Overview of the I/O subsystem

Producer/consumer relationships
Hierarchical relationships: drivers call other drivers.
Every device has a major and minor number.  
  Major number: determines kind of device.  
  Minor number: determines which device of a kind.  
Inside the OS, devices are known entirely by number.  
  Names are given to devices only to aid humans in understanding them.
Driver table structure

kernel drive table

0
1 major device
2
3

device type
⇒ device #
⇒ entry in driver table
Kernel's eye view of a device

![Diagram of a device viewed by the kernel]

- Drive
- Device
- Drivers
- Process
- Process
- Process
- Process
- Process

functions.
Inside a kernel driver table entry:

```c
int (*open)(...);
int (*close)(...);
int (*read)(void *where, int size, int fd);
int (*write)(void *where, int size, int fd);
int (*ioctl)(int fd, int operation, int flags);
...
```

Structure of the kernel driver table:

- To define a device, define its methods.
- Pointers to function: pointers to specific methods to use for various tasks.
- Iocctl: I/o control: performs control operations on the device.
- Example: changing the buffering of a terminal device.
Window into OS: /dev
/dev: directory of devices
  block-special: functions in block mode only
  character-special: can write by character as well as block.
ls -l /dev/device:
  lists major and minor number
  b=block special, c=character special
  the files listed are made by mknod
  (these are not normal files at all!)

one use of raw devices: disk copying
dd if=/dev/fd0H1440 of=copy.img bs=20B
  • if=input file= raw floppy device
  • of=output file = file on disk
  • bs=block size for copy = 20 physical blocks
dd if=copy.img of=/dev/fd0H1440 bs=20B
  • from file
  • to raw disk
  • 20 physical blocks at a time
What do drivers do?

handle idiosyncrasies of each device
maintain a consistent idea of device state
driver: code that maintains state
descriptor: a representation of that state.

The idea is simple, but a descriptor can maintain any concept of state.
The state is a void * pointer passed to driver functions.
So what is a file?

/dev contains devices.
Certain kinds of devices contain files.
But we are getting ahead of ourselves.
Let's build up a model of the whole filesystem from the bottom up.
I/O and disks
So far, we've studied devices and major/minor numbers. This is a single-layer model. Disks have a multi-layer I/O model. Drivers talk to each other.

open calls the filesystem driver, which calls the raw disk driver.
The raw disk driver defines open, close, read, and write for disk blocks. The filesystem driver defines open, close, read, and write for files.
A modern disk is a set of layers:

The physical media.
  Geometry
  Sparing and defragmentation
The raw device.
  Operations: write/read
  Paging
  Scheduling
The filesystem device.
  The concept of a file.
  The concept of a directory.
Ambiguity: use open() to open either devices or files

**File I/O**: talks with a layer above device.
**Filesystem**: a structure for internal format of disk devices
**Mounting**: associate a disk with a place in the directory hierarchy. (opens the disk device!)
**mkfs/newfs**: creates the filesystem on the raw disk device
**Unmounting**: dissassociate disk and place in hierarchy. (closes the disk device!)
Lots of layers between device and file access.

Device I/O: talks with device directly.
Sometimes called "raw device I/O".
Strips layers away.
Can access disk as raw device.
Sometimes used to **recover data** from bad disks.
A corrupt disk, when mounted, can **crash** the operating system!
Disk structure (and illusions)
Round platter
Divided into tracks and sectors
Tracks: offsets from center.
Sectors: rotational offset.
NOT!
Modern disks
"geometry" is a **baldfaced lie.**
Inside a disk: 1,000,000 lines of C code.

Lots of internal tricks that are not visible to the OS:
Sparing: replacing one sector with another
Variable-sector maps: more sectors on outside than inside.

Sparing: recovering (automatically) from disk errors
Bad block: scratched disk platter or equivalent.
Spare table: a special track or tracks used to replace bad blocks.
Virtual geometry

It is physically impossible to have the same number of sectors for every track. Outer tracks have more media than inner tracks. Thus there are more sectors at the edge than in the middle!
How sparing works.
How do disks fail?

Globally: controller failure.
Scratches: from dropping a spinning disk.
Scratches affect a group of sectors.
Very subtle point:
- Disks can only read and write blocks at a time.
- Processes often write parts of blocks to files.
- How to mediate?

Ambiguity:
Several blocking factors
- How big is a "page" of the filesystem?
  "page" =~ 8192 bytes
- How much of the disk can be read at a time?
  "sector" =~ 512 bytes

Answer: make memory images of disk "blocks". This is called "the disk paging subsystem". Standardize on one page as the read/write unit. Ignore sectors.
The paging subsystem

- When one opens a file, allocate the file pages from a "page pool" of available kernel pages.
- When a process writes to the file,
  - Read current page of file into page pool
  - Perform change to page in page pool.
  - When convenient, "flush" changes from page pool to file.
  - Do this as late as possible.

Page pool is
  A kernel array
  A fixed-length resource.
  Allocated at boot time.
Big mystery: why can't one just turn the power off on a UNIX machine:
○ Answer: disk image of page cache for files is never up to date.
○ Page pool contains "dirty copies" of
  ▪ directory blocks
  ▪ file blocks
  ▪ inodes: where files are located
○ A "dirty page" is out of sync between memory and disk.
○ To turn a machine off, must synchronize page pool to disk image (command: sync)
○ unix delegates page pool flushes to a separate process in user space: "update".
○ Flushes do not happen when file is closed!
○ Absolutely no way that a normal process can assure that its data is flushed to disk!

○ shutdown a machine:
  ○ shutdown now "for a quick reboot"
○ If the machine is on fire:
  ○ sync; sync
    followed by flipping off the power.
Page pool in operation:

Process

write

driver

page

OS

write data into pool image

write data into pool

write data into pool image

weird page out, so lazy.

Inhibit page in

Disks_and_Layers Page 21
Paging concepts

- Dirty flag: marks each page as
  - "clean" (no write, no need to flush)
  - "dirty" (need to flush)
- If not "dirty", can reuse page without flush => update is not called.
- If "dirty", must flush page before reuse => update has to do the flush.
Disk scheduling

If processors are so fast, why is my computer so sloooow?
Answer: disk is largest bottleneck in a modern system.

Basic problem: disk geometry:
  ○ Two phases to a disk read or write
    ▪ Seek: put head in the right track.
    ▪ Read or write: read or write a sector of the track.
  ○ Two components to "seek"
    ▪ "Rotational delay": how long we have to wait for the sector to be under the head.
    ▪ Arm movement, A.K.A "offset": how long do we have to wait for the arm to be at the appropriate track?
Disk scheduling issues:
   In a multiprocessing system, processes contend for the same disk to read/write different data. The order in which seeks are made greatly determines throughput.
Question: how should one schedule/prioritize/order requests from autonomous processes so that throughput is optimal?
Better question: what is "optimal"?
Disk scheduling strategies:

Last-in, first-out:
  - Optimal for sequential access.
  - Not so optimal for random access.

Shortest service-time first (SSTF)
  - Keep queue of requests small.
  - High utilization.

SCAN: back and forth over disk
  - Pattern accesses so that arm zig-zags.

C-SCAN: sawtooth: forth with fast return.
  - Lower variability in service with direction of scan.

N-step scan: segment request queue, do round-robin of requests for each segment of tracks.
Flat areas: arm is still, read a sector
Slanted areas: chase tracks.

"SCAN"
don't seek to every track
Imparting meaning

In the preceding, we studied how blocks are written to disk.
Now, we will **impert meaning** to the blocks and build something new: a filesystem.
Layers and meanings:

So far, we have one meaning for disk data: blocks. We will build up the concept of files in layers.

Each layer

Imparts a new form of meaning.

That is not known at lower layers.

First layer: the raw disk
Meaning: a physical location L on the disk contains data.

Second layer: the raw device
Meaning: maps logical block N -> physical location L

Third layer: the filesystem
Meaning: make a file out of a sequence of blocks. Give the file a unique number in the filesystem.

Fourth layer: directories:
Meaning: gives numbered files human-readable names.
Components of a filesystem:

**Inode**: describes where files are
- Contains owner, group, protection.
- Does not contain names.
-Stored in a logical array.
- Indexed by number.
- Stored in sections of the disk called "inode groups". Multiple inodes/block.

**Block**: describes content of a file.
- A file is logically a sequence of blocks.
- A logical linked list.
- Allocated only in whole-block units.
- Stored in sections of the disk called "block groups".

**Superblock**: a special kind of block that describes the architecture of the disk and where to find inode groups, block groups, and the root directory (/).

**Directory**: a special kind of file
- Contains name to inode mapping.
- Has its own inode.
Two concepts of structure of a filesystem
   Logical structure: how blocks are logically arranged to create files.
   Physical structure: how this maps to a device with block numbers.
As usual:
   Logical structure is simple.
   Physical structure is complex.
Reason: performance.
Logically, a null-terminated linked list. Inode determines file properties. Blocks determine file contents. **Minimum file size is one block**, incremented in one block increments.
Inode table is logically an array. Each inode points to a linked list of blocks that is the file.
The logical structure of a directory

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>50</td>
</tr>
<tr>
<td>bar</td>
<td>22</td>
</tr>
<tr>
<td>cat</td>
<td>32</td>
</tr>
</tbody>
</table>
Logical layout of an entire Linux filesystem:
Directories form a "tree". In each directory,
".." points to the parent directory.
"." points to the current directory.
Directories **map** names to inodes. **Inodes** contain access privileges, and point to blocks representing the file.
A file or directory is a **sequence** of blocks.
Solution:
Ditch the lists.
Put an array in the inode.
First few entries point to the first few blocks.
The next entry points to a block that is -- in actuality -- an array of pointers to blocks.