The basics of file systems

Neapolis University Pafos

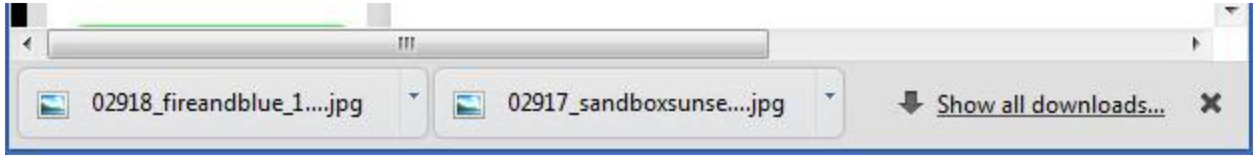
JETBRAINS

What a file system must do:

- store our data

What a file system must do:

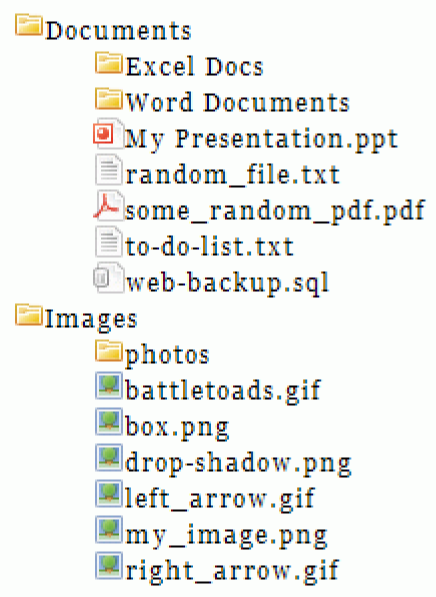
- store our data



This organisation becomes inconvenient
if we have thousands of files.

What a file system must do:

- store our data,
- organise data into a hierarchy of files and directories



What a file system must do:

- store our data,
- organise data into a hierarchy of files and directories,
- provide and limit the access to stored files.

Today we will limit ourselves to file systems that store data in a single computer and provide access only to local users.

The desired interface to a file system:

```
f = open(“./pstorage-fes/src/fes.c");
read(f, buffer, size);
.....
write(f, buffer, size);
.....
close(f);
```

The desired interface to a file system:	The interface of a storage device:

f = open(“./pstorage-fes/src/fes.c");
read(f, buffer, size);
.....
write(f, buffer, size);
.....
close(f);
	* read a sector* nr. N,
	* write a sector nr. M.
	
	** a sector is a contiguous piece of a storage device that is 512 bytes or 4096 bytes long; the start of a sector is a multiple of the sector size*

The desired interface to a file system:

```
f = open(“./pstorage-fes/src/fes.c");
read(f, buffer, size);
.....
write(f, buffer, size);
.....
close(f);
```

The interface of a storage device:

* r

*

This restriction is not always true. Chips are getting denser, and nowadays a single PCI-e card can host a computer with 16 ARM cores, 32G RAM, 4x400Gbit ethernet, and dedicated accelerators for NVMe-oF, compression and erasure coding. For example, see Mellanox (Nvidia) BlueField. Such devices can provide a much more sophisticated API.

The desired interface to a file system:

```
f = open(“./pstorage-fes/src/fes.c");
read(f, buffer, size);
.....
write(f, buffer, size);
.....
close(f);
```

The interface of a storage device:

* read a sector* nr. N,
* write a sector nr. M.

** a sector is a contiguous piece of a storage device that is 512 bytes or 4096 bytes long; the start of a sector is a multiple of the sector size*

The task of a file system:

Atop of a block device provide an API that enables users to

- create files and directories,
- find files and directories by their name,
- write and read files at arbitrary offsets (not necessarily sector-aligned),
- do these operations fast and reliably.

A problem that a file system must solve: how does one store a list of files*?

An array with file names, unsorted:

file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0

** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

A problem that a file system must solve: how does one store a list of files*?

An array with file names, unsorted:

file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0

We need up to 16 string comparisons to find a file.

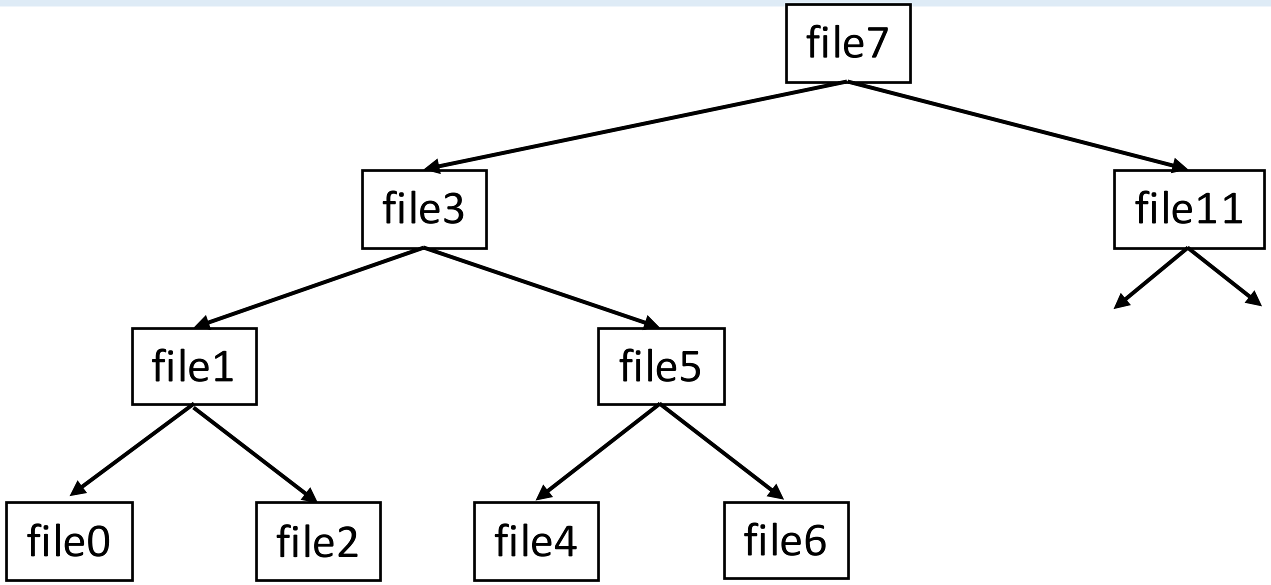
** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

A problem that a file system must solve: <u>how does one store a list of files*?</u>

An array with file names, unsorted:

file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0

A balanced binary search tree:



We need up to 16 string comparisons to find a file.

We need at most 4 comparisons to find a file. Win?

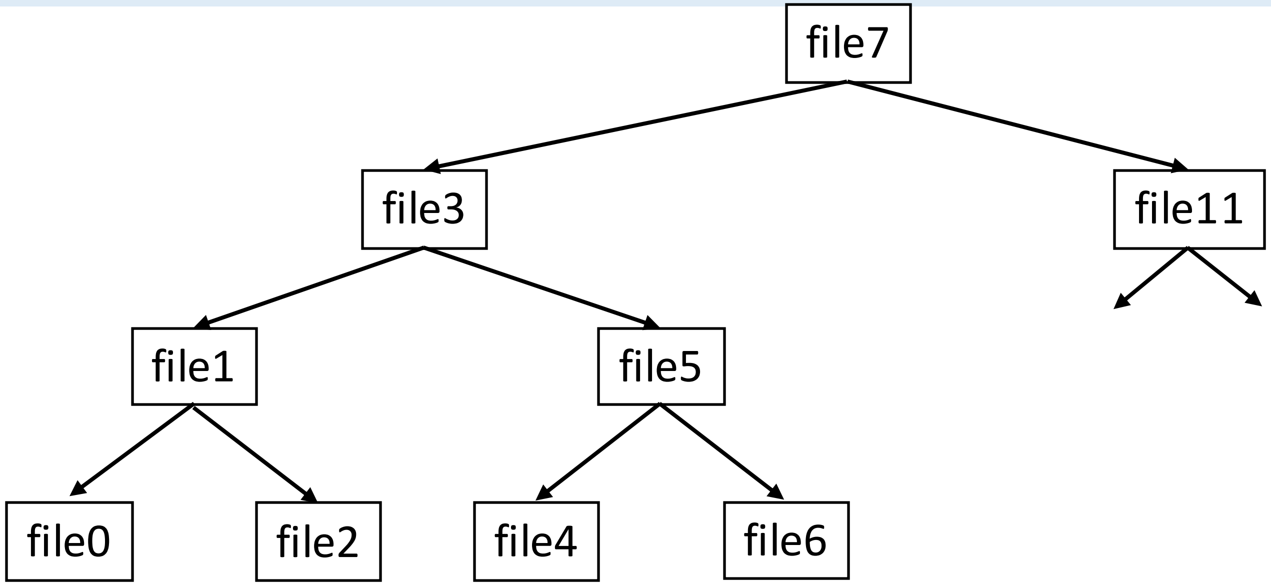
** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

A problem that a file system must solve: how does one store a list of files*?

An array with file names, unsorted:

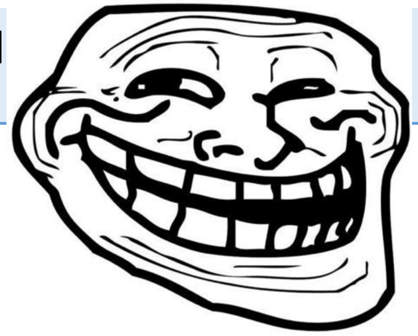
file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0

A balanced binary search tree:



We need up to 16 string comparisons to find a file.

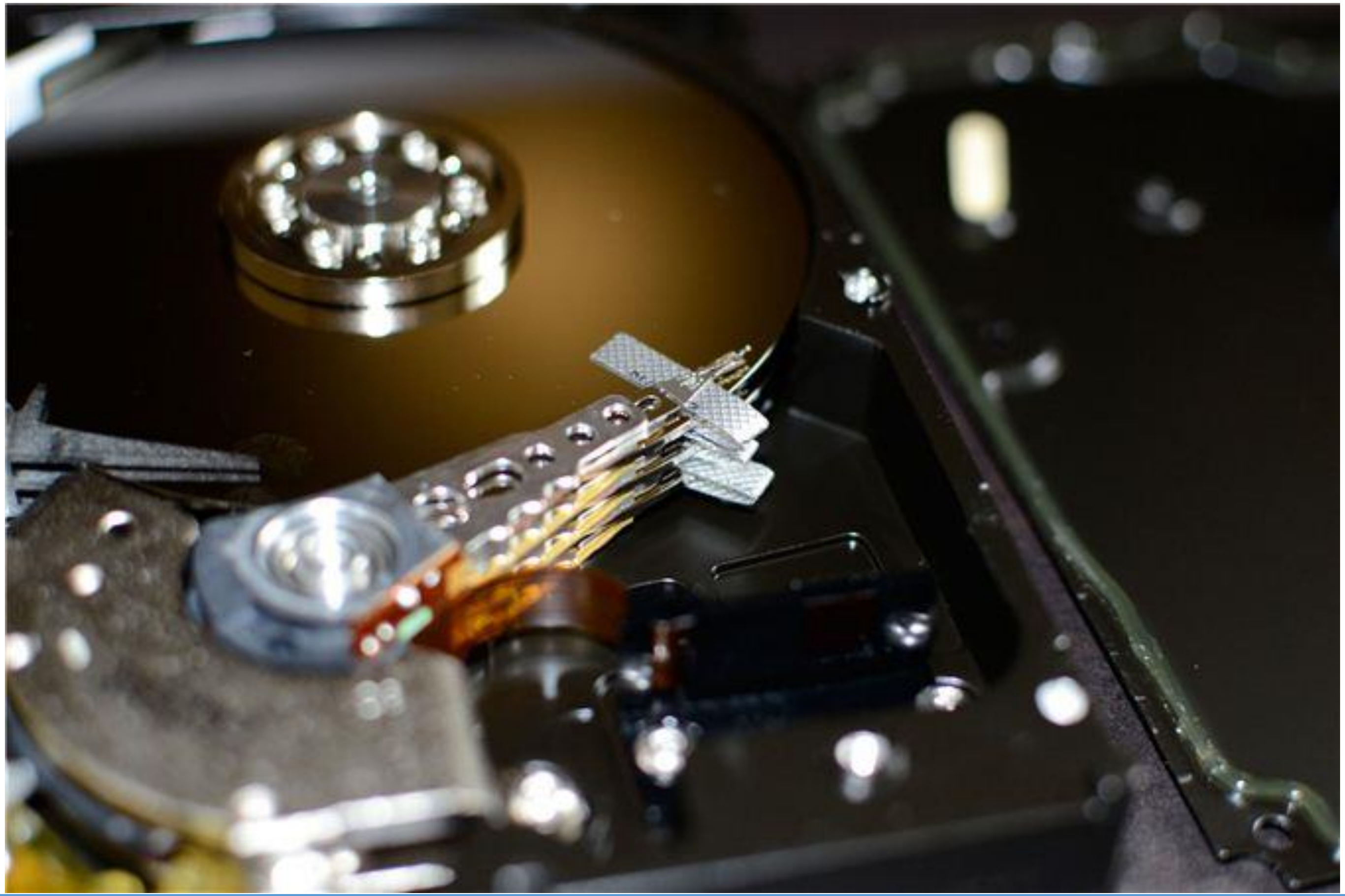
We need at most 4 comparisons to find a fil



** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

The reader head needs to be positioned very precisely. It takes much time to reposition it which makes random reads from HDD very slow. For reference:

- the speed of sequential reads from an HDD is ≈100 MB/sec, which is ≈10 ms per 1 MB,
- time to reposition the reader head is also ≈10 ms.

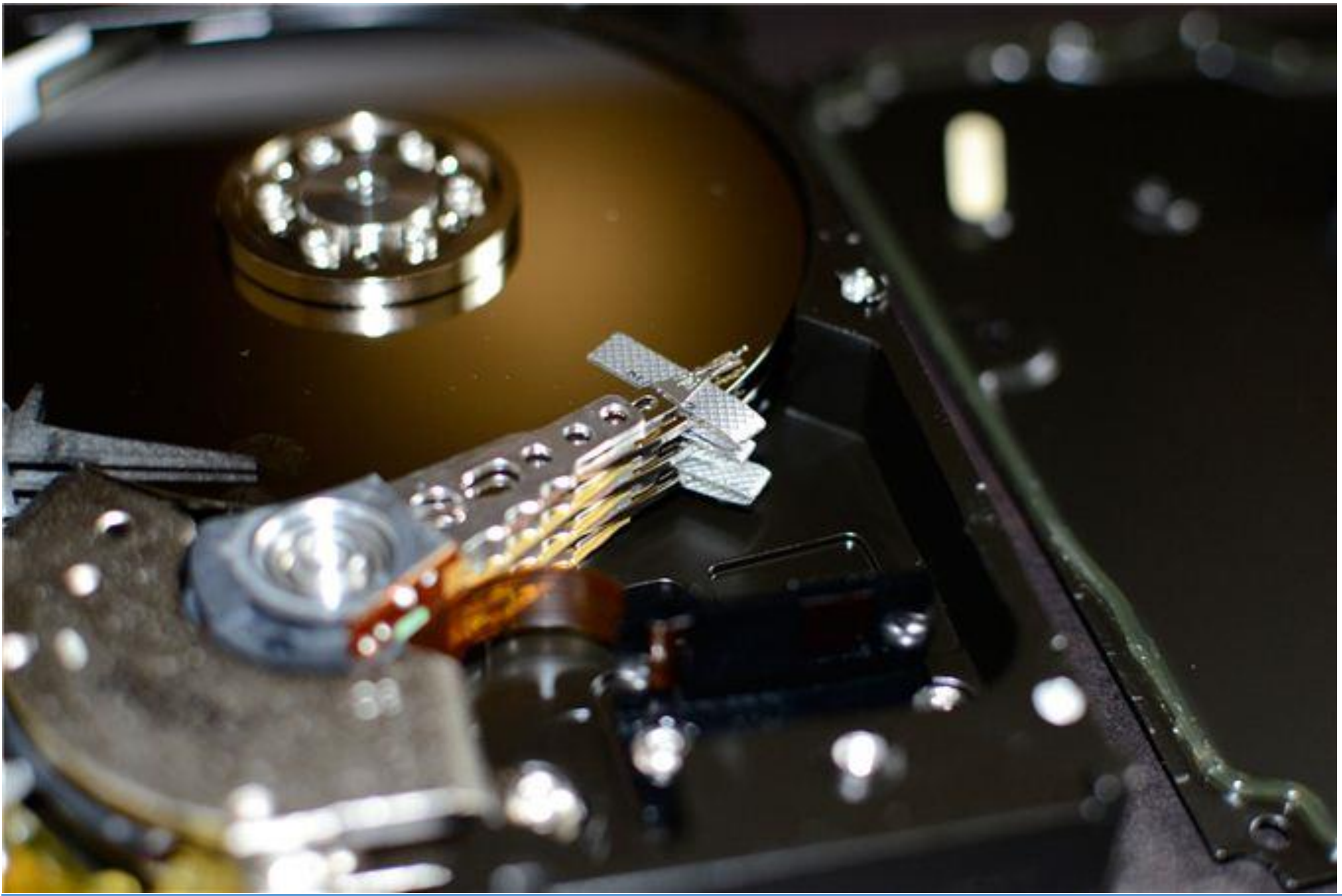


Modern HDDs are more performant. For example, Seagate Exos 16T does sequential reads at the rate of up to 250MB/sec and has a 6ms latency of random accesses.

The values of 100 MB/sec and 10ms are much handier for estimates.

The reader head needs to be positioned very precisely. It takes much time to reposition it which makes random reads from HDD very slow. For reference:

- the speed of sequential reads from an HDD is ≈100 MB/sec, which is ≈10 ms per 1 MB,
- time to reposition the reader head is also ≈10 ms.

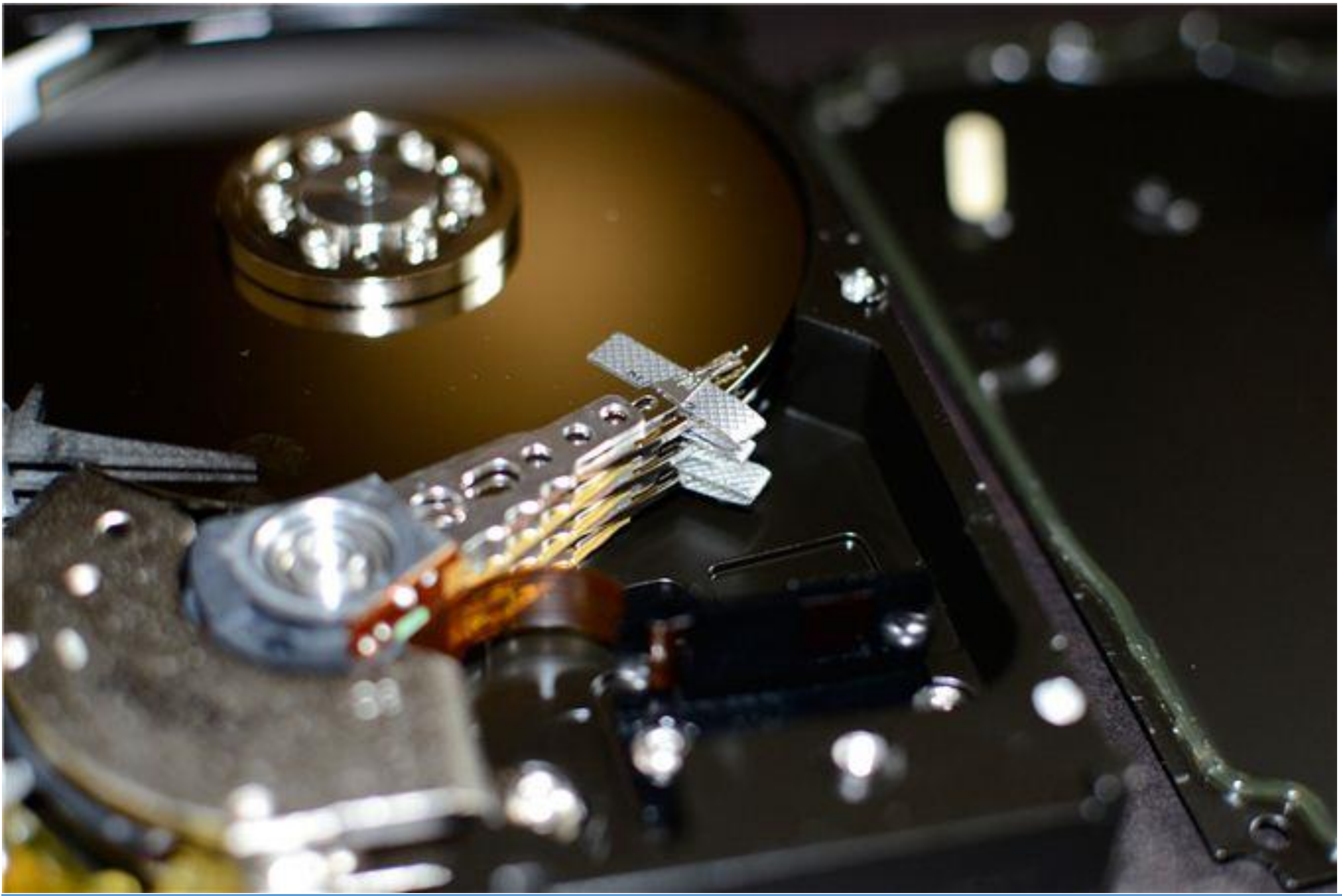


Let us change the scale and have durations that we see in the everyday life:
1ns ---> 1s

L1 latency (Zen 4, 5.7Ghz)	**0.7s**
L2 latency	**2.5s**
L3 latency	**10s**
RAM latency	**≈84s**

The reader head needs to be positioned very precisely. It takes much time to reposition it which makes random reads from HDD very slow. For reference:

- the speed of sequential reads from an HDD is ≈100 MB/sec, which is ≈10 ms per 1 MB,
- time to reposition the reader head is also ≈10 ms.



Let us change the scale and have durations that we see in the everyday life:
1ns ---> 1s

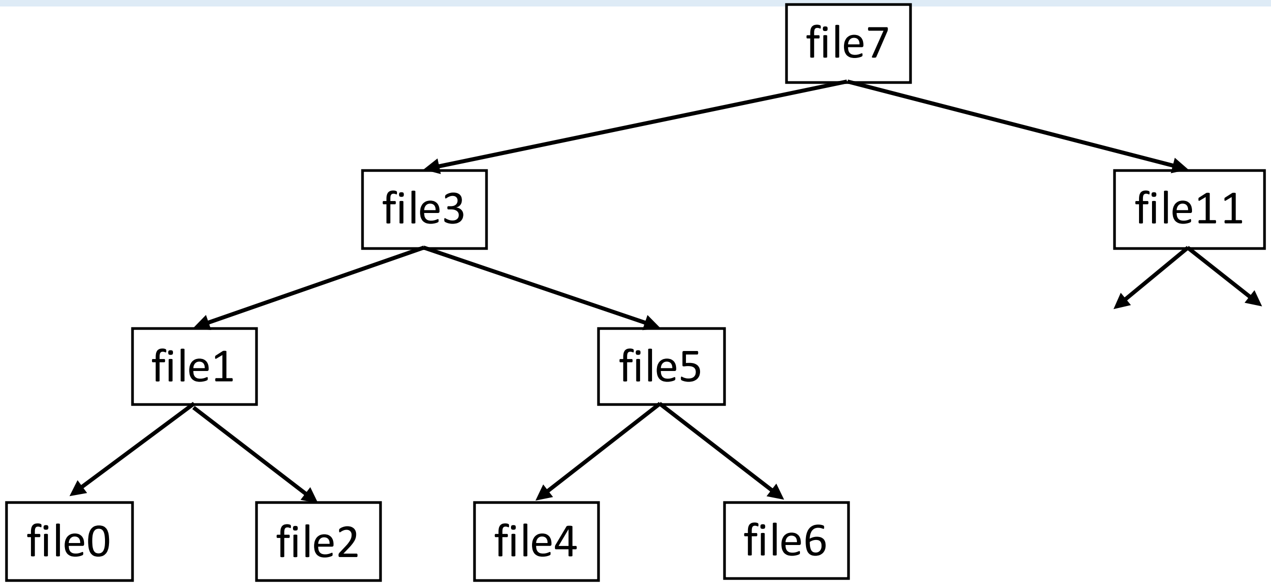
L1 latency (Zen 4, 5.7Ghz)	**0.7s**
L2 latency	**2.5s**
L3 latency	**10s**
RAM latency	**≈84s**
A read from an HDD	**116 days only to position the reader head**

A problem that a file system must solve: <u>how does one store a list of files*?</u>

An array with file names, unsorted:

file15, file1,
file2, file3,
file4, file9,
file6, file8,
file7, file5,
file12, file11,
file10, file13,
file14, file0

A balanced binary search tree:



Jump to the start of the list:	≈10msec
Read the whole list:	<1msec (<100K)
Scan the array in RAM:	<1msec

Most files need 4 or 3 random accesses which translates to latencies above 30 msec.

** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

Kinds of storage devices:

HDD (Hard Disk Drive, a.k.a. Rotating drive, a.k.a. Spinning rust)	+ sequential access is reasonably fast (≈250 MB/sec) - random access is very slow (≈100 IOPS*)

** IOPS stands for "Input/output Operations Per Second".*

** As per the spec of Intel SSD DC S3700 u D7-P5600.*

Kinds of storage devices:

HDD (Hard Disk Drive, a.k.a. Rotating drive, a.k.a. Spinning rust)	+ sequential access is reasonably fast (≈250 MB/sec) - random access is very slow (≈100 IOPS*)
Flash memory	+ fast sequential reads + no mechanical reader heads to reposition - no random writes; it is only possible to rewrite whole “rewrite blocks” that are several MB long - low number of rewrite cycles due to physical degradation of memory cells

** IOPS stands for “Input/output Operations Per Second”.*

** As per the spec of Intel SSD DC S3700 и D7-P5600.*

Kinds of storage devices:

HDD (Hard Disk Drive, a.k.a. Rotating drive, a.k.a. Spinning rust)	+ sequential access is reasonably fast (≈250 MB/sec) - random access is very slow (≈100 IOPS*)
Flash memory	+ fast sequential reads + no mechanical reader heads to reposition - no random writes; it is only possible to rewrite whole “rewrite blocks” that are several MB long - low number of rewrite cycles due to physical degradation of memory cells
SSD (Solid State Drive), SATA	Flash + a computer that hides the complexity of managing “rewrite blocks”. + fast sequential access (≈500 MB/sec sequential read*) + fast random access (≈75.000 IOPS)

** IOPS stands for “Input/output Operations Per Second”.*

** As per the spec of Intel SSD DC S3700 u D7-P5600.*

Kinds of storage devices:

HDD (Hard Disk Drive, a.k.a. Rotating drive, a.k.a. Spinning rust)	+ sequential access is reasonably fast (≈250 MB/sec) - random access is very slow (≈100 IOPS*)
Flash memory	+ fast sequential reads + no mechanical reader heads to reposition - no random writes; it is only possible to rewrite whole “rewrite blocks” that are several MB long - low number of rewrite cycles due to physical degradation of memory cells
SSD (Solid State Drive), SATA	Flash + a computer that hides the complexity of managing “rewrite blocks”. + fast sequential access (≈500 MB/sec sequential read*) + fast random access (≈75.000 IOPS)
SSD, NVMe	SSD with a faster interface: ≈5 GB/sec sequential read, ≈1M IOPS*.

** IOPS stands for “Input/output Operations Per Second”.*

** As per the spec of Intel SSD DC S3700 u D7-P5600.*

Kinds of storage devices:

HDD (Hard Disk Drive, a.k.a. Rotating drive, a.k.a. Spinning rust)	+ sequential access is reasonably fast (≈250 MB/sec) - random access is very slow (≈100 IOPS*)
Flash memory	+ fast sequential reads + no mechanical reader heads to reposition - no random writes; it is only possible to rewrite whole “rewrite blocks” that are several MB long - low number of rewrite cycles due to physical degradation of memory cells
SSD (Solid State Drive), SATA	Flash + a computer that hides the complexity of managing “rewrite blocks”. + fast sequential access (≈500 MB/sec sequential read*) + fast random access (≈75.000 IOPS)
SSD, NVMe	SSD with a faster interface: ≈5 GB/sec sequential read, ≈1M IOPS*.
Storage-class memory (3D cross-point memory, etc.)	Byte-addressable non-volatile random-access memory that connects to PCI-e or DRAM busses. “Non-volatile” means “does not lose data when powered off”. + the bandwidth and latency is comparable to DRAM, + the size is up to single-digit terabytes.

** IOPS stands for “Input/output Operations Per Second”.*

** As per the spec of Intel SSD DC S3700 u D7-P5600.*

APIs for working with file systems:

An operating system must hide hardware details from applications and provide a single API that can be used with different underlying storages.

- POSIX (Portable Operating System Interface),
- Windows API.

POSIX file system API

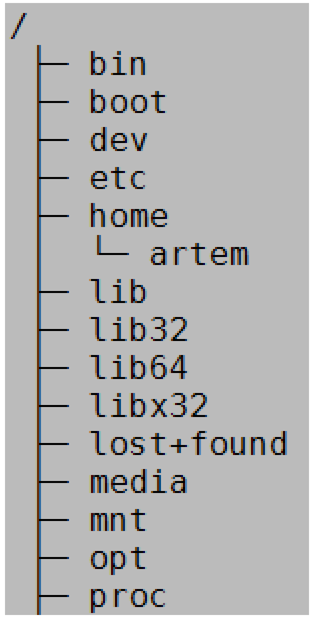
A file system is a tree of directories and files:

```
/
├── bin
├── boot
├── dev
├── etc
├── home
│   └── artem
├── lib
├── lib32
├── lib64
├── libx32
├── lost+found
├── media
├── mnt
├── opt
├── proc
```

The basics of file systems

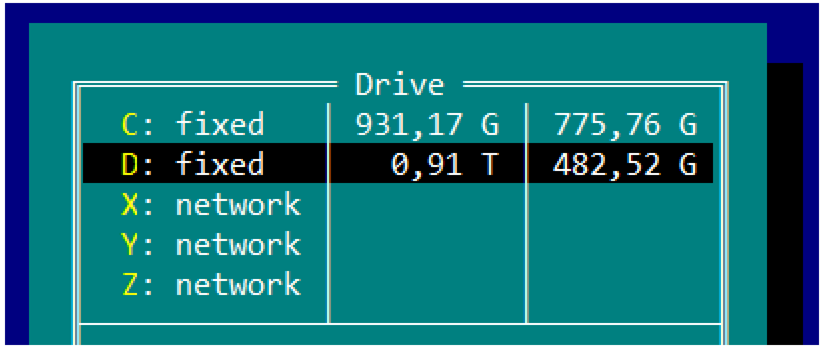
POSIX file system API

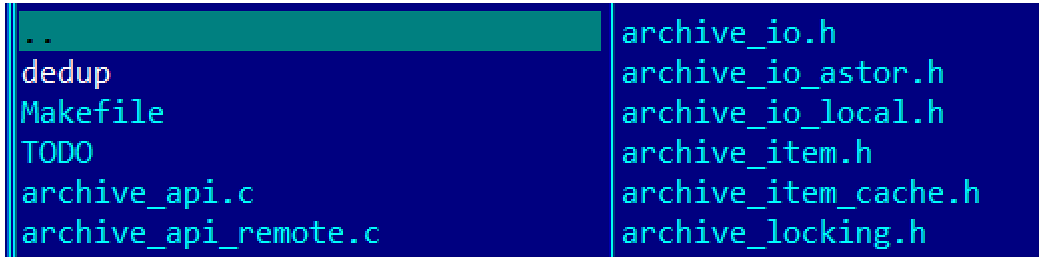
A file system is a tree of directories and files:



Windows file system API

In Windows, a file system is a collection of trees that Windows calls “drives”:

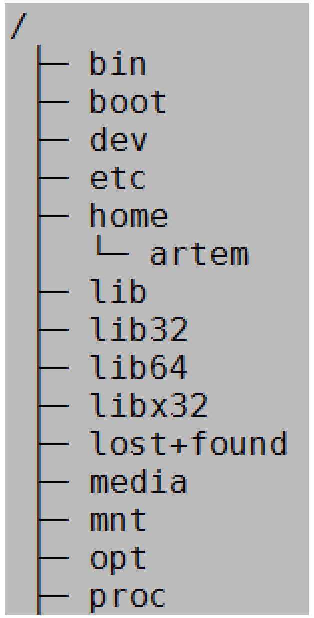




The basics of file systems

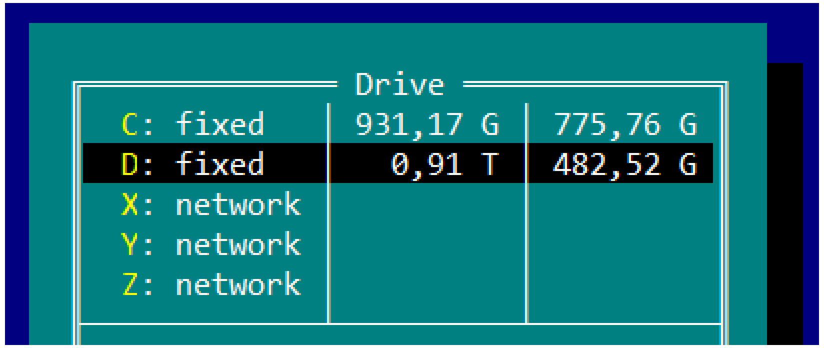
POSIX file system API

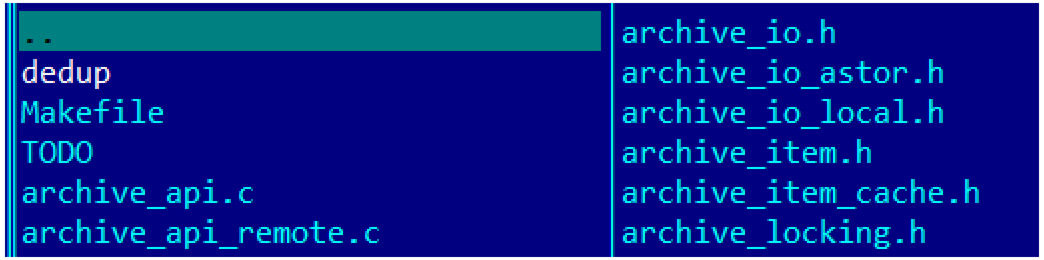
A file system is a tree of directories and files:



Windows file system API

In Windows, a file system is a collection of trees that Windows calls “drives”:

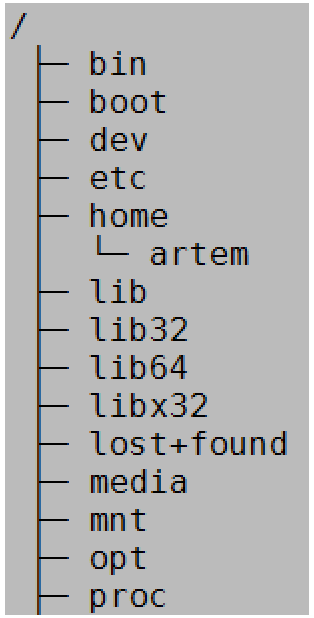


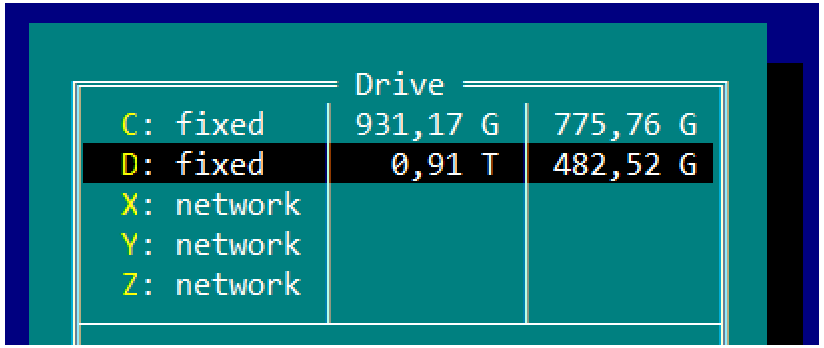


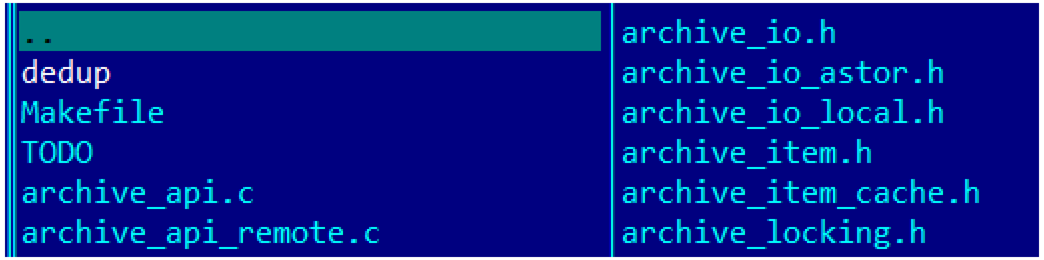
Filesystem Hierarchy Standard:
Linux:
http://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.pdf
FreeBSD:
https://www.freebsd.org/doc/handbook/dirstructure.html

The basics of file systems

POSIX file system API	Windows file system API
A file system is a tree of directories and files:	In Windows, a file system is a collection of trees that Windows calls “drives”:







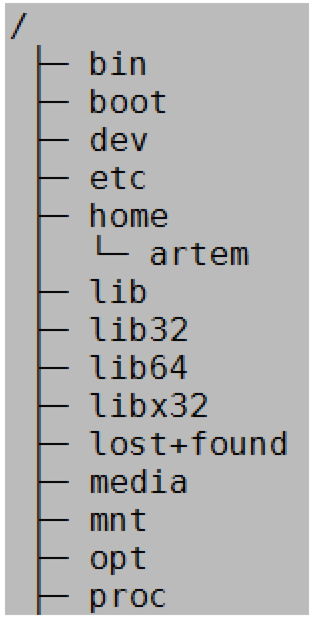
Filesystem Hierarchy Standard:
Linux:
http://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.pdf
FreeBSD:
https://www.freebsd.org/doc/handbook/dirstructure.html

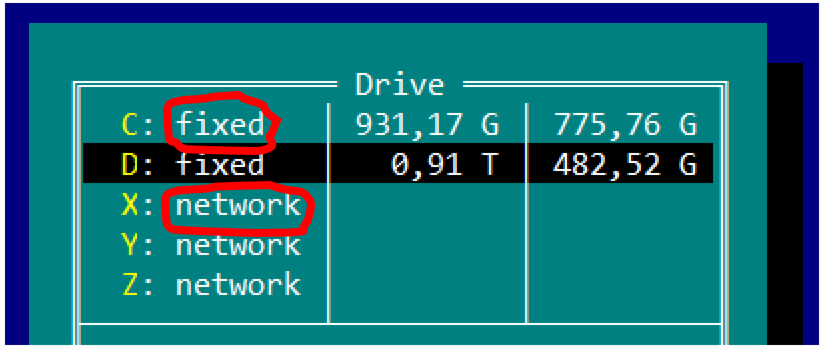
\Global??\C:\foo\bar.txt

An ambiguity in the terminology

A file system is a hierarchy of directories and files presented to a user

A file system is a mechanism of storing files and directories on a storage device.





A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)

- O_CREAT,
- O_EXCL,
- O_NOATIME,
- O_CLOEXEC.

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc

dirfd replaces the “current working directory” for openat(). This gives multiple improvements:

- ???

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc

dirfd replaces the “current working directory” for openat(). This gives multiple improvements:

- no races with chdir(),
- per-thread working directories instead of a process-global one,
- fewer steps to traverse the file system.

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc
5. symlink() / readlink()

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc
5. symlink() / readlink()
6. link() / unlink()

In POSIX, files and their names exist separately. The following situations are allowed:

- files with multiple names,
- files with no names.

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc
5. symlink() / readlink()
6. link() / unlink()

In POSIX, files and their names exist separately. The following situations are allowed:

- files with multiple names,
- files with no names.

open(O_TMPFILE) creates a file that has no name from the outset.

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc
5. symlink() / readlink()
6. link() / unlink()
7. Special files:

- directory,
- character devices,
- block devices,
- pipes,
- unix domain sockets.

A refresher on the POSIX Filesystem API

1. open(path, flags, mode) / close(fd)
2. mkdir(path, flags) / rmdir(path)
3. chdir(path), chroot(path)
4. openat(dirfd, path, flags) / mkdirat() / rmdirat() / etc
5. symlink() / readlink()
6. link() / unlink()
7. Special files
8. mmap() / munmap()