spanned (very long records)



## MVS/VSAM Extents & Control Areas

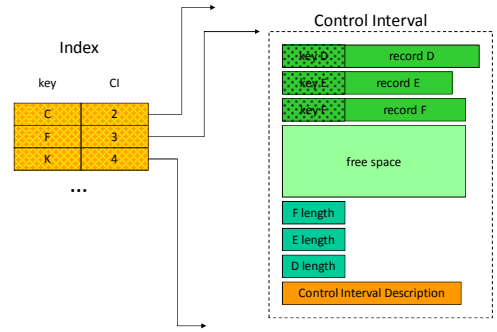





## (MVS VSAM Files)

- Data Organization
  - each contiguous extent is a "Control Area"
  - each "Control" Area is divided into "Control Intervals"
  - each "Control Interval" contains (keyed) data records
  - each record has a key, records appear in key-order
- Index Structure
  - indices are maintained as separate (inverted) files
  - index contains one record for each Control Interval
    - location (within VSAM file) of control interval
    - number of highest key stored in that control interval

## VSAM Control Intervals & Records



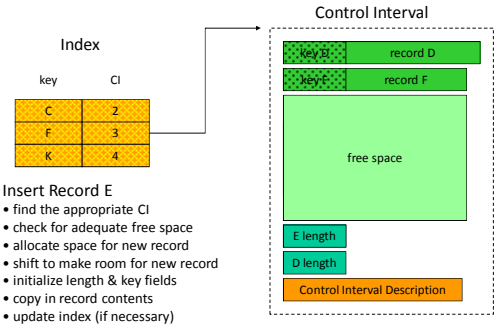



## MVS VSAM files – rationale

- Control Areas
  - all records in CA can be read without head-motion
  - thus, the order of CIs within a CA is not critical
- Control Intervals
  - index search identifies CI containing desired record
  - we may read or write an entire CI in one operation
  - records within a CI are in key-order, efficient searches
- Leave free space in each CI, free CIs in each CA
  - so we can do record insertions w/o I/O or head motion



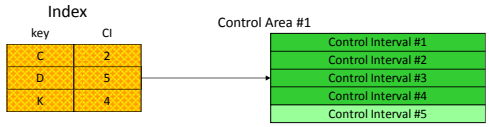
## MVS/VSAM Record Insertion (room in Control Interval)




Insert Record E
- find the appropriate CI
- check for adequate free space
- allocate space for new record
- shift to make room for new record
- initialize length & key fields
- copy in record contents
- update index (if necessary)



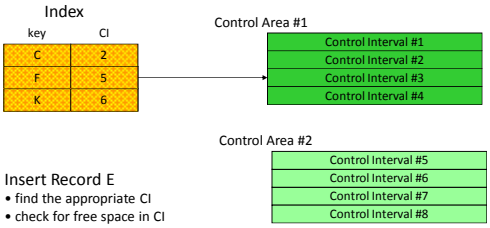
## MVS/VSAM Record Insertion (Control Interval overflow)




Insert Record E
- find the appropriate CI
- check for free space
- check for a free CI in same CA
- move ½ of records from full CI to new one
- update index to reflect new CI structure
- perform the insert as before



## MVS/VSAM Record Insertion (Control Area overflow)




Insert Record E
- find the appropriate CI
- check for free space in CI
- check for a free CI in same CA
- allocate a new CA (extent)
- move ½ of CIs in full CA into new CA
- update index to reflect new CA/CI structure
- perform the insert as before



## (MVS VSAM Record Insertion)

- Use index to find appropriate Control Interval
  - insert new record into the existing Control Interval
  - may involve shifting existing records to the right
- If Control Interval is is already full
  - find a free control interval within the control area
  - move half records in full control interval into new one
- if Control Area is completely full
  - allocate a new extent/Control Area
  - move half of CIs from full CA into the new CA
- update index accordingly

## MVS VSAM Free Space Management

- Dataset initially created with free space included
  - free record space within each Control Interval
  - free Control Intervals within each Control Area
  - user specifies reserve percentages for each
- After insertions/deletions dataset degenerates
  - free space is no longer evenly distributed throughout it
  - consecutive records are no longer contiguous
- Dataset can be copied out and reloaded
  - consecutive records will be allocated consecutively
  - free space will be well distributed throughout dataset

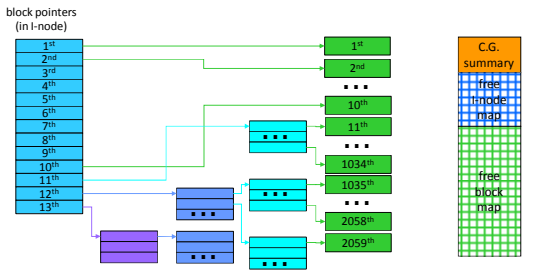


## File Systems: which is better

- DOS file systems are very simple
  - designed for simple sequential reads and writes
- MVS lets programs control file organization
  - select extent sizes to minimize fragmentation
  - size and alignment to maximize I/O throughput
  - many different file organizations are possible
- UNIX combines flexibility with ease of use
  - large & small files, random & sequential access
  - efficient allocation and I/O w/o help from program



## Unix File Extension



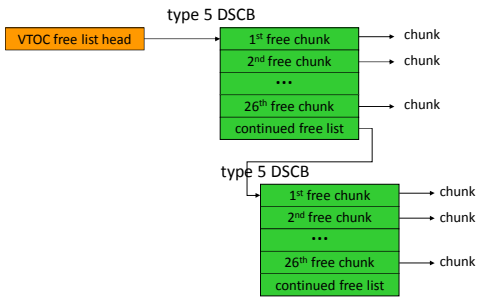



## (Extending a BSD/UNIX file)

- 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



## MVS Free Space Management

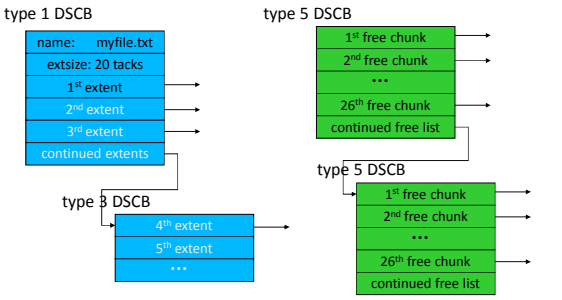





## (MVS Free Space)

- free chunks are of variable length
  - due to variable size of allocated extents
- described by format 5 DSCBs in VTOC
  - DSCB points to 26 non-contiguous free chunks
  - pointer to next format 5 DSCB in free list
- large extents make first-fit allocation practical
  - if a chunk is the right size, its location is secondary
- allocation, freeing and coalescing require no I/O
  - if the entire VTOC is kept in memory

## MVS Data and Free Extents



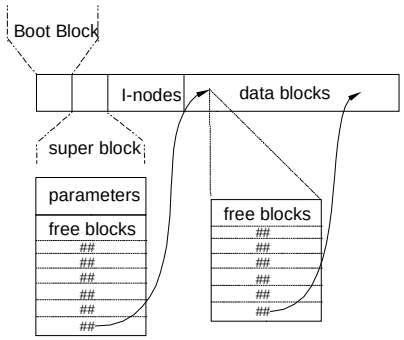



## (Extending an MVS file)

- note the required extent size (from format 1 DSCB)
- find an appropriate free chunk of space
  - first-fit search the chain of format 5 DSCBs
  - remove required extent size from chosen chunk
  - update format 5 DSCB to point at remainder (if any)
- attach the new chunk to the end of the file
  - find the last extent pointer in the file
  - if all full, allocate and attach a new format 3 DSCB
  - put pointer to new chunk in the last extent pointer



## Unix System V Free Space






## Unix System V free space

- superblock in memory for each active file system
  - contains list of first 100 free blocks
  - last of these blocks contains list of next 100 free blocks
- when you allocate the 100th block
  - move its pointer list into superblock
- when you free the 100th block
  - move superblock list into your block
- performance
  - only one I/O operation per hundred allocates/frees
  - allocation in specific cylinder is impractical