Operating Systems (234123)

Filesystems

Dan Tsafrir (19/6/2017, 26/6/2017, 3/7/2017)

Partially based on: Chapter #5 in OS notes; Silberschatz; slides by Hagit Attiya, Idit Keidar, Kaustubh Joshi, Michael Swift; VSFS stuff taken from “Operating Systems: Three Easy Pieces” by Remzi & Andrea Arpac-Dusseau (http://pages.cs.wisc.edu/~remzi/OSTEP/)
Why disk drives? (Memory not enough?)

• Disk drive = secondary storage
  – Persistent (non-volatile)
  – Bigger
  – Cheaper (per byte)

• Note
  – CPU (or rather, its MMU = memory management unit), can’t access disk drives directly. It’s all done by the OS

• We’ll typically think on HDDs
  – Hard disk drives
  – Still heavily used in data centers (albeit rapidly replaced by SSDs)
  – Spinning, moving head => notably, for random access
Goals: when using disks, we want...

- **Persistence**
  - Outlive lifetime of a process, tolerate power outage

- **Convenient organization of data**
  - Such that we could easily find & access our data

- **Support ownership**
  - Users, groups; who can access? (read/write)

- **Robustness**
  - In the face of failures

- **Performance**
  - As performant as possible

- **Concurrency support**
  - What happens if we access the information concurrently

- **Storage drive abstraction**
  - Access similar to HDD, SSD, DVD, CD, tape = all are “block devices”
    - Block device = IO is done in block (=several bytes) resolution
Achieve our goals through: filesystems

• **Provides abstractions**
  – To help us organize our information

• **Main abstractions**
  – File
  – Directory (in UNIX), folder (in Windows)
  – Soft links (in UNIX), shortcuts (in Windows)
  – Hard links
  – Standardized by POSIX

• **We will discuss**
  – The filesystem API
  – The abstractions
  – And their implementation
FILES AND DIRECTORIES
File

• A logical unit of information
  – ADT (abstract data type)
  – Has a name
  – Has content
    • Typically a sequence of bytes (we’ll exclusively focus on that)
    • But could be, e.g., a stream of records (database)
  – Has metadata/attributes (creation date, size, …)
  – Can apply operations to it (read, rename, …)

• Persistent (non-volatile)
  – Survives power outage, outlives processes
  – Process can use file, die, then another process can use file
POSIX file concurrency semantics

• Reading concurrently from a file
  – Isn’t a problem
  – For example, if a file is an executable (a program), it can serve as the text of multiple processes simultaneously

• Writing concurrently to a file
  – Local filesystem ensures **sequential consistency**
    • “…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”
      Leslie Lamport, 1979
  – **Writes** to the file must appear as **atomic** operations to any readers that access the file during the write; the reader will see either all or none of any write
File metadata (a.k.a. attributes)

• Examples
  – Size
  – Owner
  – Permissions
    • Readable? Writable? Executable?
    • Who can read? Who can write? Who can execute?
  – Timestamps
    • Creation time
    • Last time content was modified
    • Last time content or metadata were modified
  – Location
    • On disk (recall that a disk is a “block device”)
    • Where do the file’s blocks reside on disk?
  – Type (means various things in various filesystems)
    • E.g., regular file vs. directory
POSIX file descriptors (FDs)

- A successful open<"file name"> of a file returns
  - A nonnegative integer
  - An index to a per-process array called the “file descriptor table”
  - Each entry in the array saves, e.g., the current offset
  - Threads share the array (and hence the offset)

- FD or filename?
  - Some file-related POSIX system calls operate on FDs
    - read, write, fchmod, fchown, fchdir, fstat, ftruncate...
  - Others operate on file names
    - chmod, chown, chdir, stat, truncate
  - Has security implications: FD versions are more secure in some sense
    - Because association of FD to underlying file is immutable
      - Once an FD exists, it will always point to the same file
    - Whereas association between file & its name is mutable
    - Using names might lead to TOCTTOU (time of check to time of use) races

end of lecture
Canonical POSIX file operations

• **Creation**  *(syscalls: creat, open; C: fopen)*
  – Associate with a name; allocate physical space (at least for metadata)

• **Open**  *(open; C: fopen, fdopen)*
  – Load required metadata to allow process to access file

• **Deletion**  *(unlink, rmdir; C: remove)*
  – Remove name/file association & (possibly) release physical content

• **Close**  *(close; C: fclose)*
  – Mark end of access; release associated process resources

• **Rename**  *(rename; -)*
  – Change associated name

• **Stat**  *(stat, lstat, fstat; -)*
  – Get the file’s metadata (timestamps, owner, etc.)

• **Chmod**  *(chmod, fchmod; -)*
  – Change readable, writable, executable properties
Canonical POSIX file operations

- **Chown** (chown, fchown; -)
  - Change ownership (user, group)

- **Seek** (lseek; C: fseek, rewind)
  - Each file is typically associated with a “current offset”
  - Pointing to where the next read or write would occur
  - Lseek allows users to change that offset

- **Read** (read, pread, readv; C: fscanf, fread, fgets)
  - Reads from “current offset”; pread gets offset from caller; v=vector of IOs
  - Need to provide buffer & size

- **Write** (write, pwrite, writev; C: fprintf, fwrite, fputs)
  - Change content of file; pwrite gets the offset explicitly; v=vector of IOs
  - Likewise, need to provide buffer & size
  - If the current offset (or the given offset in the case of pwrite) points to end of file, then file grows
Canonical POSIX file operations

- **Sync** (sync, fsync; <> C: fflush)
  - Recall that
    - All disk I/O goes through OS “page cache”, which caches the disk
    - OS sync-s dirty pages to disk periodically (every few seconds)
  - Use this operation if we want the sync now
  - ‘sync’ is for all the filesystem, and ‘fsync’ is just for a given FD
  - Sync <> fflush; the latter flushes user-space buffers to the kernel

- **Lock** (flock, fcntl; C: flockfile)
  - Advisory lock (processes can ignore it, if they wish)
  - There exists mandatory locking support
    (in Linux and other OSes)
    - E.g., every open implicitly locks; and can’t open more than once
      [Link](https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt)
  - But that’s not POSIX
Reminder: man sections (in Linux)

- To see exact semantics of file (and other) ops, use ‘man’ command
  - Synopsis: man [options] [section] <name>
- Section 1: user (shell) commands
  - man sync  man 1 sync  man 1 read  man 1 printf
- Section 2: system calls
  - man fsync  man 2 fsync  man 2 read  man 2 write
- Section 3: C library functions
  - man fflush  man 3 fflush  man fread  man 3 printf
- Section 4: devices & special files
  - man 4 null (for /dev/null)  man 4 zero (for /dev/zero)
- Section 5: file formats
  - man fstab  man 5 fstab  man crontab  man 5 crontab
- Section 6: games et al.
- Section 7: miscellanea
- Section 8: sys admin & daemons
File types

• **Some systems (not Unix/POSIX) distinguished between**
  – Text & binary

• **Some older systems decided type by name extensions**
  – In DOS, executables have the extensions: com, exe, bat

• **In UNIX (POSIX), types are**
  – Regular file
  – Directory
  – Symbolic link (= shortcut), a.k.a. soft link
  – FIFO (named pipe)
  – Socket
  – Device file
Magic numbers in the UNIX family

• A semi-standard Unix way to tell the type of a file
  – Store a "magic number" inside the file itself
  – Originally, first two 2-byte => only $2^{16}$ => not enough
  – Nowadays, much more complex

• Examples
  – Every GIF file starts with the ASCII strings: GIF87a or GIF89a
  – Every PDF file starts with the ASCII string: %PDF
  – Script files start with a “shebang”, followed by an executable name, which identifies the interpreter of the script; the shell executes the interpreter and feeds the script to it as input (“#” indicates comment for the interpreter, so it ignores this line)
    • #!/usr/bin/perl
      [ Perl stuff here.... ]
    • #!/usr/bin/py
      [Python stuff here]
Magic numbers in the UNIX family

• **Pros**
  – File’s content, rather than metadata, determines what this file is
  – (Metadata like the file's name might be altered independently of the content, potentially erroneously)

• **Cons**
  – Magic logic became fairly complex
  – Somewhat inefficient because
    • Need to check against entire magic database, and
    • Need to read file content rather than just metadata

• **More details**

• **Helpful – the ‘file’ utility**
  – A shell utility that, given a file, identifies its type
  – [http://linux.die.net/man/1/file](http://linux.die.net/man/1/file)
POSIX file protection: ownership & mode

• **Motivation**
  – In a multiuser system, not everyone is allowed to access a given file
  – Even if they’re allowed to “access”, we don’t necessarily want to allow them to perform every conceivable operation on that file

• **For each file, POSIX divides users into 3 classes**
  – “User” (the owner of the file), “group”, and “all” the rest
  – (Users can belong to several groups; see: man 2 getgroups)

• **POSIX associates 3 capabilities with each class**
  – Read, write, and execute

• **Hence, each file is associated with**
  – $3 \times 3 = 9$ class/capabilities => this is called the “file mode”
  – Mode is controlled by the (f)chmod syscall
  – Ownership (user & group) is controlled by the (f)chown syscall
POSIX file protection: ownership & mode

- Stat returns these 9, and ‘ls -l’ displays them
  - 10 “bits” are displayed
  - Leftmost is the “type”
  - Remaining 9 bits are read/write/execute X user/group/all

```
brw-r--r--  1 unixguy staff  64, 64 Jan 27 05:52 block
chrw-r--r--  1 unixguy staff  64, 255 Jan 26 13:57 character
-rw-r--r--  1 unixguy staff  290 Jan 26 14:08 compressed.gz
-rw-r--r--  1 unixguy staff 331836 Jan 26 14:06 data.ppm
drwxrwxr-x  2 unixguy staff  48 Jan 26 11:28 directory
-rwxrwxr-x  1 unixguy staff  29 Jan 26 14:03 executable
prw-r--r--  1 unixguy staff  0 Jan 26 11:50 fifo
lrwxrwxrwx  1 unixguy staff  3 Jan 26 11:44 link -> dir
-rw-rw----  1 unixguy staff  217 Jan 26 14:08 regularfile
```
Access control lists (ACLs)

• OSes can support a much finer, more detailed protection
  – Who can do what

• Most OSes/filesystems support some form of ACLs
  – Many groups/users can be associated with a file
  – Each group/user can be associated with the 3 attributes (r/w/x)
    [http://static.usenix.org/events/usenix03/tech/freenix03/gruenbacher.html](http://static.usenix.org/events/usenix03/tech/freenix03/gruenbacher.html)
  – Or more, finer attributes (“can delete”, “can rename”, etc.)

• Con: not part of POSIX
  – Effort to standardize ACLs abounded in Jan 1998
    • Participating parties couldn’t reach an agreement...
  – Hence, it’s hard to make programs that use ACLs portable
    • (Recall: a program is “portable” across a set of OSes if it works on all of them without having to change its source code; in particular, a program that adheres to POSIX works unchanged in all OSes that comply with POSIX = the UNIX family: Linux, AIX, Solaris, FreeBSD, Mac OS, ...)
Filesystem building blocks

• Abstractions
  – Directories,
  – Hard links, and
  – Symbolic links

• Implementation
  – Inodes, and
  – Dirents
Directories

- A filesystem has a hierarchical (tree) structure
- A recursive structure
- Directories can nest
- Every directory can contain
  - Non-directory ("leaf") files
  - Other, nested directories

```
bin
usr
/etc
/home
/lib

/admin
/stuff
/stud

/jon

/home/admin/jon  /home/admin/steve
```

(OS (234123) - files)
**Absolute & relative file paths**

- **Terminology**
  - File name = file path = path

- **Every process has its own “working directory” (WD)**
  - In the shell
    - Can print it with ‘pwd’ (= print WD)
    - Can move between WDs with ‘cd’ (= change directory)
  - System calls to change WD
    - chdir, fchdir

- **A path is absolute if it starts with “/” (the root directory)**
  - E.g., /users/admin/jon

- **A path is relative (to the WD) otherwise**
  - If WD is “/”, then “/home/admin/jon” = “home/admin/jon”
  - If WD is “/home/” then “/home/admin/jon” = “admin/jon”
  - If WD is “/home/admin/” then “/home/admin/jon” = “jon”
POSIX directory operations

• mkdir(2) – create empty directory
• rmdir(2) – remove empty directory (fails if nonempty)
• Special directory names (relative paths that always exist)
  – Current directory “.
  – Parent directory “..” (root is its own parent)
  – Hidden by default by ‘ls’, as start with a “.”
• Dir content traversal – opendir(3), readdir(3), closedir(3)
  –
    ```c
    struct dirent *de;
    DIR *p = opendir( "." );
    while( (de = readdir(p)) != NULL )
      printf("%s\n", de->d_name);
    closedir( p );
    
    – What do we need to do to make it recursive?
```
Hard links – intro

• **File != file name**
  – They are *not* the same thing
  – In fact, the name is not even part of the file’s metadata
  – A file can have many names, which appear in unrelated places in the filesystem hierarchy
  – Creating another name => creating another “hard link”

• **System calls**
  – link( srcpath, dstpath )
  – unlink( path )

• **Shell**
  – ln <srcpath> <dstpath>
  – rm <path>
Hard links – example

<0>dan@csa:~$ echo "hello" > f1
<0>dan@csa:~$ cat f1
hello
<0>dan@csa:~$ ls -l f1
-rw-r--r-- 1 dan 6 Jun 10 06:48 f1
<0>dan@csa:~$ ln f1 tmp/f2;
<0>dan@csa:~$ cat tmp/f2;
hello
<0>dan@csa:~$ ls -l f1;
-rw-r--r-- 2 dan 6 Jun 10 06:48 f1
<0>dan@csa:~$ echo "goodbye" > f1;
<0>dan@csa:~$ cat tmp/f2;
goodbye

# “1” = how many links to the file
# ln <src> <dst> creates the link
# f1 & f2 are links to same file
# so they have the same content
# ‘ls -l’ reveals how many links:
# 2 links
# override content of f1
# content of f2 also changes
Hard links – when is a file deleted?

• Every file has a “reference count” associated with it
  – link() ⇔ ref_count++
  – unlink() ⇔ ref_count--

• if( ref_count == 0 )
  – The file has no more names
  – It isn’t pointed to from any node within the file hierarchy
  – So it can finally be deleted

• What if an open file is deleted? (its ref_count==0)
  – Can we still access the file through the open FD(s)?
    • Yes
  – If >=1 processes have the file open when the last link is removed
    • The link shall be removed before unlink() returns
    • But the removal of the file contents shall be postponed until all references (file descriptors) to the file are close()-ed
Hard links – what they do to the hierarchy

• **Before hard links**
  – Tree graph

• **After**
  – Any graph

• **Hard links to directories?**
  – Hard links to directories are usually disallowed & unsupported by the filesystem (though POSIX does allow directory hard links)
  – => Acyclic graph (no circles)
  – What’s the benefit?

• **Notable exception**
  – HFS+, the Mac OS filesystem (since ~2007)
  – For time-machine (allows for cheap filesystem snapshots)
  – [http://appleinsider.com/articles/07/10/12/road_to_mac_os_x_leopard_time_machine/page/3](http://appleinsider.com/articles/07/10/12/road_to_mac_os_x_leopard_time_machine/page/3)
Directory hard links

• A noted, most filesystems don’t support directory hard links
  – HFS+ is an exception

• Still, all filesystems that adhere to POSIX provide at least some support to directory hard links
  – Due to the special directory names “.” and “..”
  – What’s the minimum number of hard links for directory?
    • 2 (due to “.”)
  – What’s the maximum?
    • Depends on how many subdirectories nest in it (due to “..”)

OS (234123) - files
Symbolic links (soft links)

• Unlike hard links
  – Which point to the actual underlying file object

• Symlinks ("shortcuts" in Windows terms)
  – Point to a name of a “target” file (their content is typically this name)
  – They’re not counted in the file’s ref count
  – They can be “broken” / “dangling” (point to a nonexistent path)
  – They can refer to a directory (unlike hard links in most cases)
  – They can refer to files outside of the filesystem / mount point
    (whereas hard links must point to files within the same filesystem)

• When applying a system call to a symlink
  – The system call would seamlessly applied to the target file
  – For example, open(), stat(), chmod(), chown(), ...
Symbolic links (soft links)

• Exception: the unlink(2) syscall (and thus the ‘rm’ utility)
  – Will remove the symlink, not the target file

• Symlink-specific system calls
  – symlink(2) creates a symbolic link, linking it to a specified target file
  – readlink(2) read “content” of symlink, which is the target file
    • Q: what does read(2) do when applied to a symlink?
    • A: it’s a trick question; read(2) operates on a file descriptor, which is returned by open(2), which, as noted, applies to the target file

• Shell
  – ln –s <srcpath> <dstpath>
Symbolic links – example

<0>dan@csa:~$ echo hey > f1
<0>dan@csa:~$ ln -s f1 f2; # f2 is a symlink to f1
<0>dan@csa:~$ ls -l f2
lrwxrwxrwx 1 dan 2 Jun 10 07:56 f2 -> f1 # notice arrow & perms

<0>dan@csa:~$ cat f2; # content of f1
hey

<0>dan@csa:~$ rm -f f1; # f1 no longer exists
<0>dan@csa:~$ cat f2; # so the ‘cat’ fails
cat: f2: No such file or directory
Implementation – inode

• The OS data structure that represents the file
  – Each file has its own (single) inode (= short for “index node”)
  – You can think of the inode as “the file” or as “the file object”
  – Internally, file names point to inodes

• The inode contains all the metadata of the file
  – Timestamps, owner, permissions, ... all that
  – Pointers to the actual physical blocks, on disk
inodes & *stat

- stat(2) & fstat(2) retrieve metadata held in inode
  - POSIX promises that at least the following fields are found in the stat structure and have meaningful values
  
  1. `st_dev` ID of device containing file
  2. `st_ino` inode number, unique per `st_dev`
  3. `st_mode` determines type of file, among other things
  4. `st_nlink` number of hard links to the file
  5. `st_size` file size in bytes
  6. `st_uid` user ID of owner
  7. `st_gid` group ID of owner
  8. `st_ctime` last time metadata (=inode) or data changed
  9. `st_mtime` last time data changed
  10. `st_atime` last time metadata (=inode) or data accessed
  11. `st_blksize` block size of this file object
  12. `st_blocks` number of blocks allocated for this file object

- Why do we need `st_size` as well as `st_blksize` & `st_blocks`?
inodes & *stat

- `lstat(2)`
  - Exactly the same as `stat(2)` if applied to a hard link
  - But if applied to a symlink, would return the information of this symlink *(not to the target of the symlink)*
  - In this case, POSIX says that the only fields within the stat structure that you can portably use are:
    - `st_mode` which will specify that the file is a symlink
    - `st_size` symlink content length (= length of target file)
  - The value of the rest of the fields could be valid, but it is not specified by POSIX
  - Notably, it is not specified if a symlink has a corresponding inode
    - Will be discussed shortly
Implementation – directory

- A simple flat file comprised of directory entries
  - For example, it could be a sequence of
    ```c
    struct dirent {
        ino_t d_ino; /* POSIX: inode number */
        off_t d_off; /* offset to next dirent */
        short d_reclen; /* length of this record */
        char d_type; /* type of file */
        char d_name[NAME_MAX+1]; /* POSIX: null-term fname */
        /* must we always use NAME_MAX+1 chars? */
    };
    ```

- Importantly, note that
  - The name of a file is **not** saved in the inode
    (there can be very many names associated with a file)
  - Rather, it is stored in the directory file as a simple string, in d_name
  - If the file is a symbolic link
    - The target file can be retrieved from the contents of the symlink
Simple example

<table>
<thead>
<tr>
<th>d_ino</th>
<th>d_off</th>
<th>d_reclen</th>
<th>d_type</th>
<th>d_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>n1</td>
<td>n1</td>
<td>&lt;reg&gt;</td>
<td>first.txt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d_ino</th>
<th>d_off</th>
<th>d_reclen</th>
<th>d_type</th>
<th>d_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>89</td>
<td>n1+n2</td>
<td>n2</td>
<td>&lt;reg&gt;</td>
<td>second.something</td>
</tr>
</tbody>
</table>

$ ls
first.txt
second.something
README

\[
\sum_{k=1}^{3} n_k = 1002
\]

n3

<table>
<thead>
<tr>
<th>d_ino</th>
<th>d_off</th>
<th>d_reclen</th>
<th>d_type</th>
<th>d_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>\sum_{k=1}^{3} n_k</td>
<td>n3</td>
<td>&lt;reg&gt;</td>
<td>README</td>
</tr>
</tbody>
</table>
Implementation – symlink

• POSIX doesn’t specify whether a symlink should have an inode
  – In principle, a symlink could be implemented as a directory entry and nothing else

• But filesystems often define an inode per symlink
  – In that case, however, if the name of the target file is small enough
  – Then the target file name is saved in the inode itself, rather than in a data block (the target file is then said to be “inlined”)
    • Q: What’s the benefit?
Path resolution process

• Resolving a path
  – Get a file path
  – Return, say, the inode that corresponds to the path

• All system calls that get a file name as an argument
  – Need to resolve the path

• To this end, all these system calls use the same algorithm
  – Which is oftentimes called “namei” (internally, within the kernel)

• The namei algorithm consists of at least n steps
  – Atomic = doesn’t contain a slash (“/”) = a name component
  – n = number of atomic name components that comprise the file path
  – Including the components that comprise the symlinks along the path
  – Recursively speaking
Path resolution process

- **Let n be the (recursive) number of atoms comprising a path**
- **Assume the file path is /x/y/z**
  - n=4 if x and y are directories and z is a regular file (4 because of “/”)
  - n=5 if x and y are dirs and z is a symlink (relative path) to a regular file w
    - /x/y/z => /z/y/w (that is, w resides under /x/y/)
    - the target of the symlink z is a relative path (“w”)
  - n=8 if x is a directory, y is a symlink to /r/p/q/ (such that r, p, and q are directories), and z is a regular file
    - In this case, /x/y/z \(\equiv\) /r/p/q/z
    - And n=8 because the path resolution process traverses the following components: (1) “/” (2) “x” (3) “y” (4) “/” (5) “r” (6) “p” (7) “q” (8) “z”
- **Notice that**
  - The permissions of the calling user are checked against each non-symlink component along the path: in our example, the user must be able to search all directories that lead to “z” (in addition to “z” itself)
  - “z” (= the last non-symlink component) is denoted **the terminal point** of the path
Path resolution process

- To illustrate, here's a simplistic pseudo code version of a user-mode component-by-component open

```c
#define SYS ( call ) if( (call) == -1 ) return -1

int my_open( char * fname ) {
    if( fname is absolute ) chdir( "/") + make fname relative
    foreach atom in fname do // atoms of "x/y" are "x" and "y"
        if( is symlink ) SYS( fd = my_open( atom's target ) )
        else SYS( fd = open( atom ) )
        if( not last ) SYS( fchdir( fd ) + close( fd) )
        else break

    return fd
}
```
Path resolution process

• Q: n is a lower bound on the complexity of name; why?
  A: Because finding each individual directory component along the path may also be a linear process
  – Recall the procedure to print all files in a directory:
    ```c
    struct dirent *de;
    DIR *p = opendir( "." );
    while( (de = readdir(p)) != NULL )
      printf("%s
", de->d_name);
    closedir( p );
    ```

• We can speed up the process within the kernel by caching directory entries
  – Learn(ed) this in the Tirgul

• All the details of the (Linux) path resolution process:
HARD DISK DRIVES
HDDs
HDDs

• **Latency components**
  – Seek time
  – Rotational latency

• **Seek time**
  – Time of head to travel to track

• **Rotational latency**
  – Delay caused by waiting for disk rotation to bring required sector under head

• **Performance**
  – ~5 milliseconds latency
  – Faster sequential accesses (~100MB/s), slower random access (~1MB)
Example – enterprise level (mid 2013)

• **Seagate Cheetah 15K.7 SAS; by spec:**
  - “Highest reliability rating in industry”
  - SAS (Serial Attached SCSI; rotates faster & is pricier than SATA’s 5.4K/7.2K RPM HDDs)
  - Price: $260 (@ amazon.com)
  - Spindle speed 15K (RPM = rounds per minute)
  - Capacity: 600 GB
  - Cache: 16MB (DRAM)
  - Form factor: 3.5 inch (laptops have 2.5” or even 1.8” HDDs)
  - Throughput: 122 MB/s (min) – 204 MB/s (max)
  - Avg rotational lat.: 2.0 ms
  - “Typical avg seek time” 3.4 ms (read) 3.9 ms (write)
    • Typical single track 0.2 ms (read) 0.44 ms (write)
    • Typical full stroke 6.6 ms (read) 7.4 ms (write)
    - Full stroke = time head moves from outer to inner portion of the disk (full length of the drive head's motion)
  - Error rate < 1 in $10^{21}$
Example – consumer grade (mid 2014)

- Western Digital 4 TB Green SATA III 5400 RPM
  64 MB Cache Bulk/OEM Desktop Hard Drive WD40EZRX

<table>
<thead>
<tr>
<th></th>
<th>500GB</th>
<th>1TB</th>
<th>1.5TB</th>
<th>2TB</th>
<th>3TB</th>
<th>4TB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>$54</td>
<td>$58</td>
<td>$75</td>
<td>$76</td>
<td>$110</td>
<td>$147</td>
</tr>
</tbody>
</table>

- Interface: SATA 6 Gb/s
- Form factor: 3.5”
- Max throughput: 150 MB/s
- (No data in spec about latency)
- Non-recoverable read errors per bits read < 1 in $10^{14}$
(Anecdote: why “3.5 inch form-factor”?)

• 3.5” = 8.89 cm

• But standard dimensions of a 3.5 inch disk are
  
  – Length: 5.79” ≈ 14.7 cm
  – Height: 1.03” ≈ 2.61 cm
  – Width: 4.00” = 10.16 cm
(Anecdote: why “3.5 inch form-factor”?)

• 3.5” = 8.89 cm

• But standard dimensions of a 3.5 inch disk are
  – Length: 5.79” ≈ 14.7 cm
  – Height: 1.03” ≈ 2.61 cm
  – Width: 4.00” = 10.16 cm

• Historical reason
  – Capable of holding a platter that resides within a 3.5” (≈ 90mm width) floppy disk drive
FILESYSTEM LAYOUT ON DISK
Which blocks on disk constitute a file?

• We know that HDDs are “block devices”
  – They are accessed in resolution of sectors
    • (Typically 512 bytes per sector)
    • (Sequential access much faster than random access)
  – Sectors are identified by their LBA (logical block address)
    • 0,1,2, ..., N-1
    • Adjacent LBAs imply physical contiguity on disk
      => Sequential access

• However
  – Which sectors together constitute a given file?
  – And how do we find the right sector if we want to access the file at a particular offset?
Need to understand...

- **Data structures**
  - On-disk structures that organize the data/metadata
    - Array?, linked list?, tree?, ...

- **Access methods**
  - How to map system calls to the above structure
    - open(), read(), write(), ...

- **Unlike virtual memory, the above decisions are done purely in SW**
  - => Lots of flexibility
  - => Lots of filesystems,
  - Literally, from AFS (Andrew FS) to ZFS (Sun’s Zettabyte FS)

- **Let’s use VSFS = Very Simple File System**
  - A toy example to help us understand some of the concepts
But before, let’s rule out contiguous block allocation

• **Pros**
  – Simple file representation
  – Perfect sequential access when reading a file sequentially
  – Minimal seek for random access
    • Just one seek for each contiguous access

• **Cons**
  – External fragmentation
    • Deletions might leave holes of unusable sizes
  – Problem: how to append to a growing file?
    • Need to guess size in advance
VSFS – layout on disk

• Assume we have a really small disk of \( N = 64 \) blocks (0,1,\ldots 63)
  – Block size = 4 KB (=> overall size of \( 4 \times 64 = 256 \) KB)

  \[
  \begin{array}{cccccccccccccccccccccccccccc}
  \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} & \text{BBBBBBBBB} \\
  \end{array}
  \]

• VSFS layout

  \[
  \begin{array}{cccccccccccccccccccccccccccc}
  \text{-----------------------} & \text{The Data Region} & \text{-----------------------} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|} & \text{|}
  \end{array}
  \]

  \[
  \begin{array}{cccccccccccccccccccccccccccc}
  \text{SidIIIII} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD} & \text{DDDDDDDD}
  \end{array}
  \]

  \[
  \begin{array}{cccccccccccccccccccccccccccc}
  \end{array}
  \]

  – \( D \) = data blocks (56 out of the 64); all the rest is metadata:
  – \( I \) = inode table (on-disk array, 5 blocks our of the 8)
    • Assume 128 B inode => At most: \( 5 \times 4\text{KB} / 128 = 5 \times 32 = 160 \) files

  – \( d \) = bitmap of allocated data blocks (at most 56 bits are on)
    \( i \) = bitmap of allocated inode (at most 160 bits are on)
    • (An entire block for ‘d’ and ‘i’ is too big for VSFS, as we only need 56 and 160 bits, respectively; we do it for simplicity)
  – \( S \) = “superblock” = filesystem information
Filesystem superblock

• Contains information about the particular filesystem
  – How many inodes? (VSFS: 160)
  – How many data blocks? (VSFS: 56)
  – Start of inode table (VSFS: block 3)
  – Magic number = identifies the filesystem (VSFS, there are many others)
  – The inode of the root directory
  – [... and everything else needed to work with the filesystem]

• Location of the superblock (of any FS) must be well-know
  – For when “mounting” the system (next slide)
Partitions, fdisk, mkfs, mount

- A disk can be subdivided into several “partitions” using ‘fdisk’
  - Partition = contiguous disjoint part of the disk that can host a filesystem

- Disks are typically named
  - /dev/sda, /dev/sdb, ...

- Disk partitions are typically named
  - /dev/sda1, /dev/sda2, ...
  - Listing partitions on disk /dev/sda:

  ```
  $ sudo fdisk -l /dev/sda
  
  Disk /dev/sda: 1979.1 GB, 1979120025600 bytes
  255 heads, 63 sectors/track, 240614 cylinders, total 3865468800 sectors
  Units = sectors of 1 * 512 = 512 bytes
  
  Device Boot Start   End     Blocks   Id  System
  /dev/sda1 *   2048 499711   248832   83  Linux
  /dev/sda2   501758 3865466879 1932482561   5 Extended
  /dev/sda5   501760 3865466879 1932482560   8e Linux LVM
  ```
Partitions, fdisk, mkfs, mount

• Every filesystem implements a mkfs ("make FS") utility
  – Creates an empty filesystem of that type on a given partition
  – (What’s the implementation of mkfs for VSFS?)

• After an FS is created, need to “mount” it to make it accessible
  – mount = תִּפֵּס, עָלָה, עָלָה; רָכַב; עָלָה; תִּכְנֵּן, אִרְגֵּן; הֶעְלָה, הַרְכִּי
  – For example
    • mount -t ext3 /dev/sda1 /home/users
      Mounts the filesystem that resides in the partition /dev/sda1 onto the directory /home/users; the sda1 filesystem is of the type ext3
  – From now on
    • ls /home/users
      will list the content of the root directory of the said ext3 filesystem
  – The ‘mount’ shell utility utilizes the ‘mount’ system call
  – Obviously, mount makes use of the superblock of the filesystem
  – Reverse of mount: umount
Partitions, fdisk, mkfs, mount

- **Running mount** in shell with no args prints all mounts, e.g.

```
proc  on  /proc  type  proc  (rw)
sysfs on  /sys  type  sysfs  (rw)
tmpfs on  /dev/shm  type  tmpfs  (rw)
/dev/sda1 on  /  type  ext3  (rw)
/dev/sda2 on  /usr  type  ext3  (rw)
/dev/sda3 on  /var  type  ext3  (rw)
/dev/sda6 on  /tmp  type  ext3  (rw)
```

- **For more details see**
  - “Learn Linux, 101: Hard disk layout”
  - “Learn Linux, 101: Create partitions and filesystems”
VSFS – finding an inode

• VSFS layout

|---------------------| The Data Region ---------------------|
| SidIII  DDDDDDDD  DDDDDDDD  DDDDDDDD  DDDDDDDD  DDDDDDDD  DDDDDDDD  DDDDDDDD |
| 01234567 8 15 16 23 24 31 32 39 40 47 48 55 56 63 |

• Given an inumber (=index of inode in inode table), find inode:

  – sector = (inodeStartAddr + (inumber x sizeof(inode_t)) / sectorSize

    |---- in bytes ---|                    |--- in bytes --|

• Once found, the inode tells us all the file’s metadata

  – Number of allocated blocks, protection, times, …, and

  – Pointer(s) to where data is stored

• Pointers could be, for example,

  1. Direct

    • Point directly to all data blocks => |file| ≤ pointerNum x blockSize

  2. Indirect (multi-level)

  3. Linked list
Multi-level index (in the classic Unix FS)

- Assume each block pointer is 4 bytes long, and each block is 4KB
- 12 pointers in inode point directly to data blocks
  - Consumes 4x12=48 B
- Single-indirect pointer points to a block completely comprised of pointers to data blocks
  - 1024 data blocks
  - \((12+1024) \times 4\text{KB} = 4144\text{ KB}\)
  - \(~4\text{MB}\)
- Double-indirect adds
  - \(1024^2 = 4\text{KB} \times 2^{20} = 4\text{ GB}\)
- Triple-indirect adds
  - \(1024^3 = 4\text{KB} \times 2^{30} = 4\text{ TB}\)
Multi-level index (in the classic Unix FS)

- The indirect blocks are blocks like “ordinary” data blocks
- They do not reside in the inode
- The filesystem allocates them exactly as it allocates “ordinary” blocks
- (They are not included in the “size” of the file)
Why is hierarchy imbalanced like so?

• Data from
  – 12 million files from over 1000 Unix filesystems

• Y axis
  – CDF = cumulative distribution function = how many items [in %] are associated with the corresponding X value, or are smaller

• ‘Files’ curve
  – Percent of existing files whose size is $\leq x$

• ‘Bytes’ curve
  – Percent of stored bytes that belong to a file whose size is $\leq x$
Why is hierarchy imbalanced like so?

- Data from
  - 12 million files from over 1000 Unix filesystems

- Observations
  - ~20% of the files <= 512 B
  - ~90% of the files <= 16 KB
  - ~10% of stored bytes belong to files that are up to 16 KB

- Vertical arrows
  - Middle = “joint ratio” (= 10/90 in this case)
    - Determined by where sum of the two curves = 1
    - 90% of the files (are so small they) account for 10% of all stored bytes
    - 10% of the files (are so big they) account for 90% of all stored bytes
  - Left = half of the files are so small they account for only ~2% of bytes
  - Right = half of stored bytes belong to (only) ~0.3% of the files
• This phenomenon is commonly observed in computer systems
  – A typical file is small, but a typical byte belongs to a large file
  – A typical job/process is short, but a typical cycle belongs to a long job/process
  – A typical youtube clip is download only a few time, but a typical download is associated with a clip that has been downloaded very many times

• The phenomenon has been named “mass-count disparity”
  – A small number of items account for the majority of mass, whereas all small items together only account for negligible mass
  – (Disparity = שונת, שָׁוְיָה, נִבְדָלוּת; in our example: mass=bytes & count=files)
Extents

• **Motivation**
  - Hierarchical block structure could mess up sequential access
  - Dynamic append/truncate/delete I/O ops might hinder sequential access (= every leaf block might reside someplace else)

• **Solution: extent-based allocation**
  - Extent = contiguous area comprised of variable number of blocks
  -_inode saves a list of (pointer, size) pairs

<table>
<thead>
<tr>
<th></th>
<th>foo.c</th>
<th>bar.c</th>
<th>core666.</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.c</td>
<td>(0,3)</td>
<td>(6,2)</td>
<td>(16,2)</td>
</tr>
<tr>
<td>bar.c</td>
<td>(3,1)</td>
<td>(12,4)</td>
<td></td>
</tr>
<tr>
<td>core666.</td>
<td>(8,3)</td>
<td>(18,1)</td>
<td></td>
</tr>
</tbody>
</table>
Extents

• **Pros**
  
  – Promotes sequentiality (less pointer chasing, simplified hierarchy)
  
  – Efficient seek (one I/O op when extent found)

• **Cons**
  
  – Harder management: like memory management in the face of dynamic deletes/appends
  
  – Need to maintain holes with usable sizes
  
  – Need to predict size of file
    
    • Over-prediction => internal fragmentation
    
    • Under-prediction
      => allocate another extent
      => more seeks when searching for a given offset

---

### Catalog

<table>
<thead>
<tr>
<th>File</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.c</td>
<td>(0,3)</td>
<td>(6,2)</td>
</tr>
<tr>
<td>bar.c</td>
<td>(3,1)</td>
<td>(12,4)</td>
</tr>
<tr>
<td>core666.</td>
<td>(8,3)</td>
<td>(18,1)</td>
</tr>
</tbody>
</table>
Extents

• **In ext4**
  – Current default FS for Linux
  – With 4KB block, size of a single extent can be between: 4KB ... 128MB (32K blocks)
  – Up to 4 extents stored in inode
  – Hierarchical if more than 4
    • Indexed by an “Htree” (similar to Btree but with constant depth of 1 or 2)
    • Allows for efficient seek/offset search

• **Supported by most current filesystems**
  – ntfs, ext4, hfs+, btrfs, reiser4, xfs, ...

• **Comments**
  – ext4 = extended filesystem 4 (name unrelated to “extent”)
  – ex4 is backward compatible: ext2 and ex3 can be mounted as ext4
Linked list block allocation

• **A file could be a linked list**
  – Such that every block points to the next
  – Each directory entry points to 1st and last file blocks of file
    • (Need last for quick append)

• **Severe performance overhead**
  – Because seek operations require linear number of ops
  – Each op is a random disk read

• **Optimization**
  – Move pointers out of blocks
  – Put them in external table that can be cached in memory
  – Rings a bell?
**FAT filesystem**

- **FAT = file allocation table**
  - The DOS filesystem, still lives
  - Named after its main data structure

<table>
<thead>
<tr>
<th>name</th>
<th>start block</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>... 2</td>
</tr>
<tr>
<td>bar</td>
<td>... 4</td>
</tr>
</tbody>
</table>

**the FAT**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**before**

**after**
FAT

• **Layout**
  – One table for all files
  – One table entry for every disk block
  – -1 (0xFFFF) marks “the end”
  – Directory file: content of each entry points to start of file

• **FAT (table) is copied to (cached in) memory**
  – But of course must also be saved on disk (why?)
  – Solves random access problem of file pointers

• **Pros**
  – Simple to implement:
    • File append, free block allocation management, easy allocation
  – No external fragmentation
  – Fast random access since table is in memory (for block pointers, not necessarily for the blocks themselves)
FAT

- Cons
  - File contiguity can be lost (this is why extents were invented)
  - Table can be huge
    - 32 GB (disk) / 4KB (block) = $2^5 \times 2^{30} / 2^{12} = 2^{23} = 8$ M
    - Assuming 4B pointer, this means 32 MB table
    - Exercise: what’s the size of FAT for a 4TB disk?
Redundant Array of Independent Disks

RAID
Motivation

• Problem #1
  – Sometimes disks fail

• Problem #2
  – Disk transfer rates can be much slower than CPU performance

• Solution – part #1: striping
  – We can use multiple disks to improve performance
  – By striping files across multiple disks (placing parts of each file on a different disk), we can use parallel I/O to improve the throughput

• But striping makes problem #1 even worse
  – Reduces reliability: 100 disks have 1/100th the MTBF (=mean time between failures)

• Solution – part #2: redundancy
  – Add redundant data to the disks (in addition to striping)
RAID (Redundant Array of Inexpensive Disks)

• **Individual disks are small and relatively cheap**
  – So it’s easy to put lots of disks (10s, 100s) in one box/rack for
    • Increased storage, performance, and reliability

• **Data plus some redundant information are striped across the disks in some way**
  – How striping is done is key to performance & reliability
  – We decide on how to do it depending on the level of redundancy and performance required
  – Called "RAID levels"; standard RAID levels:
    • RAID-0, RAID-1, RAID-2, RAID-3, RAID-4, RAID-5, RAID-6

• **Proposed circa 1987**
  – By David Patterson, Garth Gibson, and Randy Katz (@ Berkeley)
RAID-0

• **Non-redundant disk array**
  – Files striped evenly across $N \geq 2$ disks
  – Done in **block** resolution: in principle, technically, a block can be as small as 1 byte; but is a multiple of drive’s sector size in most standard RAID levels taught today (0,1,4,5,6)

• **Pros**
  – High read/write throughput
    • Potentially $N$ times faster
    • Best write throughput (relative to other RAID levels)
  – Faster aggregated seek time, that is
    • The seek time of one disk remains the same
    • But if we have $N$ independent random seeks, potentially, we can do it $N$ times faster

• **Con**
  – Any disk failure results in data loss
RAID-1

- **Mirrored disks**
  - Files are striped across half the disks
  - Data written to 2 places: data disk & mirror disk

- **Pros**
  - On failure, can immediately use surviving disk
  - Read performance: similar to RAID 0 (Why? How about write performance?)

- **Cons**
  - Wastes half the capacity

- **Related: nowadays, in the cloud / data center**
  - Replicating storage systems are prevalent
  - Often 3 replicas of each data “chunk” (chunk sizes are usually in MBs, to amortize seeks)
  - Replicas get “randomly” assigned to disks to balance the load
  - That is, not using classical RAID 1 disk pairs; rather, every disk is striped across many other disk
Reminder: parity

- **Modulo-2 computations**
  - 0 = even number of 1s
  - 1 = odd number of 1s

- **When one sequence bit is lost**
  - Can reconstruct it using the parity

- **In storage systems**
  - Parity computed on a block level

<table>
<thead>
<tr>
<th>#</th>
<th>bit sequence</th>
<th>parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
RAID-4

- **Use parity disks**
  - Each block (= multiple of sector) on the parity disk is a parity function (=xor) of the corresponding blocks on all the N-1 other disks

- **Read ops**
  - Access data disks only

- **Write ops**
  - Access data disks & parity disk
  - Update parity

- **Upon failure**
  - Read remaining disks plus parity disk to compute missing data

- **Pros**
  - In terms of capacity, less wasteful than RAID-1 (wastes only 1/N)
  - Read performance similar to RAID-0 (How about write performance? Next slide...)

![RAID 4 Diagram](image-url)
RAID “small write problem”

- Applicable to all RAIDs that utilize “erasure codes” (like parity)
  - Write is a slower op
  - As it requires to also update the parity
- When performing a small write (e.g., a few bytes), we need to
  1. Read the block of the data
  2. Read the block of the parity
  3. Compute difference between old and new data
  4. Write-update parity block accordingly
  5. Write-update data block
- We thus need to perform four I/O ops
  - Instead of one!
- Why is this just a *small* write problem?
  - When writing A1, A2, and A3, we don’t need to read anything
  - Instead we can compute their parity and write it to Ap
RAID-5

- Similar to RAID-4, but uses **block interleaved distributed parity**
  - Distribute parity info across all disks
  - A “stripe” is a concurrent series of blocks \((A1B1C1, A2B2C2, \ldots)\)
  - The parity of each stripe resides on a difference disk
- **Pros & Cons**
  - Like RAID-4, but better because it eliminates the hot spot disk
  - E.g., when performing two small writes in RAID-4
    - They must be serialized
  - Not necessarily so in RAID-5
    - \(\Rightarrow\) better performance
RAID-6

- Extends RAID-5 by adding an additional “parity” block
  - Ap and Aq must be independent of each other, algebraically speaking
- Pros & cons relative to RAID-5
  - Can withstand 2 disk failures (2 equations, can find 2 variables)
  - But wastes 2/N rather than 1/N of the capacity
RAID-2 & RAID-3

• Like RAID-4
  – But in bit and byte resolution, respectively

• Requires “lockstep”
  – All disks spin synchronously => almost never used
Notation: RAID n+k

- n blocks of regular data
- k blocks that provide redundancy
- Example:
  - Previous slide: 3+2
  - If we have 12 disks and we’d like to be able to tolerate 3 failures: 9+3
Erasure vs. replication in a data center

• **Erasure (n+k)**
  - Pros: optimizing capacity, “wasting” only k/(k+n) on redundancy
    - RAID-5: 1/(1+n); RAID-6: 2/(2+n); typically < 1/2 (as k < n)
  - Cons
    - Small write problem
    - Degraded reads = occasional high read latency due to having to reconstruct data by reading n blocks to compute a missing block
    - Repair traffic = having to wire said n blocks through the network to a machine that computes the missing block
    - Potentially indirectly hurting throughput & latency
      - To reconstruct a failed disk need to read n disks (lots of storage IO) and send their data through the network (lots of network IO)

• **Replication (k)**
  - Pros: none of the above cons
  - Cons: wasting (k-1)/k of the capacity
THE END
Virtual File System

VFS
Filesystem architecture in a nutshell

- **Reusing code**
  - As much as possible
  - Different physical filesystems are implemented differently
    - VFS hides this from syscalls
    - dentry = directory entry
  - All filesystems read/write through the page cache
  - Which in turn uses a generic block device
  - Block ops are reordered & scheduled
    - To make accesses as sequential as possible
  - Only then need device-specific info
Why VFS?

• There are many filesystems...
  – ~70 in filesystem directories under http://lxr.linux.no/#linux+v3.9.7/fs/

• Examples
  – Local: Ext2, ufs (Solaris), svfs (SysV), ffs (BSD), ...
  – Network: RFS, NFS, Andrew, Coda, Samba, Novell, ...
  – Journaling: Ext3, Ext4, Veritas, ReiserFS, XFS, JFS, ...
  – Media-specific: jffs, ISO9660 (cd), UDF (dvd), ...
  – Special: /proc, /sys, tmpfs, sockfs, ramfs, ...
  – proprietary: MSDOS, VFAT, NTFS, Mac, Amiga, ...
  – Interesting: sshfs, gmailfs, FUSE (user space FS), unionfs, ...
Why VFS?

• **Object-oriented abstraction of a physical filesystem**
  – Separates filesystem generic options from implementation details
  – Same API for multiple filesystems
    • Local & remote
    • Device can be HDD, but also USB, DVD, network, DRAM, ...
    • …

• **Easier to implement system calls**
  – Kernel system call code operates without regard to the specific type of filesystem being accessed

• **Users can view directory tree as single entity**
  – Even when tree is made up of a number of diverse filesystems (mounted on different directories)
VFS stakeholders

• **VFS objects:**
  – inode, file, superblock, dentry
    • VFS defines per-object ops (C struct)
    • Each object (C struct instance) has pointer to a function table
      – Addresses of routines implementing functions for that object

• **VFS “users”**
  – File-related system calls (~100!)
  – Utilizes pointers to functions in VFS objects
  – “Everything is a file” (devices: /dev/ttys0 serial port; processes: /proc/123; ...)

• **VFS implementations**
  – Physical filesystem that translate abstract VFS ops to real ops
  – Store on disk, send via network, ...
  – Implements functions pointed to by VFS objects

• **Example (we’re discussing Linux’s VFS)**
  – libc’s read(2) => kernel’s sys_read => file->f_op->read(...)
System calls utilizing VFS

- **Filesystem ops**
  - (u)mount, flush, chroot, pivot_root

- **Directory ops**
  - chdir, getcwd, link, unlink, rename, symlink

- **File ops**
  - open, close, (p)read(v), (p)write(v), seek, truncate, dup, fcntl, creat, flock

- **Inode ops**
  - stat, chmod, chown

- **Memory mapping**
  - m(un)map, madvise (rand/seq access?), m(un)lock(all) (pins memory)

- **Flushing**
  - sync(fs) (commit page cache / fs), f(data)sync , msync (for mapped)

- **Wait for input**
  - poll, select
BTW (scatter/gather)

• **readv/writev** = read/write to/from multiple buffers
  – Get an array of
    ```c
    struct iovec {
        void *iov_base; /* start address */
        size_t iov_len;   /* size of */
    };
    ```

• **Also known as scatter/gather I/O**
  – A single call that sequentially writes data from multiple buffers to a single file or reads data from a file to multiple buffers
  – The buffers are given in a vector of buffers (iovec array).
  – Scatter/gather refers to the process of gathering data from, or scattering data into, the given set of buffers
Linux VFS objects

- **struct file**
  - Info about an open file of a process, includes current position
- **struct dentry**
  - Reflects one directory entry, including name and inode#
  - Need a cache for these for performance ("/1/2" => "/" + "/1" + "/1/2")
- **struct inode**
  - Unique descriptor of a file,
  - Includes inode#, block maps, times, permissions, ...
- **struct superblock**
  - Descriptor of the mounted filesystem
- **struct file_system_type**
  - info how to read a superblock
- **struct vfsmount**
  - Represents one mounted instance of a filesystem
  - (One superblock can be mounted in more than one place)
  - For mount points, file lookup requires both a dentry and a vfsmount
Data structure relationships
What we didn’t cover, alas

• network file systems
• log structured filesystems
• journaling
• snapshots
• distributed filesystems
• deduplication
• ...

OS (234123) - files