Distributed Computing Systems

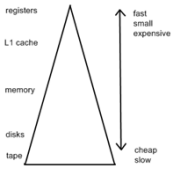
File Systems

Motivation – Process Need

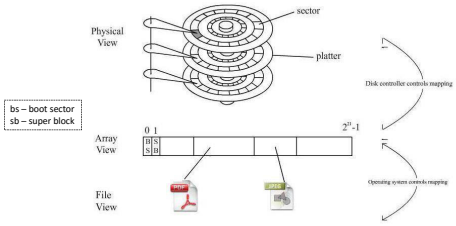
- Processes store, retrieve information
- When process terminates, memory lost
- How to make it persist?
- What if multiple processes want to share?

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

Solution? Files 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 file 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, SSD, IDE/SCSI, NFS
- Users
  - don't care about implementation details
  - care about interface
- OS
  - cares about implementation (efficiency and robustness)

File System Concepts

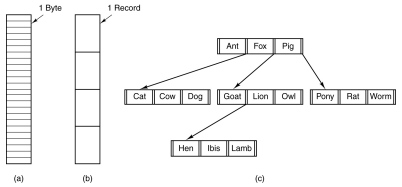
- *Files* - store the data
- *Directories* - organize files
- *Partitions* - separate collections of directories (also called "volumes")
  - all directory information kept in partition
  - `mount` file system to access
- *Protection* - allow/restrict access for files, directories, partitions

Files: The User's Point of View

- Naming: how does user refer to it?
- Does case matter? Example: `blah`, `BLAH`, `Blah`
  - Users often don't distinguish, and in much of Internet no difference (e.g., domain name), but sometimes (e.g., URL path)
  - Windows: generally case doesn't matter, but is preserved
  - Linux: generally case matters
- Does extension matter? Example: `file.c`, `file.com`
  - 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 (rarely today)
  c) *Tree* - organized records within file space (rarely today)



Type and Access

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

(see next slide)

Determining File Type – Unix `file` (1 of 2)

```
$ cat TheLinuxCommandLine
```

- What is displayed?

```
%PDF-1.6^M%âãÏÓ^M 3006 0 obj^M<>stream^M
...
```

- How to determine file type? → `file`

```
$ file grof
```

- **`grof: PostScript document text`**

```
$ file Desktop/
```

- **`Desktop/: directory`**

```
$ file script.sh
```

- **`script.sh: Bourne-Again shell script, ASCII, exe`**

```
$ file a.out
```

- **`a.out: ELF 32-bit LSB executable, Intel 80386...`**

```
$ file TheLinuxCommandLine
```

- **`TheLinuxCommandLine: PDF document, version 1.6`**

Determining File Type – Unix `file` (2 of 2)

1. Use `stat()` system call (see Project 1; see also `stat` system program)
   - "mode" for special file (socket, sym link, named pipes)
2. Try examining parts of file ("magic number")
   - Bytes with specific format, specific location
     `25 50 44 46, offset 0` → PDF
   - Custom, too (see `man magic`)
3. Examine content
   - Text? → Examine blocks for known types (e.g., `tar`)
4. Otherwise → `data`

https://en.wikipedia.org/wiki/List_of_file_signatures

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 (1 of 2)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <libgen.h>
#include <errno.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 */
```

"man" tells what system include files are needed

Command line args.

Example: Program to Copy File (2 of 2)

```
    /* Copy loop */
    while (1) {
        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                                                /* error on last read */
        exit(5);
}
```

Next → Zoom in on **open()** system call

```
int in_fd = open(argv[1], O_RDONLY); /* open file for reading */
```

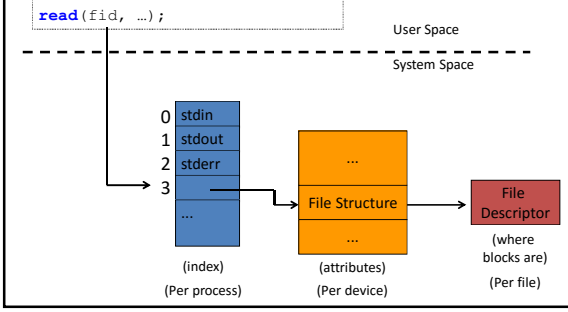
Example: Unix **open()**

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

- `path` is name of file (NULL terminated string)
- `flags` is bitmap to set switch
  - O_RDONLY, O_WRONLY, O_TRUNC …
  - O_CREATE then use `mode` for permissions
- success, returns `index`
  - On error, `-1` and set `errno` (see Project 1)

Unix **open()** – Under the Hood

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



Example: Windows `CreateFile()`

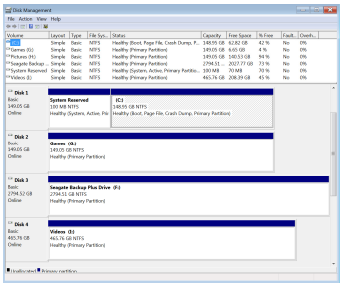
- Returns file object handle:

```
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

Disk Partitions

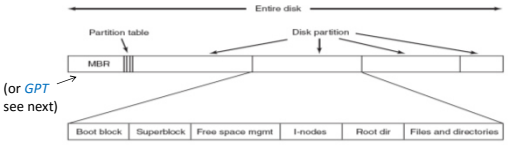
- Partition large group of sectors allocated for specific purpose
- Specify number of cylinders to use
- Specify type
- Linux: "`df -h`" and "`fdisk`"



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

File System Layout

- BIOS reads in program ("bootloader", e.g., `grub`) from known disk location (*Master Boot* Record or *GUID Partition Table*)
- MBR/GPT 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,…

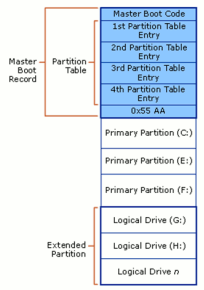


MBR vs. GPT

- MBR = *Master Boot Record*
  - Older standard, still in widespread use
- GPT = *GUID (Globally Unique ID) Partition Table*
  - Newer standard
- 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 & 10 default GPT

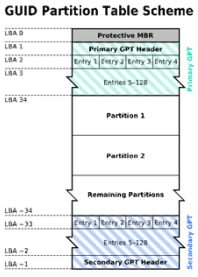
Master Boot Record (MBR)

- 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!

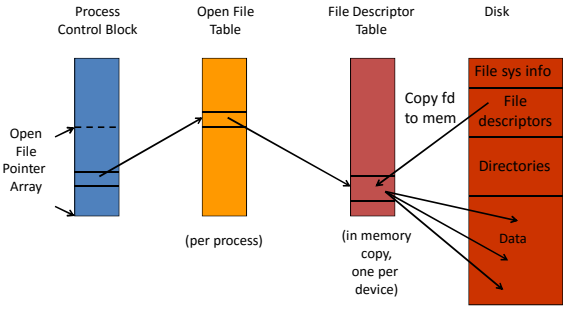


GUID Partition Table (GPT)

- Newest standard
- GUID = globally unique identifiers
- Unlimited partitions (but most OSes limit to 128)
- Since 64-bit, 1 billion TB (1 Zettabyte) partition size (Windows limit ~18 million 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, refers to open file table

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

Example – Linux (2 of 3)

The `files_struct` data structure describes files process has open, refers to file descriptor table

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

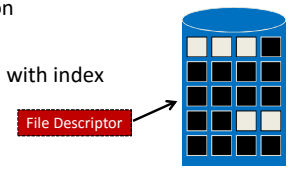
Example – Linux (3 of 3)

Each open file is represented by file descriptor, refers to blocks of data on disk

```
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 (next)
        struct file_operations *f_op;
        unsigned long f_version;
        void *private_data;
};
```

File System Implementation

- Core data to track: which blocks with which file?
- *File descriptor* implementations:
  - Contiguous allocation
  - Linked list allocation
  - Linked list allocation with index
  - inode



Contiguous Allocation (1 of 2)

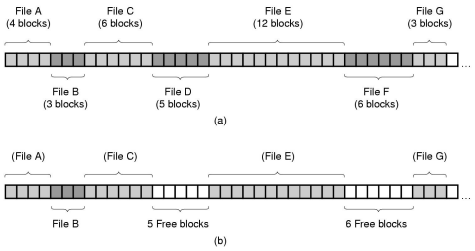
- Store file as contiguous block
  - ex: w/ 1K block, 50K file has 50 consecutive blocks

```
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?

(Example next slide)

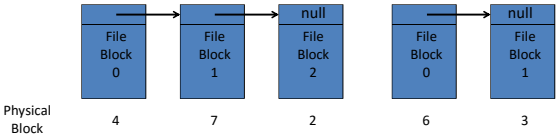
Contiguous Allocation (2 of 2)



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

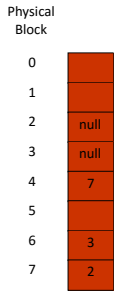
Linked List Allocation

- Keep linked list with disk blocks



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

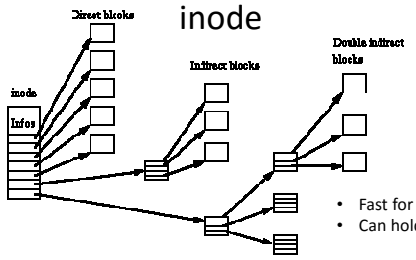
Linked List Allocation with Index



- Table in memory
  - MS-DOS FAT, Win98 VFAT
- Good: faster random access
- Bad: 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 & additional features not needed

inode



- Fast for small files
- Can hold large files

- Typically 15 pointers
  - 12 to direct blocks
  - 1 single indirect
  - 1 doubly indirect
  - 1 triply indirect
- Number of 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 of file? Same – it depends upon block size and pointer size
  - e.g., 4KB block, 4 byte pointer → max size 2 TB

Linux File System: ext3 inode

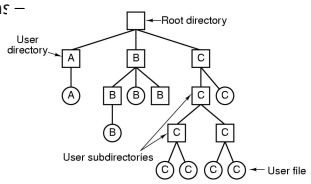
```
// linux/include/linux/ext3_fs.h
#define EXT3_NDIR_BLOCKS 12                     // Direct blocks
#define EXT3_IND_BLOCK   EXT3_NDIR_BLOCKS + 1  // Indirect block index
#define EXT3_DIND_BLOCK  EXT3_IND_BLOCK + 1    // Double-ind. block index
#define EXT3_TIND_BLOCK  EXT3_DIND_BLOCK + 1   // Triple-ind. block index
#define EXT3_N_BLOCKS    EXT3_TIND_BLOCK + 1   // (Last index & total)

struct ext3_inode {
  __u16   i_mode;    // File mode
  __u16   i_uid;     // Low 16 bits of owner Uid
  __u32   i_size;    // Size in bytes
  __u32   i_atime;   // Access time
  __u32   i_ctime;   // Creation time
  __u32   i_mtime;   // Modification time
  __u32   i_dtime;   // Deletion time
  __u16   i_gid;     // Low 16 bits of group Id
  __u16   i_links_count; // Links count
  __u32   i_blocks;  // Blocks count
  ...
  __u32   i_block[EXT3_N_BLOCKS]; // Pointers to blocks
  ...
}
```

Outline

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

Directory Layout - Paths

- Two types of file system paths – Absolute & Relative
- Absolute
  - Full path from root to file
  - e.g., /home/joe/cs4513/hw1.pdf
  - e.g., C:\Users\home\Doc\hw1.pdf
- Relative
  - OS keeps track of process working directory for each process
  - Path relative to current working directory
  - e.g., [working directory = /home/joe]:
    syllabus.docx → /home/joe/syllabus.docx
    cs4513/hw1.pdf → /home/joe/cs4513/hw1.pdf
    ./cs4513/hw1.pdf → /home/joe/cs3513/hw1.pdf
    ../peter/hw1.pdf → /home/peter/hw1.pdf



37

Directory Implementation

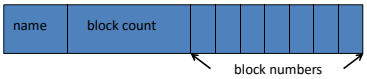
- Just like files ("wait, what?")
  - Have data blocks
  - File descriptor to map which blocks to directory
- But have special bit set so user process cannot modify contents
  - Data in directory is information / links to files
  - Modify only through system call
  - (See `ls.c`)
- Organized for:
  - 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

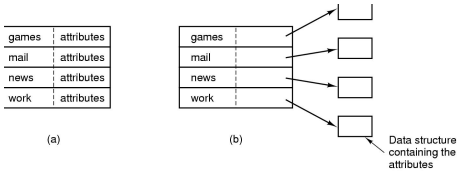
- Before reading file, must be opened
- Directory entry provides information to get blocks
  - Disk location (blocks, address)
- Map ASCII name to *file descriptor*



Where are file attributes (e.g., owner, permissions) stored?

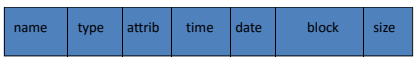
Options for Storing Attributes

a) Directory entry has attributes (Windows)
b) Directory entry refers to file descriptor (e.g., inode), and descriptor has attributes (Unix)



Windows (FAT) Directory

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



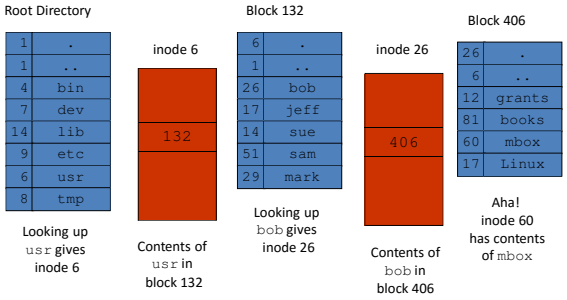
Unix Directory

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

inode	name

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

Unix Directory Example

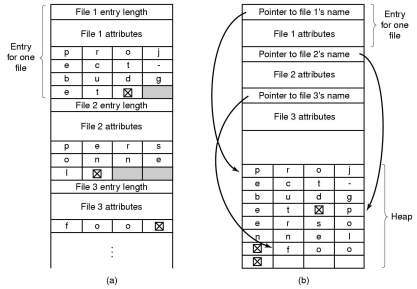


Note: handled by OS in system call to open( ), not user or user code

Length of File Names

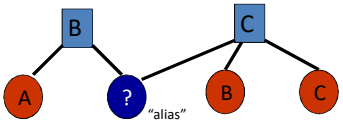
- Each directory entry is name (and maybe attributes) plus descriptor
- How long should file names be?
- If fixed small, will hit limit (users don't like)
- If fixed large, may be wasted space (internal fragmentation)
- Solution → allow variable length names

Handling Long Filenames



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

User Access to Same File in More than One Directory



(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?

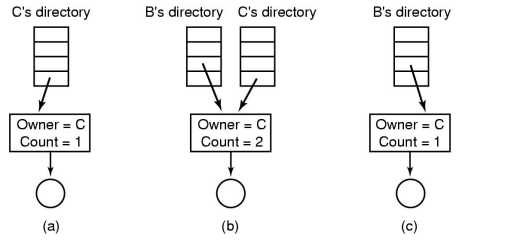
II. Directory entry points to file descriptor?
  – If removed, refers to non-existent file
  – Must keep count, remove only if 0
  – Remember Linux ext3 inode?

```
__u16   i_links_count; // Links count
```

  – *Hard link*
  – Similar if delete file in use

Example: try "`ln`" and "`ls -i`"

Possible Implementation ("hard link")



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

What about hard link across partitions? (example)

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 (example)
- What about soft link across partitions? (example)

Example: try "`ln -s`"

Need for Robust File Systems

- Consider upkeep for removing file
  a. Remove file from directory entry
  b. Return all disk blocks to pool of free disk blocks
  c. Release file descriptor (inode) to pool of free descriptors
- What if system crashes in middle?
  - a) inode becomes orphaned (`lost+found`, 1 per partition)
  - b) blocks free *and* allocated
  - if flip steps, blocks/descriptor free but directory entry exists!
- *Crash consistency problem*

Crash Consistency Problem

- Disk guarantees that single sector writes are atomic
  - But no way to make multi-sector writes atomic
- How to ensure consistency after crash?
  1. Don't bother to ensure consistency
     - Accept that the file system may be inconsistent after crash
     - Run program that fixes file system during bootup
     - File system checker (e.g., *fsck*)
  2. Use transaction log to make multi-writes atomic
     - Log stores history of all writes to disk
     - After crash log "replayed" to finish updates
     - Journaling file system

52

File System Checker – the Good and the Bad

- Advantages of File System Checker
  - Doesn't require file system to do any work to ensure consistency
  - Makes file system implementation simpler
- Disadvantages of File System Checker
  - Complicated to implement *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - Usually **super slow**
    - Scans entire file system multiple times
    - Consider really large disks, like 400 TB RAID array!

53

Journaling File Systems

1. Write intent to do actions (a-c) to log *before* starting
   - Option - read back to verify integrity before continue
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)

Journaling Example

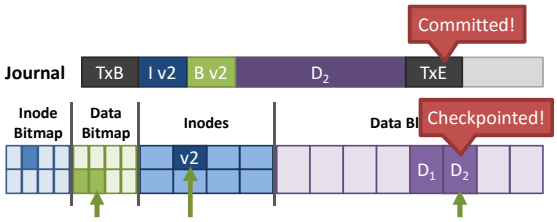
- Assume appending new data block (D_2) to file
  - 3 writes: inode v2, data bitmap v2 (next topic), data D_2
- Before executing writes, first log them

1. Begin new transaction with unique ID=1
2. Write updated meta-data block
3. Write file data block
4. Write end-of-transaction with ID=*1*

55

Commits and Checkpoints

- Transaction committed after all writes to log complete
- After transaction is committed, OS checkpoints update



- Final step: free checkpointed transaction

56

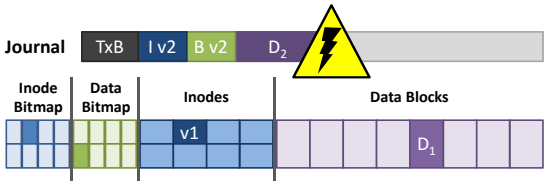
Journal Implementation

- Journals typically implemented as circular buffer
  - Journal is **append-only**
- OS maintains pointers to front and back of transactions in buffer
  - As transactions are freed, back moved up
- Thus, contents of journal are never deleted, just overwritten over time

57

Crash Recovery (1 of 2)

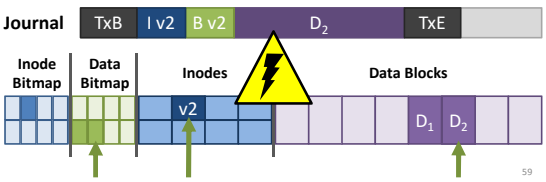
- What if system crashes during logging?
  - If transaction not committed, data lost
  - But, file system remains consistent!



58

Crash Recovery (2 of 2)

- What if system crashes during checkpoint?
  - File system may be inconsistent
  - During reboot, transactions committed but not free replayed in order
  - Thus, no data is lost and consistency restored!



59

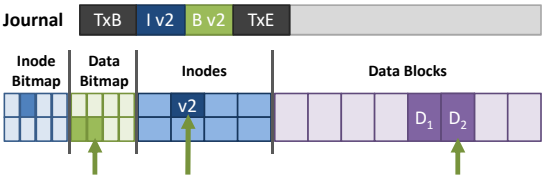
Journaling – The Good and the Bad

- Advantages of journaling
  - Robust, fast file system recovery
    - No need to scan entire journal or file system
  - Relatively straight forward to implement
- Disadvantages of journaling
  - Write traffic to disk doubled
    - Especially file data, which is probably large

60

Meta-Data Journaling

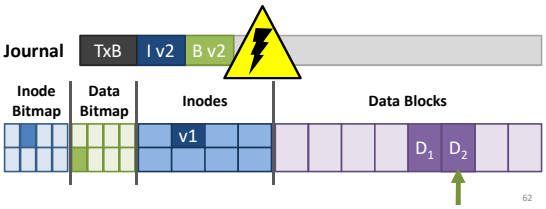
- Most expensive part of data journaling writing file data twice
  - Meta-data small (<1 block), file data is large
- So, only journal *meta-data*



61

Crash Recovery Redux (1 of 2)

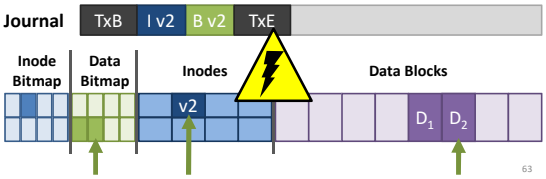
- What if system crashes during logging?
  - If transaction not committed, data lost
  - D_2 will eventually be overwritten
  - File system remains consistent



62

Crash Recovery Redux (2 of 2)

- What if system crashes during checkpoint?
  - File system may be inconsistent
  - During reboot, transactions committed but not free replayed in order
  - Thus, no data lost and consistency is restored



63

Journaling Summary

- Today, most OSes use journaling file systems
  - ext3/ext4 on Linux
  - NTFS on Windows
- Provides crash recovery with relatively low space and performance overhead
- Next-gen OSes likely move to file systems with copy-on-write semantics
  - btrfs and zfs on Linux

64

Outline

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

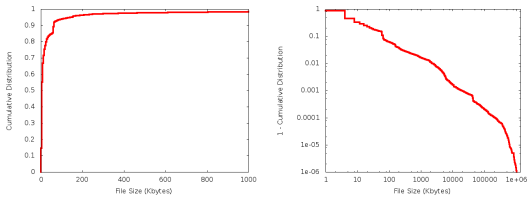
File Sizes in Practice (1 of 2)

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?

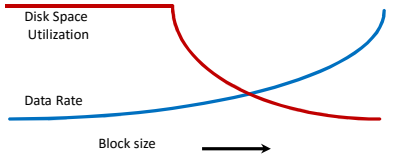
File Sizes in Practice (2 of 2)



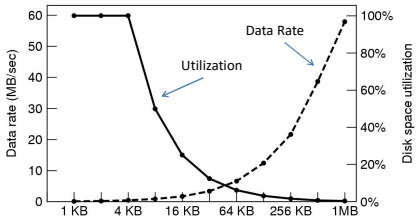
Claypool Office PC
Linux Ubuntu
March 2014

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

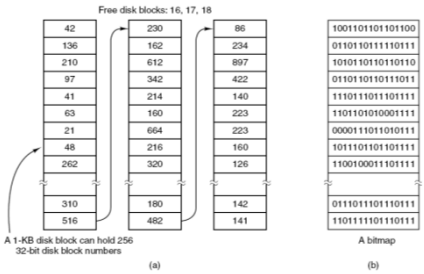


Disk Performance and Efficiency



- Assume 4 KB files.
- At crossover (~64 KB), only 6.6 MB/sec, Efficiency 7% (both bad)
- However → Most disk block sizes not larger than paging system hardware
  - On x86 is 4K → most file systems pick 1KB – 4 KB

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 number points to next block)
   - 500 GB disk has 488 millions disk blocks
     - About 1,900,000 1 KB blocks to track free 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 to track free 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
  - But linked-list blocks are not needed for space (they are free)
  - Only matters for some maintenance (e.g., consistency checking)
- 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 → e.g., some blocks are "more important" than others
  - Block heavily used always in memory
  - Crash w/inode can lead to inconsistent state
  - Some rarely referenced (double indirect block)

Modified LRU

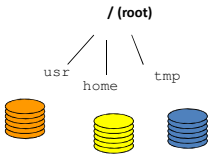
- Is block likely to be needed soon?
  - If no, put at beginning of list
- Is block essential for consistency of file system?
  - Write immediately
- Occasionally write out all
  - `sync`

Outline

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

Partitions

- `mount`, `unmount`
  - pick access point in file-system
  - load *super-block* from disk
- Super-block
  - file system type
  - block size
  - free blocks
  - free file descriptors (inodes)



Mount isn't Just for Bootup

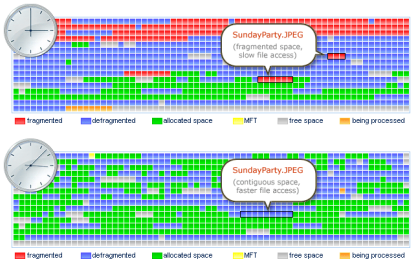
- When plug storage devices into running system, mount executed in background
  - e.g., plugging in USB stick
- What does it mean to "safely eject" device?
  - Flush cached writes to that device
  - Cleanly unmount file system on that device

78

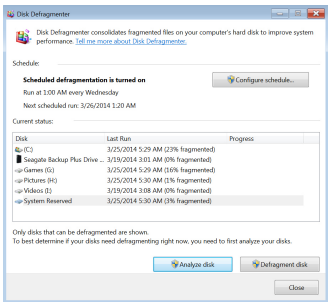
File System Maintenance

- Format:
  - Create file system structure: super block, descriptors (inodes)
  - `format` (Windows), `mke2fs` (Linux)
    (e.g., Windows: "`format /?`" and Linux: "`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)

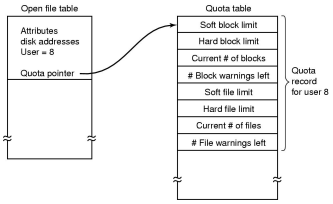


Defragmenting (Example, 2 of 2)



Disk Quotas

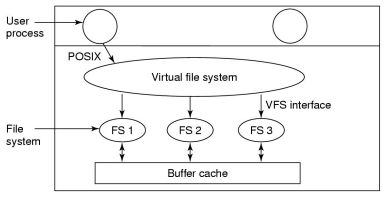
- Table 1: Open file table in memory
  - When file size changed, charged to user
  - User index to table 2
- Table 2: quota record
  - Soft limit checked, exceed allowed w/warning
  - Hard limit never exceeded
- Limit: blocks & file descriptors (inodes)
  - Running out of inodes as bad as running out 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)
- ext4 is "default" for many, most popular (but ext3 still widely in use, e.g., CCC)

Linux File System: extfs

- "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`
  - Larger files (2 TB), Larger file system (32 TB)
  - Fully compatible with `ext2`
  - Adds journaling
- `ext4fs`
  - Larger files (16 TB), Larger file systems (1 EB)
  - Extents (for free space management)
  - Improved perf (multi-block allocation, journal checksum…)

Linux File System: inodes (1 of 2)

- Uses inodes
  - *mode* for file, directory, symbolic link …

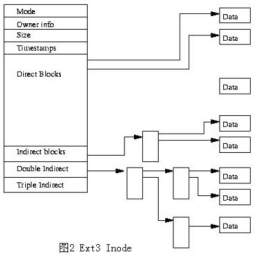


图2 Ext3 Inode

Linux File System: inodes (2 of 2)

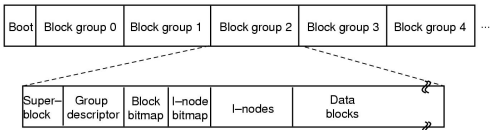
Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation 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)

```
struct ext3_inode {
  __u16   i_mode;    // File mode
  __u16   i_uid;     // Low 16 bits of owner Uid
  __u32   i_size;    // Size in bytes
  __u32   i_atime;   // Access time
  __u32   i_ctime;   // Creation time
  __u32   i_mtime;   // Modification time
  __u32   i_dtime;   // Deletion time
  __u16   i_gid;     // Low 16 bits of group Id
  __u16   i_links_count; // Links count
  __u32   i_blocks;  // Blocks count
  ...
  __u32   i_block[EXT3_N_BLOCKS]; // Pointers to blocks
  ...
}
```

Linux File System: Blocks

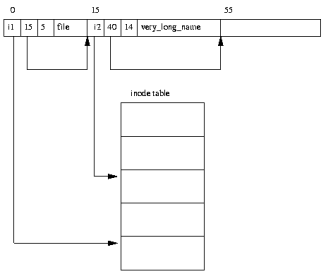
```
$ sudo tune2fs -l /dev/sda1 | grep Block
Block count:              60032256
Block size:               4096
Blocks per group:         32768
```

- 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 inodes
  - Copy of super block

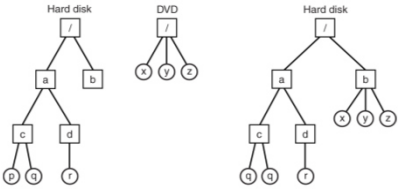


Linux File System: Directories

- Directory just special file with names and inodes



Linux File System: Unified



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

Linux Filesystem: ext3fs & ext4fs

- 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 since blocks grouped

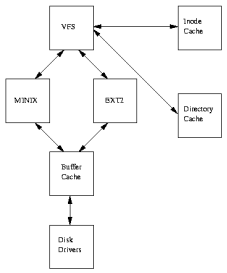
```
yukon% sudo fsck -nvf /dev/sda1
…
942826  inodes used (6.28%)
  1138  non-contiguous files (0.1%)
   821  non-contiguous directories (0.1%)
```

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 New Technology File System: NTFS

- Background: Windows had (has) 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 can have metadata (e.g., thumbnail image)
- Streams fragile, and not always preserved by utilities over network or when copied/backed up

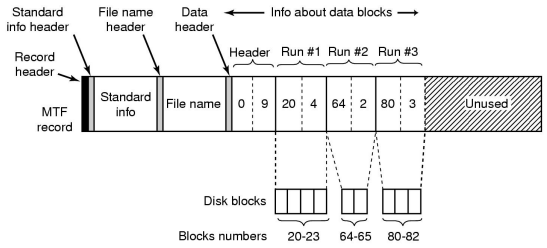
NTFS: Fundamental Concepts

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

NTFS: File System Structure

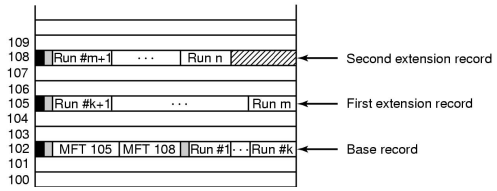
- Basic allocation unit called a *cluster* (block)
  - Sizes from 512 bytes to 64 Kbytes (most 4 KBytes)
  - Referred to by offset from start, 64-bit number
- Each volume has Master File Table (MFT)
  - Sequence of 1 KByte records
  - Bitmap to keep track of which MFT records are free
- Each MFT record
  - 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

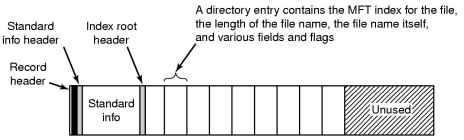
NTFS: Storage Allocation



- 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

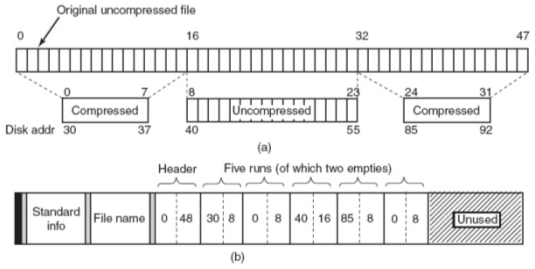


NTFS: Directories

- 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