Distributed Computing Systems

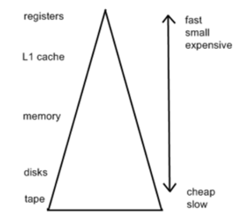
File Systems

Motivation – Process Need

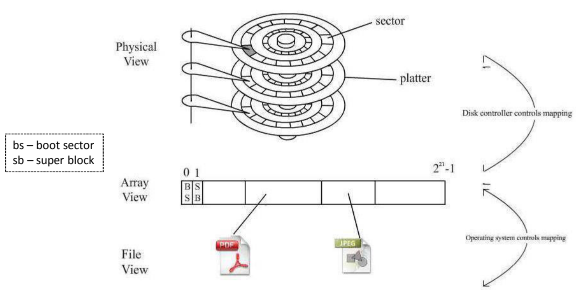
- Processes store, retrieve information
- When process terminates, memory lost
- How to make it persist?
- What if multiple processes want to share?

- Requirements:
  - large
  - ***persistent***
  - concurrent access

Solution? Disks are large, persistent!



Motivation – Disk Functionality (1 of 2)



- Sequence of fixed-size blocks
- Support reading and writing of blocks

Motivation – Disk Functionality (2 of 2)

- Questions that quickly arise
  - How do you find information?
  - How to map blocks to files?
  - How do you keep one user from reading another's data?
  - How do you know which blocks are free?

Solution? File Systems

Outline

- Files (next)
- Directories
- Disk space management
- Misc
- Example systems

File Systems

- Abstraction to disk (convenience)
  - "The only thing friendly about a disk is that it has persistent storage."
  - Devices may be different: tape, USB, IDE/SCSI, NFS
- Users
  - don't care about implementation details
  - care about interface
- OS
  - cares about implementation (efficiency and robustness)

File System Concepts

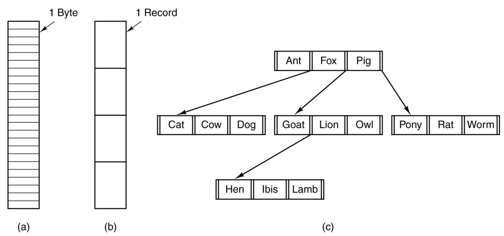
- *Files* - store the data
- *Directories* - organize files
- *Partitions* - separate collections of directories (also called "volumes")
  - all directory information kept in partition
  - `mount` file system to access
- *Protection* - allow/restrict access for files, directories, partitions

Files: The User's Point of View

- Naming: how does user refer to it?
- Example: `blah`, `BLAH`, `Blah`
  - Does case matter?
  - Users often don't distinguish, and in much of Internet no difference (e.g., email), but sometimes (e.g., URL path)
  - Windows: generally case doesn't matter, but is preserved
  - Linux: generally case matters
- Example: `file.c`, `file.com`
  - Does extension matter?
  - Software may distinguish (e.g., compiler for `.cpp`, Windows Explorer for application association)
  - Windows: explorer recognizes extension for applications
  - Linux: extension ignored by system, but software may use defaults

Structure

- What's inside?
  a) *Sequence of bytes* (most modern OSes (e.g., Linux, Windows))
  b) *Records* - some internal structure
  c) *Tree* - organized records



Type and Access

- Type:
  - *ascii* - human readable
  - *binary* - computer only readable
  - Allowed operations/applications (e.g., executable, c-file …) (via "magic number" or extension)
- Access Method:
  - *sequential* (for character files, an abstraction of I/O of serial device such as modem)
  - *random* (for block files, an abstraction of I/O to block device such as a disk)

Common Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

System Calls for Files

- Create
- Delete
- Truncate
- Open
- Read
- Write
- Append
- Seek
- Get attributes
- Set attributes
- Rename

Example: Program to Copy File

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                   /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);        /* ANSI prototype */

#define BUF_SIZE 4096                    /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                 /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

     /* Open the input file and create the output file */
     in_fd = open(argv[1], O_RDONLY);    /* open the source file */
     if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
     out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
     if (out_fd < 0) exit(3);            /* if it cannot be created, exit */
```

Example: Program to Copy File

```
     /* Copy loop */
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
     if (rd_count <= 0) break;                /* if end of file or error, exit loop */
          wt_count = write(out_fd, buffer, rd_count); /* write data */
          if (wt_count <= 0) exit(4);         /* wt_count <= 0 is an error */
     }

     /* Close the files */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                       /* no error on last read */
          exit(0);
     else
          exit(5);                            /* error on last read */
}
```

Zoom in on `open()` system call

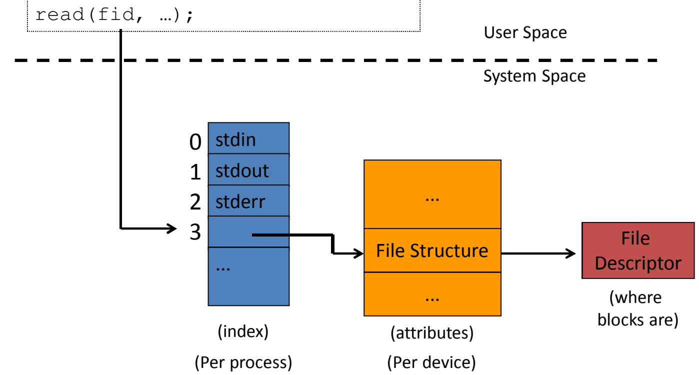
Example: Unix `open()`

```
int open(char *path, int flags [, int mode])
```

- `path` is name of file
- `flags` is bitmap to set switch
  - O_RDONLY, O_WRONLY, O_TRUNC …
  - O_CREATE then use `mode` for permissions
- success, returns index

Unix `open()` - Under the Hood

```
int fid = open("blah", flags);
read(fid, …);
```



Example: Windows `CreateFile()`

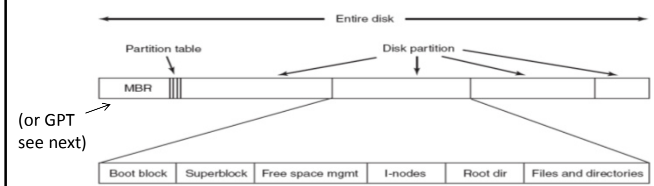
- Returns file object handle:

```
HANDLE CreateFile (
 lpFileName, // name of file
 dwDesiredAccess, // read-write
 dwShareMode,  // shared or not
 lpSecurity, // permissions
 ...
)
```

- File objects used for all: files, directories, disk drives, ports, pipes, sockets and console

File System Layout

- BIOS reads in program ("bootloader", e.g., grub) in Master Boot Record (MBR) in fixed location on disk
- MBR has partition table (start, end of each partition)
- Bootloader reads first block ("boot block") of partition
- Boot block knows how to read next block and start OS
- Rest can vary. Often "superblock" with details on file system
  - Type, number of blocks,...

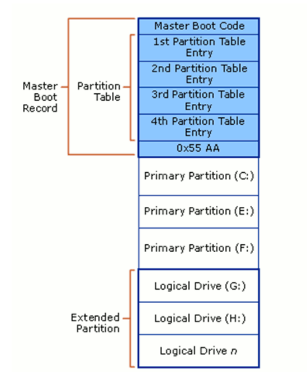


MBR vs. GPT

- MBR = Master Boot Record
- GPT = Guid Partition Table
- Both help OS know partition structure of hard disk
- Linux – default GPT (must use Grub 2), but can use MBR
- Mac – default GPT. Can run on MBR disk, but can't install on it
- Windows – 64-bit support GPT. Windows 7 default MBR, but Windows 8 default GPT

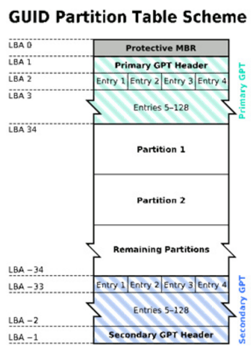
Master Boot Record (MBR)

- Old standard, still widely in use
- At beginning of disk, hold information on partitions
- Also code that can scan for active OS and load up boot code for OS
- Only 4 partitions, unless 4th is extended
- 32-bit, so partition size limited to 2TB
- If MBR corrupted → trouble!

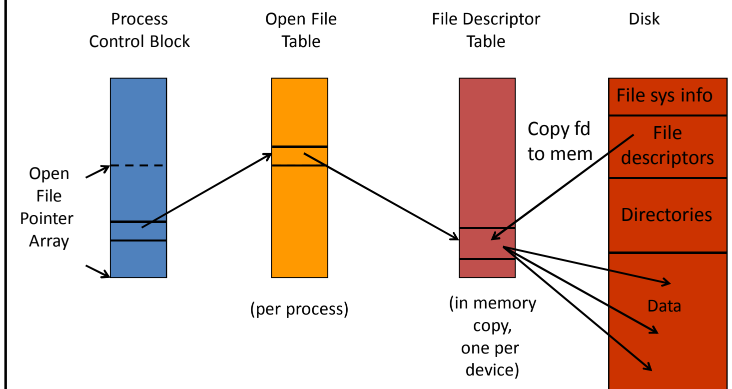


GUID Partition Table (GPT)

- Newest standard
- GUID = globally unique identifiers
- Unlimited partitions (but most OS limit to 128)
- Since 64-bit, 1 billion TB partitions (Windows limit 256 TB)
- Backup table stored at end
- CRC32 checksums to detect errors
- Protective MBR layer for apps that don't know about GPT



File System Implementation



Example – Linux (1 of 3)

Each `task_struct` describes a process

```
/* /usr/include/linux/sched.h */
struct task_struct {
     volatile long state;
     long counter;
     long priority;
     …
     struct files_struct *files;
     …
}
```

Example – Linux (2 of 3)

The `files_struct` data structure describes files process has open

```
/* /usr/include/linux/fs.h */
struct files_struct {
  int count;
  fd_set close_on_exec;
  fd_set open_fds;
  struct file *fd[NR_OPEN];
};
```

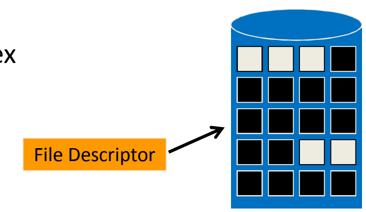
Example – Linux (3 of 3)

- Each open file is represented by a file data structure

```
struct file {
        mode_t f_mode;
        loff_t f_pos;
        unsigned short f_flags;
        unsigned short f_count;
        unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct file *f_next, *f_prev;
        int f_owner;
        struct inode *f_inode;       /* file descriptor */
        struct file_operations *f_op;
        unsigned long f_version;
        void *private_data;
};
```

File System Implementation

- Which blocks with which file?
- File descriptor implementations:
  - Contiguous
  - Linked List
  - Linked List with Index
  - I-nodes



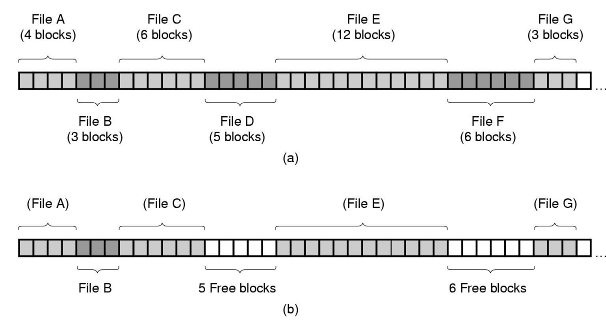
Contiguous Allocation (1 of 2)

- Store file as contiguous block
  - ex: w/ 1K block, 50K file has 50 consec. blocks

```
File A: start 0, length 2
File B: start 14, length 3
```

- Good:
  - Easy: remember location with 1 number
  - Fast: read entire file in 1 operation (length)
- Bad:
  - Static: need to know file size at creation
    - Or tough to grow!
  - Fragmentation: remember why we had paging in memory?

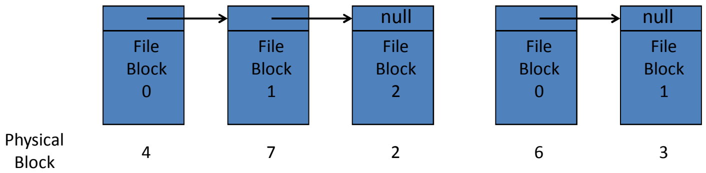
Contiguous Allocation (2 of 2)



a) 7 files
b) 5 files (file D and F deleted)

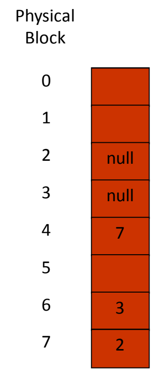
Linked List Allocation

- Keep linked list with disk blocks



- Good:
  - Easy: remember 1 number (location)
  - Efficient: no space lost in fragmentation
- Bad:
  - Slow: random access bad

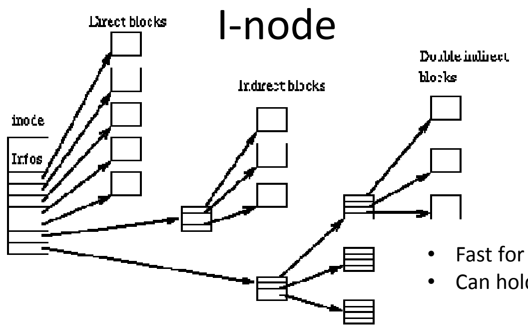
Linked List Allocation with Index



- Table in memory
  - MS-DOS FAT, Win98 VFAT
  - faster random access
  - can be large! E.g., 1 TB disk, 1 KB blocks
    - Table needs 1 billion entries
    - Each entry 3 bytes (say 4 typical)
    - → 4 GB memory!

Common format still (e.g., USB drives) since supported by many OSes

I-node



- Fast for small files
- Can hold large files

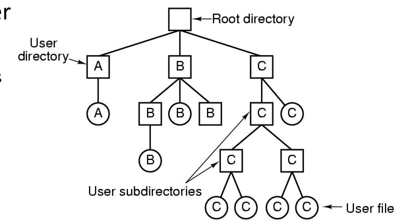
- Typically 15 pointers
  - 12 to direct blocks
  - 1 single indirect
  - 1 doubly indirect
  - 1 triply indirect
- Pointers per block? Depends upon block size and pointer size
- E.g., 1k byte block, 4 byte pointer → each indirect has 256 pointers
- Max size? Same.
- E.g., 4KB block → max size 2 TB

Outline

- Files (done)
- Directories (next)
- Disk space management
- Misc
- Example systems

Directories

- Just like files
  - Have data blocks
  - File descriptor to map which blocks to directory
- But have special bit set so user process cannot modify contents
  - data in directory is information / links to files
  - modify only through system call
  - (See `ls.c`)
- Organized for:
  - efficiency - locating file quickly
  - convenience - user patterns
    - groups (`.c`, `.exe`), same names
- Tree structure, directory the most flexible
  - User sees hierarchy of directories

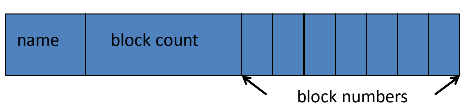


System Calls for Directories

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

Directories

- Before reading file, must be opened
- Directory entry provides information to get blocks
  - disk location (blocks, address)
- Map `ascii` name to *file descriptor*

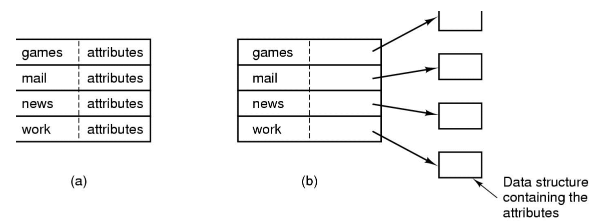


Where are attributes stored?

Options for Storing Attributes

a) Directory entry has attributes (Windows)

b) Directory entry refers to file descriptor (e.g., i-node), and descriptor has attributes (Unix)



Windows (FAT) Directory

- Hierarchical directories
- Entry:
  - name - date
  - type (extension) - block number (w/FAT)
  - time

name	type	attrib	time	date	block	size

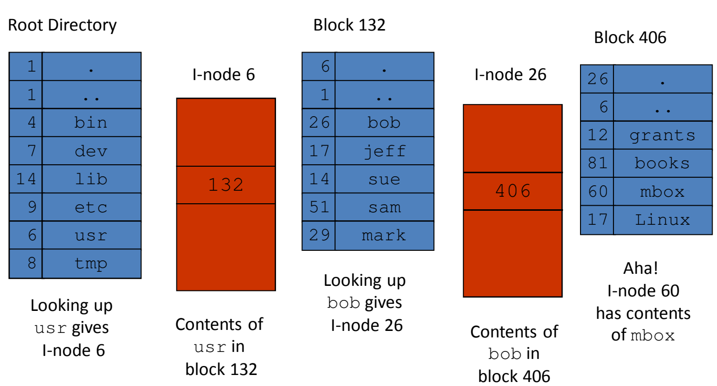
Unix Directory

- Hierarchical directories
- Entry:
  - name
  - i-node number (try "`ls -i`" or "`ls -iad .`")

inode	name

- Example, say want to read data from below file
  `/usr/bob/mbox`
  Want consents of file, which is in blocks
  Need file descriptor (i-node) to get blocks
  How to find the file descriptor (i-node)?

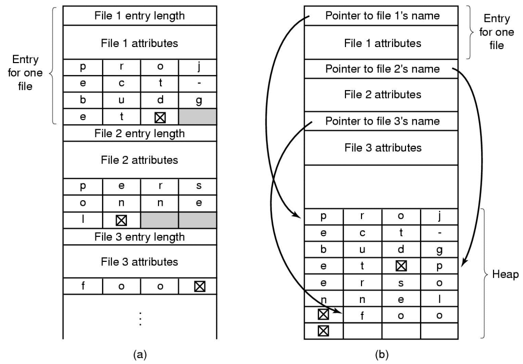
Unix Directory Example



Length of File Names

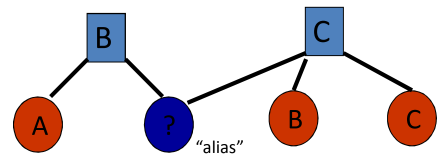
- Above, each directory entry is name (and attributes) plus descriptor
- How long should file names be?
- If fixed small, will hit limit (users don't like)
- If fixed large, may be wasted space (internal fragmentation)
- Solution → allow variable length names

Handling Long Filenames



a) Compact (all in memory, so fast) on word boundary
b) Heap to file

Same File in More than One Location



(Instead of tree, really have directed acyclic graph)

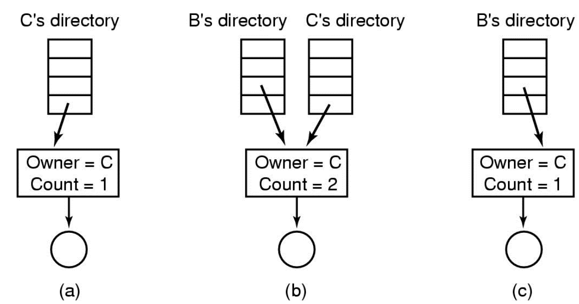
- Possibilities for the "alias":
  I. Directory entry contains disk blocks?
  II. Directory entry points to attributes structure?
  III. Have new type of file to redirect?

Will review each implementation choice, next

Possible Implementations

I. Directory entry contains disk blocks?
   – Contents (blocks) may change
   – What happens when blocks change?
II. Directory entry points to file descriptor?
   – If removed, refers to non-existent file
   – Must keep count, remove only if 0
   – *Hard link*
   – Similar if delete file in use (show example)
   – What about hard link file across partitions?

Possible Implementation ("hard link")



a) Initial situation
b) After link created
c) Original owner removes file (what if quotas?)

Possible Implementation ("soft link")

III. Have new type of file to redirect?
  - New file only contains alternate name for file
  - Overhead, must parse tree second time
  - *Soft* link (or *symbolic link)*
    - Note, *shortcut* in Windows only viewable by graphic browser, are absolute paths, with metadata, can track even if move
    - Does have mklink (hard and soft) for NTFS
  - Often have max link count in case loop (show example)
  - What about soft link across partitions?

Robust File Systems

- Consider removing a file
  a. Remove file from directory entry
  b. Return all disk blocks to pool of free disk blocks
  c. Release the file descriptor (i-node) to the pool of free descriptors
- What if system crashes in the middle?
  - i-node becomes orphaned (`lost+found`, 1 per partition)
  - if flip steps, blocks/descriptor free but directory entry exists
    - This is worse – can access blocks unintentionally!
- Solution? → Journaling File Systems

Journaling File Systems

1. Write intent to do actions a-c to log *before* starting
   - Note, may read back to verify integrity
2. Perform operations
3. Erase log

- If system crashes, when restart read log and apply operations
- Logged operations must be idempotent (can be repeated without harm)
- Windows: `NTFS`; Linux: `Ext3`

Outline

- Files (done)
- Directories (done)
- Disk space management (next)
- Misc
- Example systems

Disk Space Management

- *n* bytes → choices:
  1. contiguous
  2. blocks
- Similarities with memory management
  - *contiguous* is like variable-sized partitions
    - but compaction by moving on disk very slow!
    - so use blocks
  - *blocks* are like paging (can be wasted space)
    - how to choose block size?
- (Note, physical disk block size typically 512 bytes, but file system logical block size chosen when formatting)
- Depends upon size of files stored

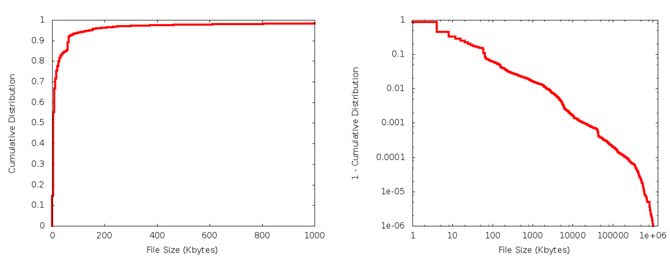
File Sizes in Practice (1 of 2)

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

- (VU – University circa 2005, Web – Commercial Web server 2005)
- Files trending larger. But most small. What are the tradeoffs?

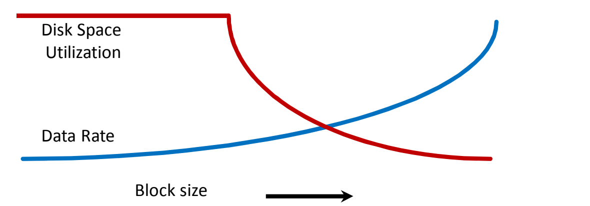
File Sizes in Practice (2 of 2)



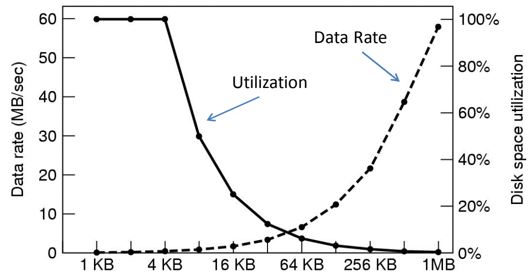
Claypool Office PC
Linux Ubuntu
March 2014

Choosing Block Size

- Large blocks
  - faster throughput, less seek time, more data per read
  - wasted space (internal fragmentation)
- Small blocks
  - less wasted space
  - more seek time since more blocks to access same data

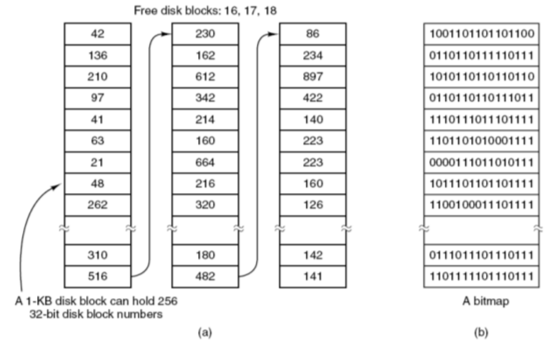


Disk Performance and Efficiency



- Assume 4 KB files.
- At crossover (~64 KB), only 6.6 MB/sec, Efficiency 7% (both bad)
- Most file systems pick 1KB – 4 KB
- But disks are cheap, so could argue for larger and not worry about waste

Keeping Track of Free Blocks



a) Linked-list of free blocks
b) Bitmap of free blocks

Keeping Track of Free Blocks

a) Linked list of free blocks
   - 1K block, 32 bit disk block number
     = 255 free blocks/block (one points to next block)
   - 500 GB disk has 488 millions disk blocks
     - About 1,900,000 1 KB blocks

b) Bitmap of free blocks
   - 1 bit per block, represents free or allocated
   - 500 GB disk needs 488 million bits
     - About 60,000 1 KB blocks

Tradeoffs

- Bitmap usually smaller since 1-bit per block rather than 32 bits per block
- Only if disk is nearly full does linked list require fewer blocks
- If enough RAM, bitmap method preferred since provides locality, too
- If only 1 “block” of RAM, and disk is full, bitmap method may be inefficient since have to load multiple blocks to find free space
  - linked list can take first in line

File System Performance

- DRAM ~5 nanoseconds, Hard disk ~5 milliseconds
  - Disk access 1,000,000x slower than memory!
  - →reduce number of disk accesses needed
- Block/buffer cache
  - cache to memory
- Full cache? Replacement algorithms use: FIFO, LRU, 2nd chance …
  - exact LRU can be done (why?)
- Pure LRU inappropriate sometimes
  - crash w/i-node can lead to inconsistent state
  - some rarely referenced (double indirect block)

Modified LRU

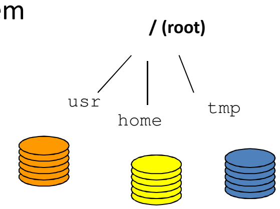
- Is the block likely to be needed soon?
  - if no, put at beginning of list
- Is the block essential for consistency of file system?
  - write immediately
- Occasionally write out all
  - `sync`

Outline

- Files (done)
- Directories (done)
- Disk space management (done)
- Misc (next)
  - partitions (`fdisk`, `mount`)
  - maintenance
  - quotas
- Example systems
- Distributed file systems

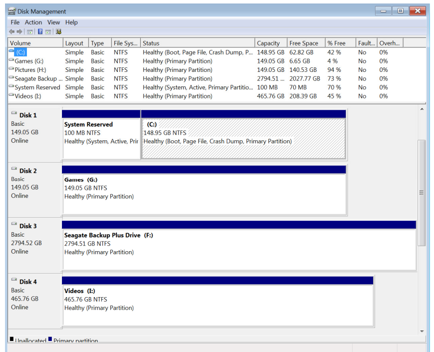
Partitions

- `mount`, `unmount`
  - load *super-block* from disk
  - pick access point in file-system
- Super-block
  - file system type
  - block size
  - free blocks
  - free i-nodes



Partitions: `fdisk`

- Partition is large group of sectors allocated for specific purpose
  - IDE disks limited to 4 physical partitions
  - logical (extended) partition inside physical partition
- Specify number of cylinders to use
- Specify type
  - "magic" number recognized by OS



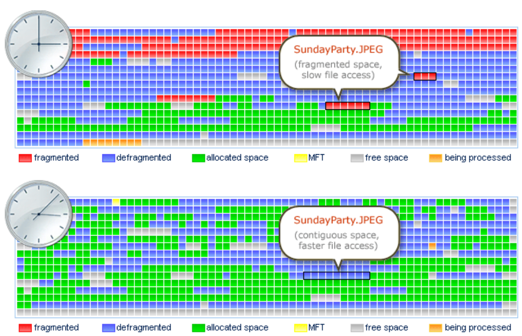
(Show example?)

("System Reserved" partition for Windows contains OS boot code and code to do HDD decryption, if set)

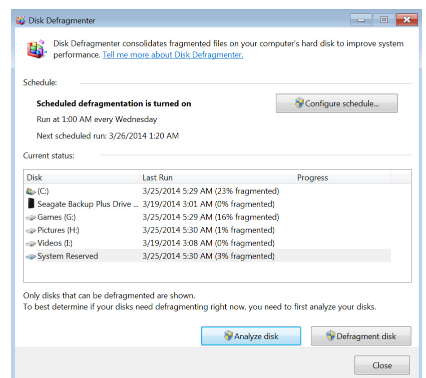
File System Maintenance

- Format:
  - create file system structure: super block, i-nodes
  - `format` (Windows), `mke2fs` (Linux)
    (Show "`format /?`", "`man mke2fs`")
- "Bad blocks"
  - most disks have some (even when brand new)
  - `chkdsk` (Win, or properties->tools->error checking) or `badblocks` (Linux)
  - add to "bad-blocks" list (file system can ignore)
- Defragment (see picture next slide)
  - arrange blocks allocated to files efficiently
- Scanning (when system crashes)
  - `lost+found`, correcting file descriptors...

Defragmenting (Example, 1 of 2)

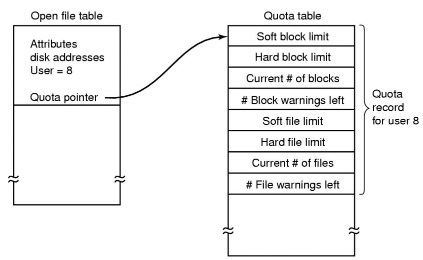


Defragmenting (Example, 2 of 2)



Disk Quotas

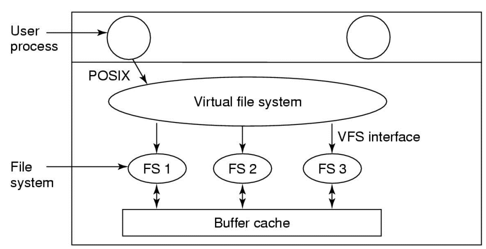
- Table 1: Open file table in memory
  - when file size changed, charged to user
  - user index to table 2
- Table 2: quota record
  - soft limit checked, exceed allowed w/warning
  - hard limit never exceeded
- Limit: blocks, files, i-nodes
  - Running out of i-nodes as bad as running out of blocks
- Overhead? Again, in memory



Outline

- Files (done)
- Directories (done)
- Disk space management (done)
- Misc (done)
- Example systems (next)
  - Linux
  - Windows

Linux File System



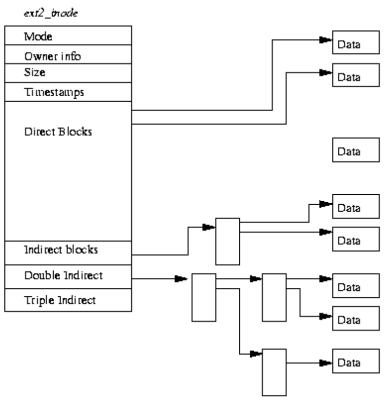
- Virtual FS allows loading of many different FS, without changing process interface
  - Still have `struct file_struct`, `open()`, `creat()`, …
- When build/install, FS choices → ext3/4, hfps, DOS, NFS, NTFS, smbfs, is9660, … (about 2 dozen)
- ext3 is “default” for many, most popular
  - Changing to ext4

Linux File System: ext3fs

- “Extended” (from Minix) file system, version 2
  - (Minix a Unix-like teaching OS by Tanenbaum)
- `ext2fs`
  - Long file names, long files, better performance
  - Main for many years
- `ext3fs`
  - Fully compatible with `ext2`
  - Adds journaling
- `ext4fs`
  - Extents (for free space management)
  - Pre-reserved, multi-block allocation
  - Better timestamp granularity

Linux File System: i-nodes (1 of 2)

- Uses i-nodes
  - *mode* for file, directory, symbolic link
  - ...



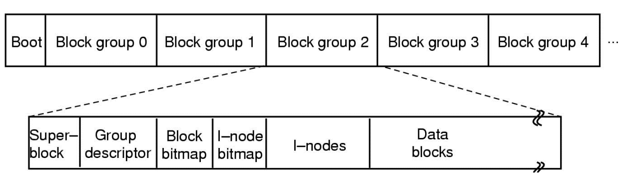
Linux File System: i-nodes (2 of 2)

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Linux File System: Blocks

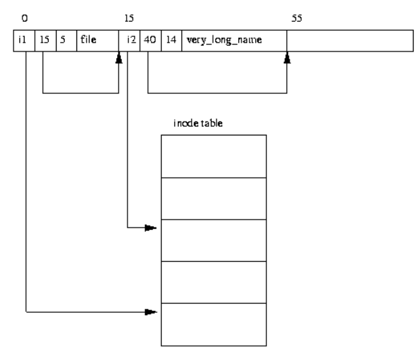
```
% sudo tune2fs -l /dev/sda1 | grep Block
Block count:              60032256
Block size:               4096
Blocks per group:         32768
```

- Default block size
- For higher performance
  - performs I/O in chunks (reduce requests)
  - clusters adjacent requests (block *groups*)
- Group has:
  - bit-map of free blocks and free i-nodes
  - copy of super block

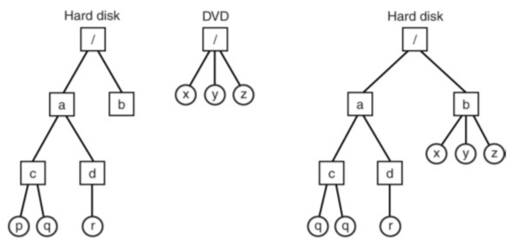


Linux File System: Directories

- Directory just special file with names and i-nodes



Linux File System: Unified



- (left) separate file trees (ala Windows)
- (right) after mounting "DVD" under "b" Linux

Linux Filesystem: ext3fs

- Journaling – internal structure assured
  - *Journal* (lowest risk) - Both metadata and file contents written to journal before being committed.
    - Roughly, write twice (journal and data)
  - *Ordered* (medium risk) - Only metadata, not file contents. Guarantee write contents before journal committed
    - Often the default
  - *Writeback* (highest risk) - Only metadata, not file contents. Contents might be written before or after the journal is updated. So, files modified right before crash can be corrupted
- No built-in defragmentation tools
  - Probably not much needed

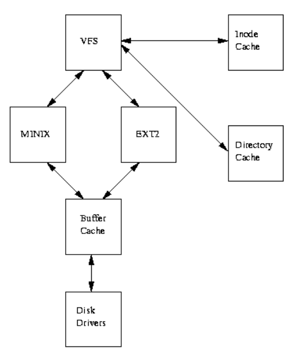
```
yukon% sudo fsck -nvf /dev/sda1
…
942826  inodes used (6.28%)
  1138  non-contiguous files (0.1%)
   821  non-contiguous directories (0.1%)
```

Linux Filesystem: `/proc`

- Contents of "files" not stored, but computed
- Provide interface to kernel statistics
- Most read only, access using Unix text tools
  - e.g., `cat /proc/cpuinfo | grep model`
- enabled by "virtual file system"

(Windows has `perfmon`)

(Show examples e.g., `cd /proc/self`)



Windows NT File System: NTFS

- Background: Windows had FAT
- FAT-16, FAT-32
  - 16-bit addresses, so limited disk partitions (2 GB)
  - 32-bit can support 2 TB
  - No security
- NTFS default in Win XP and later
  - 64-bit addresses

NTFS: Fundamental Concepts

- File names limited to 255 characters
- Full paths limited to 32,000 characters
- File names in unicode (other languages, 16-bits per character)
- Case sensitive names ("Foo" different than "FOO")
  - But Win32 API does not fully support

NTFS: Fundamental Concepts

- File not sequence of bytes, but multiple attributes, each a stream of bytes
- Example:
  - One stream name (short)
  - One stream id (short)
  - One stream data (long)
  - But can have more than one long stream
- Streams have metadata (e.g., thumbnail image)
- Streams fragile, and not always preserved by utilities over network or when copied/backed up

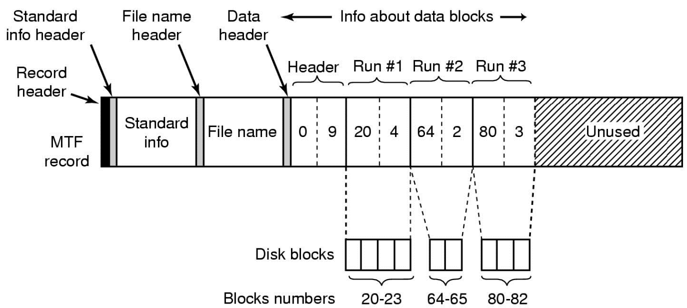
NTFS: Fundamental Concepts

- Hierarchical, with "\" as component separator
  - Throwback for MS-DOS to support CP/M microcomputer OS
- Supports links, but only for POSIX subsystem

NTFS: File System Structure

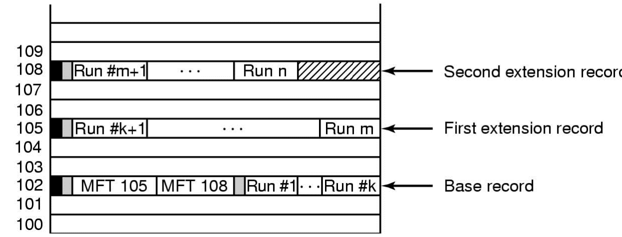
- Basic allocation unit called a *cluster* (block)
  - Sizes from 512 bytes to 64 Kbytes (most 4 KBytes)
  - Referred to by offset from start, 64-bit number
- Each volume has Master File Table (MFT)
  - Sequence of 1 KByte records
  - Bitmap to keep track of which MFT records are free
- Each MFT record
  - Unique ID - MFT index, and "version" for caching and consistency
  - Contains attributes (name, length, value)
  - If number of extents small enough, whole entry stored in MFT (faster access)
- Bitmap to keep track of free blocks
- Extents to keep clusters of blocks

NTFS: Storage Allocation



- Disk blocks kept in runs (extents), when possible

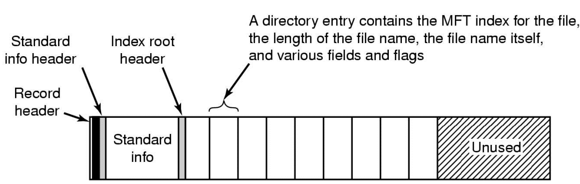
NTFS: Storage Allocation



- If file too large, can link to another MFT record

NTFS: Directories

- Name plus pointer to record with file system entry
- Also cache attributes (name, sizes, update) for faster directory listing
- If few files, entire directory in MFT record

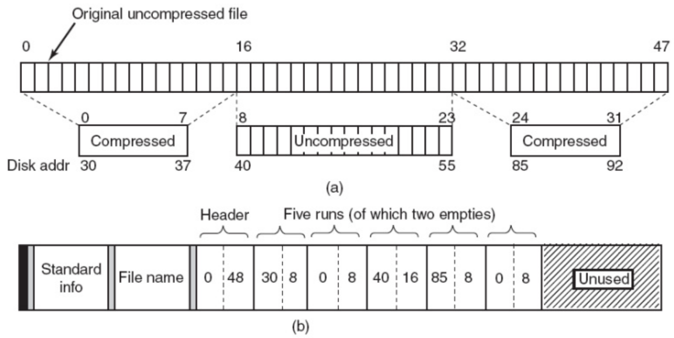


NTFS: Directories

- But if large, linear search can be slow
- Store directory info (names, perms, …) in B+ tree
  - Every path from root to leaf "costs" the same
  - Insert, delete, search all $O(\log_F N)$
    - F is the "fanout" (typically 3)
  - Faster than linear search O(N) versus $O(\log_F N)$
  - Doesn't need reorganizing like binary tree

NTFS: File Compression

- Transparent to user
  - Can be created (set) in compressed mode
- Compresses (or not) in 16-block chunks



NTFS: Journaling

- Many file systems lose metadata (and data) if powerfailure
  - `fsck`, `chkdsk` when reboot
  - Can take a looong time and lose data
    - `lost+found`
- Recover via "transaction" model
  - Log file with redo and undo information
  - Start transactions, operations, commit
  - Every 5 seconds, checkpoint log to disk
  - If crash, redo successful operations and undo those that don't commit
- Note, doesn't cover user data, only meta data