Filesystems; Tails From The Trenches

DIKU — OSM

<oleks@oleks.info>

March 14, 2016

Agenda

Today

Introduction to Filesystems, FAT32, KUDOS VFS & TFS

Friday

(Linux) Containers i.e., how Docker “works”.

The contents of the Friday lecture **is not** exam material.

Before We Begin..

What is a "file"?

What is a "file"?

— A stream of bytes.

Examples:

- `main.c`, `Makefile`, `kudos-mips32`, `report.pdf`, etc.
  - Finite length, random access, can be appended to.
- `stdin`, `stdout`, `stderr`, etc.
  - Infinite length, but no random access.

Hmm. . . what else is *accessible* as a stream of bytes?

The UNIX/Linux/BSD Approach

— Everything is a file![1]

Examples:

- Directories
- Binaries
- Shell or Python scripts
- PDF documents
- Spreadsheets
- Kernel data structures
- Hard drives
- Partitions
- Printers
- Sockets

[1] Everything is accessible through a uniform filesystem interface.

Filenames, Paths, and Pathnames

When we think of a file, it is something that has a **name** and resides in a directory at a given **path**.

We could call the concatenation of the path and the filename a **pathname**.

I/O API

- `open(2)`
- `read(2)/write(2)`
- `close(2)`

I/O API

- `open(2)`
- `read(2)/write(2)`
- `close(2)`

Oh and maybe also

- `creat(2)`
- `unlink(2)`
- `...`

DEMO

The `echo`, `cat`, and `strace` utilities.

DEMO: Log

- Put something in a file and print it:

```
$ echo hello > foo
$ cat foo
hello
```

- Use `strace` to trace system calls:

```
$ strace cat foo
execve(...
...
open("foo", O_RDONLY)
...
```

- You can limit `strace` to trace a handful system calls:

```
$ strace -e trace=open,read,write,close cat foo
```

Buffered I/O API

Cool, so should I use `open(2)`, `read(2)`, `write(2)`, `close(2)`, etc. in my code?

Buffered I/O API

Cool, so should I use `open(2)`, `read(2)`, `write(2)`, `close(2)`, etc. in my code?

— No!

Use (userland) buffered I/O:

- Include `<stdio.h>`.
- Use `fclose(3)`, `fread(3)`, `fwrite(3)`, `fclose(3)`, etc.
- Perhaps even use `fprintf(3)` and `fscanf(3)`.
- Profit from the **reduced number of system calls**.

File Descriptors

What you really use when working on an open file.

Given a pathname for a file, `open()` returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (`read(2)`, `write(2)`, `lseek(2)`, `fcntl(2)`, etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

— `open(2), 2015-12-05`

File Types

If everything is a file, then how do we distinguish file types?

- Some operating systems rely on a filename extension.
- Others rely on file attributes and file structure.

DEMO

`/dev`, `/proc`, and the `file` utility.

DEMO: Log

- `file` can often be used to determine file type.
- There's a device with a pathname `/dev/stdin`.

```
$ file /dev/stdin
/dev/stdin: symbolic link to /proc/self/fd/0
```

- That is a pathname to your filedescriptor 0.

```
$ file /proc/self/fd/0
/proc/self/fd/0: symbolic link to /dev/pts/1
```

- That is a character device:

```
$ file /dev/pts/1
/dev/pts/1: character special
```

- `pts` stands for pseudo-terminal slave.

What is a "filesystem"?

— An organization of files into a system.

Why?

- Provide an illusion of a limitless number of limitless files.
- Catalogue files into hierarchies and/or graphs.
- Provide for long-term storage, without exposing the low-level mechanics of the storage device(s).
- Provide for fast access, despite the use of notoriously *slow* long-term storage devices(s).
- Provide for durability in the face of power/user failures.
- …

Devices, Devices, Devices

Character Devices

- Read/write one character at a time.
- 1 character = 1 byte.

Block Devices

- Read/write one block at a time.
- Block size is typically a multiple of 512 bytes.

Sectors and Clusters

- A "block" is a logical abstraction.
- A "sector" is an actual minimum unit of storage on a typical physical device.
- A "block" or "cluster" is a group of sectors.

Volumes and Partitions

A physical drive is perhaps partitioned into multiple volumes.

A volume is a storage entity holding a filesystem.

Mounting

Unified filesystem

- Start at /.
- Mount any device anywhere.
- Typical of UNIX, Linux, BSD.

Volume-first

- All pathnames begin with a volumename.
- Typical of Windows, KUDOS.

Cool, so can I put a volume in a file?

Cool, so can I put a volume in a file?

— Of course, you can!

Some popular examples:

- ISO 9660, known colloquially as "`.iso`-files".
  - The file-format we use to store copies of CDs.
- `squashfs`, a compressed, read-only filesystem.
  - Used in a Linux live-CD near you.
- TFS in your KUDOS.

DEMO

The `dd` and `mkdosfs` utilities... and `mount`.

DEMO: Log (1/3)

- Create something that looks like a disk:

```
$ dd if=/dev/zero of=fatdisk bs=1M count=64
64+0 records in
64+0 records out
67108864 bytes (67 MB, 64 MiB) ...
```

- Turn `fatdisk` into a FAT32 volume (**format** as FAT32):

```
$ mkdosfs -F 32 fatdisk
mkfs.fat 3.0.28 (2015-05-16)
```

- Check your work:

```
$ file fatdisk
fatdisk: DOS/MBR boot sector...
OEM-ID "mkfs.fat"... FAT (32 bit) ...
```

- OEM-ID might tell you who formatted the volume.

DEMO: Log (2/3)

- Mount that "disk":

```
$ ... mount fatdisk workdir -o rw,umask=0000
```

- Copy our file to it, and unmount:

```
$ cp foo workdir/
$ ... umount workdir/
```

- Now let's write it to disk!

```
$ ... dd if=fatdisk of=/dev/sdb1
```

- `/dev/sdb1` is a partition on my USB drive, use `... fdisk -l` to find such partitions.

```
$ ... fdisk -l
...
/dev/sdb1          ...  3.8G  b W95 FAT32
```

DEMO: Log (3/3)

- Check your work:

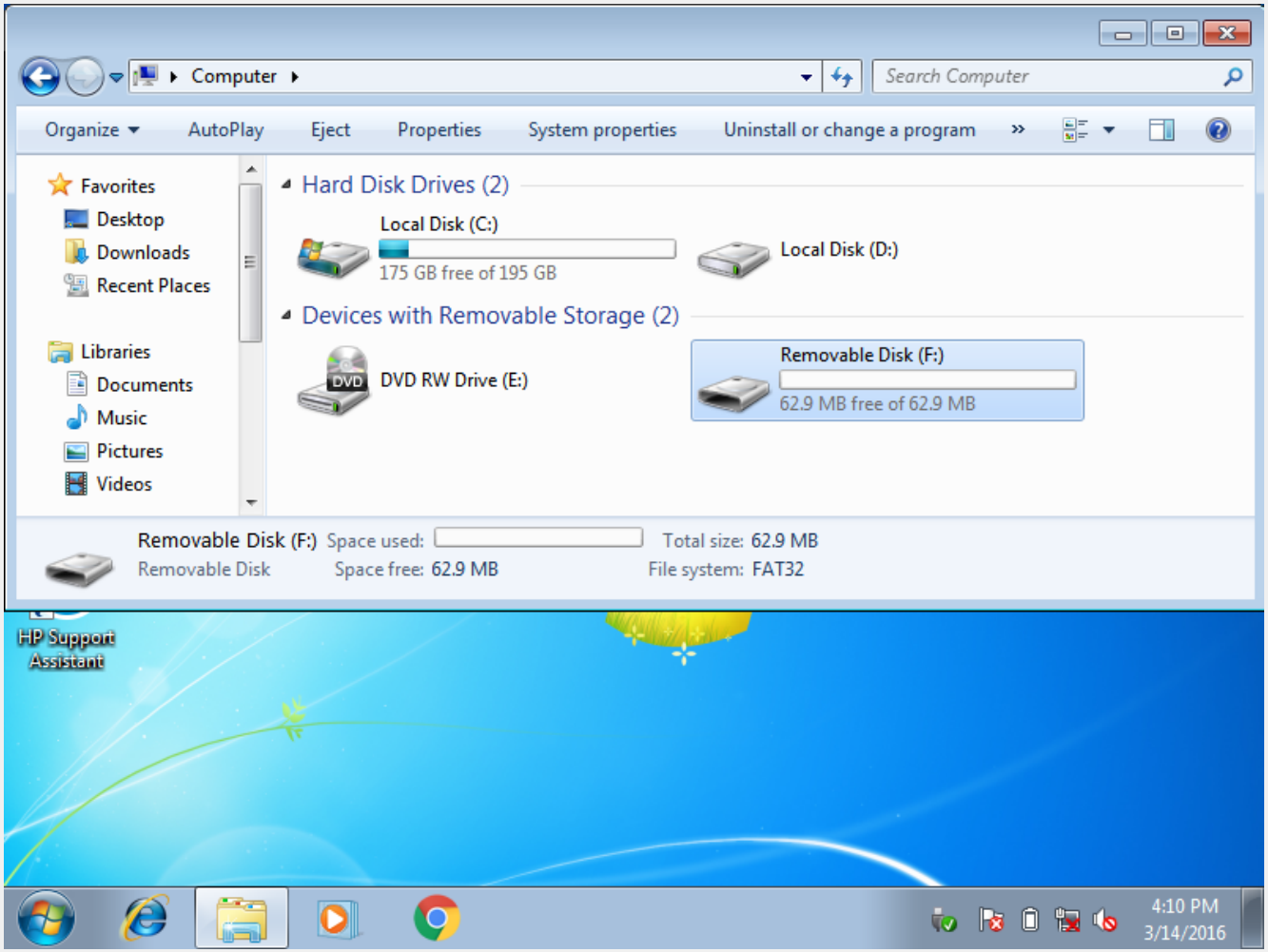
```
$ ... mount /dev/sdb1 workdir/
$ cat workdir/foo
hello
$ umount workdir/
```

Cool, so should I try this at home?

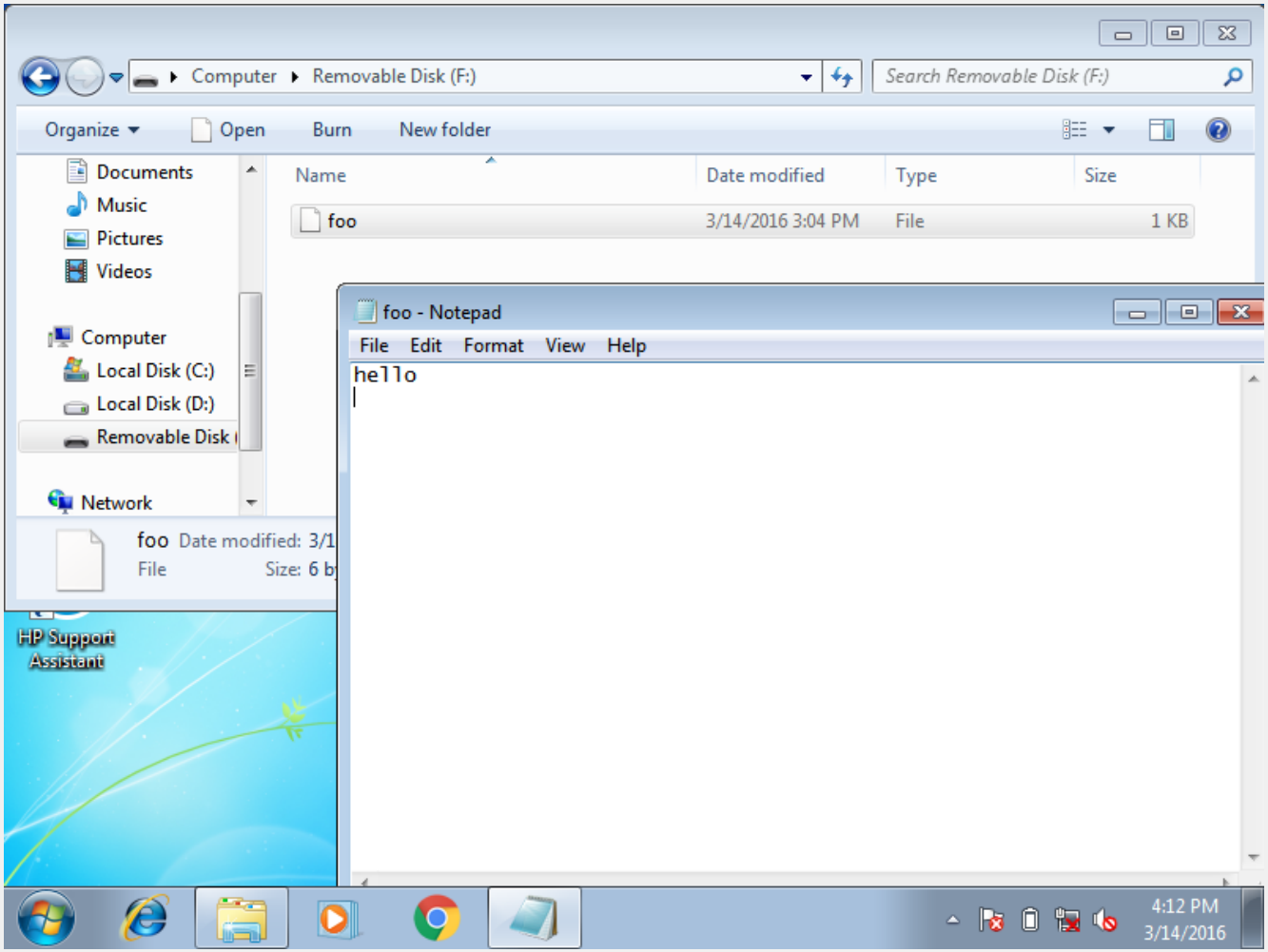
— Yes, but beware…

- It is easy to `sudo dd if=... of=/dev/sda`
- `/dev/sda` is typically your main harddrive.
- Linux **will not** stop you.
- Mount the device directly, don't `dd` filesystems (too often).
- As on the previous slide.

DEMO: Screenshots (1/2)



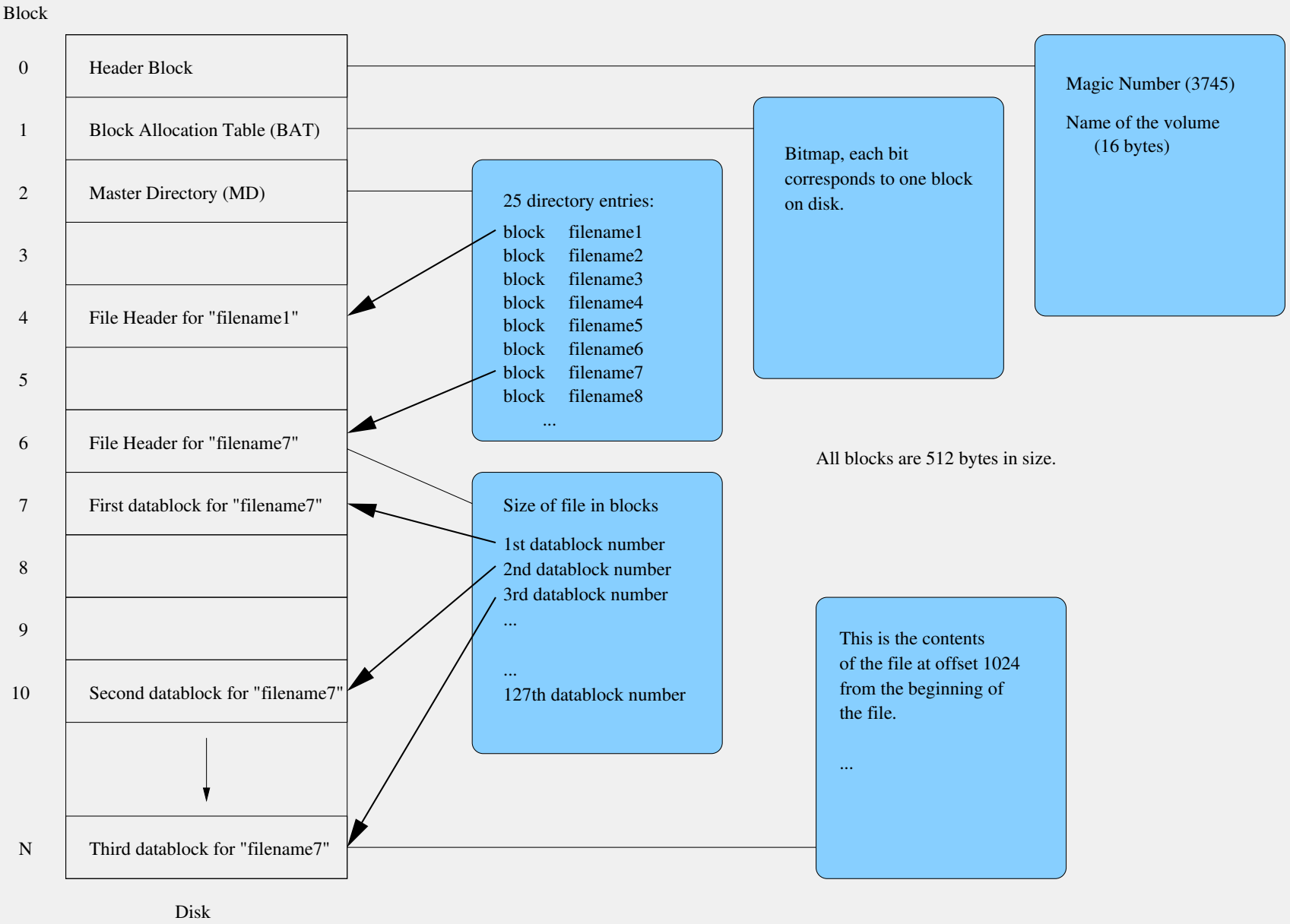
DEMO: Screenshots (2/2)



DEMO

Look at that "file"!

DEMO: TFS Overview



DEMO: Log (1/3)

- All blocks in TFS are 512 bytes; the first block is 4-bytes magic number, 16 bytes volume name, the rest is wasted:

```
union header {
  struct {
    uint32_t magic;
    char volname[16];
  };
  uint8_t padding[512];
} header;
```

- TFS is big-endian, my machine (x86_64) is little-endian:

```
uint2_t from_big_endian(uint32_t value) {
  return ((value & 0xFF000000) >> 24) |
         ((value & 0x00FF0000) >> 8)  |
         ((value & 0x0000FF00) << 8)  |
         ((value & 0x000000FF) << 24);
}
```

DEMO: Log (2/3)

Use (userland) buffered I/O:

```
FILE *stream;
size_t read;
int retval;
...
stream = fopen("store.file", "r");
...
read = fread(&header, sizeof(union header), 1, stream);
...
retval = fclose(stream);
```

DEMO: Log (3/3)

- See attached `tfs.c`:

```
$ make tfs
cc     tfs.c   -o tfs
$ ./tfs
Magic number: ea1
Volume name: disk
```

- See attached `fat32.c`:

```
$ make fat32
cc     fat32.c   -o fat32
$ ./fat32
jmpboot: eb5890
oemname: mkfs.fat
bytes_per_sec: 258
sec_per_clus: 32
```

- You could even modify `fat32.c` to read `"/dev/sdb1"` instead of `"fatdisk"` — this is how `mount` works.
- For more details on FAT, see:

  Microsoft Corporation, FAT: General Overview of On-Disk Format, Version 1.03, December 6, 2000.
  http://www.webcitation.org/6g00HfQxp

Allocation Strategies

- Contiguous allocation
- Linked allocation
  - A link to next block in every block.
  - A range of dedicated blocks for a table of links, e.g. FAT.
- Indexed allocation, e.g. TFS.
  - Index nodes or inodes.
  - Single-indirect.
  - Double-indirect.

Reading Material

- OSTEP
  - 39. Files and Directories
  - 40. File System Implementation
- KUDOS Documentation
  - Filesystems
  - Virtual Filesystem
  - Trivial Filesystem
- Dragon Book (see Absalon)
  - 11. Implementing File-Systems