

# OBL4 - Operating Systems

---

By: John Ivar Eriksen

## 1 - File systems

---

### 1.1

Name two factors that are important in the design of a file system.

Two factors important in the design of a file system would be **reliability** and **performance**.

**Reliability** (and integrity) is the file systems ability to reliably store data and maintain data integrity over time. This includes preventing data loss, both at rest and during data transfers, with techniques such as journaling and copy-on-write. It must also be able to perform error detection and correction to prevent data corruption, with e.g. checksums or parity bits. Additionally, it must be able to handle events such as system crashes or power failures, without losing data even if mid-transfer, e.g. with the use of transactional updates and atomic operations.

**Performance** is how efficiently, both in terms of read/write speed and system resource usage, the file system is in its operations. It must optimize the way in which data is read and written (disk I/O) to minimize latency and maximize throughput. Caching and buffering strategies must be effectively implemented to reduce the need for disk access and improve response time for frequently used data. Additionally it is important to choose data structures and algorithms that are suitable for managing files and directories, making access and modification operations fast and responsive.

**A note on performance:** The factors defining what "performance" is will vary, often significantly so, with what the intended area of use for the system is. For example a system designed for "general use", i.e. a human-user desktop PC or workstation, will have other priorities than a system designed for backup or archival purposes. In the former, a snappy GUI and faster response times are essential for optimizing both user performance and experience, and likely a focus on efficiency with smaller files. For the latter, a snappy GUI is wasted resources, and have more to gain on prioritizing things such as data integrity, write performance and handling of large files.

### 1.2

Name some examples of file metadata.

Metadata for a file may be timestamps like when the file was created and when last modified or accessed. Other examples are access permissions, file attributes, file size, device type, owner user ID and group ID.

## 2 - Files and directories

---

### 2.1 - Consider a Fast File System (FFS) like Linux's ext4.

#### 2.1.a

Explain the difference between a hard link and a soft link in this file system. What is the length of the content of a soft link file?

A hard link is a link between the filename and the actual data stored in the filesystem. This means that every single file will start with a single hard link; it's filename as represented in the user interface.

One can add more hard links to a file. In practice this is to create a second filename which points to the exact same data as the old filename, in exactly the same on-disk location.

For a file to be deleted, all of these separate "filename-files" must be deleted. This means that deleting one of the hard links, will not delete the underlying file.

An illustration of the concept from , where there's two "files" as seen from the user in the file explorer, names LINK-A.txt and LINK-B.txt, which both "points" to the actual file on the disk.

In practice, one could say that the file is known to the filesystem by several names, i.e. the file have one or more "[alias](#)".

A soft link, also known as a symbolic link or "symlink", is the underlying mechanism for what is commonly known as a "shortcut". In contrast to the hardlink which "is" the file it links to, only by another name, the symlink is a special type of file that merely *points* to an existing file. To illustrate, if `Symlink_A` points to `file_A.txt`, deleting `Symlink_A` will not delete `file_A.txt`, since the file and the symlink are two separate files. If you have `Symlink_A` and `Symlink_B` both pointing to `file_A.txt`, deleting both symlinks (shortcuts, soft links) will have no effect on the file `file_A.txt`, and it will still be where it was on the file system. The soft links have no effect on the file to which it points, other than directing the user there from somewhere else. But if you delete `file_A.txt`, both of the symlinks will be so-called "dangling soft links", as they will point to something that doesn't exist anymore, i.e. nowhere.

The length of the content of a soft link file is typically the file path to which it points. This is because the function of a soft link is merely to act as a shortcut, to redirect any operations (e.g. the user "opening" a file) from the symlink to the target file or directory.

#### 2.1.b

What is the minimum number of references (hard links) for any given folder?

A folder (i.e. directory) cannot have any **user created** hard links. Most distributions will not allow this, even if the user tries to force it.

That said, there are hardlinks which are "built in". Any given directory will have minimum two hardlinks:

- One from its parent directory, via its "folder name".

- One link to itself, via the dot, `.` as seen when using `cd .` in UNIX. Essentially "change directory to where I'm currently at".

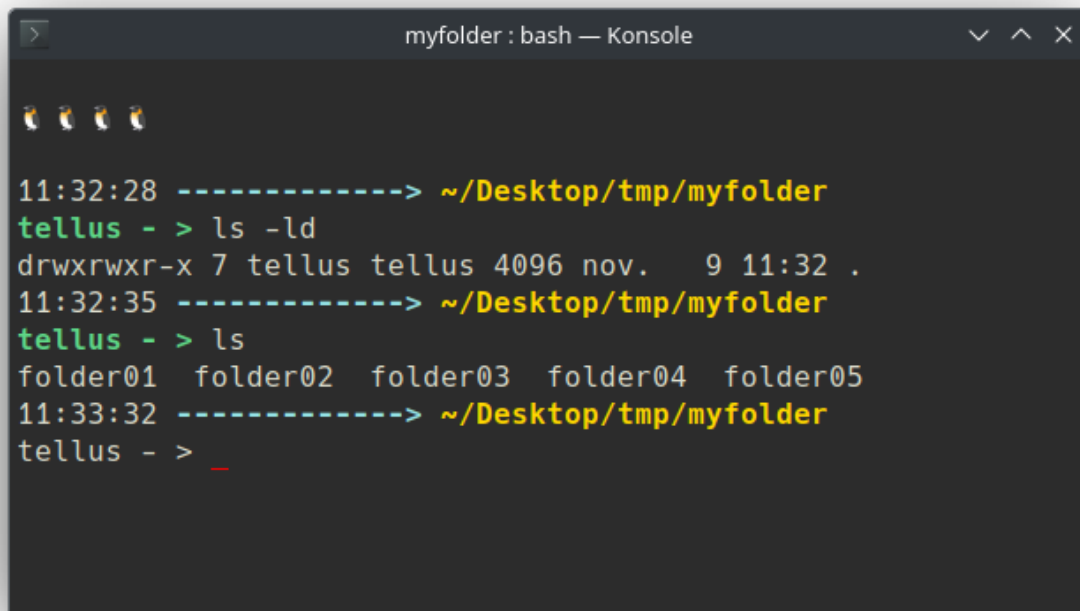
Additionally, there's the dot-dot `..` which links back to the parent directory.

## 2.1.c

Consider a folder `/tmp/myfolder` containing 5 subfolders. How many references (hard links) does it have? Try it yourself on a Linux system and include the output. Use `ls -ld /tmp/myfolder` to view the reference count (hint, it's the second column in the output).

The directory will have 7 hard links, or references. The total number of hard links in a directory will be 2 (itself and its parent) plus the number of subdirectories it contains given. In this case,  $2 + 5 = 7$ .

Screenshot of terminal output demonstrating the given example:



```
myfolder : bash — Konsole
11:32:28 -----> ~/Desktop/tmp/myfolder
tellus - > ls -ld
drwxrwxr-x 7 tellus tellus 4096 nov.  9 11:32 .
11:32:35 -----> ~/Desktop/tmp/myfolder
tellus - > ls
folder01  folder02  folder03  folder04  folder05
11:33:32 -----> ~/Desktop/tmp/myfolder
tellus - > _
```

These references are to itself, to its parent directory, and to its five child directories.

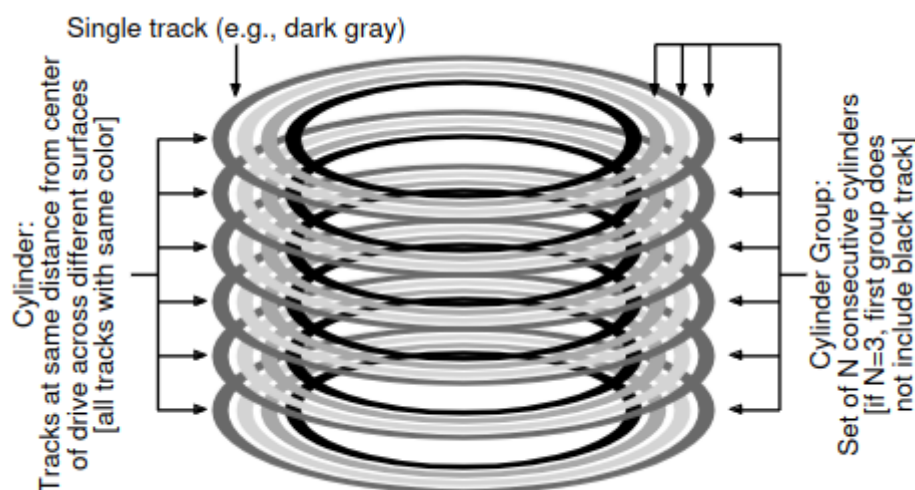
## 2.1.d

Explain how spatial locality is achieved in a FFS.

Spatial locality refers to the strategy of placing related data physically close together (literally) on the disk to reduce seek times and thus increase performance in access operations. This is achieved through several design choices on how to place data:

- **Cylinder groups:** The disk is organized into cylinder groups, which are clusters of cylinders<sup>1</sup> located close together on the disk.
  - <sup>1</sup>Yes, actual cylinders. On traditional (magnetic) hard disks, data is organized into "tracks", which are concentric circles around the disk platters. Each of the tracks is divided into sectors (degrees of a circle), which is the smallest unit of storage which can be read or written. A cylinder is a *set of tracks aligned vertically*, i.e. stacked on top of each other, through the layers of platters on the disk. All the tracks that can be accessed without repositioning the read/write head makes up the cylinder.
- **Block Allocation policy:** FFS uses a block allocation policy that aims to place the datablocks of a file in the same cylinder group as its inode. This is to reduce the distance the disk head needs to travel when accessing the entire content of a file.
- **Block and Fragment allocation:** Some more advanced systems can place fragments of datablock. This is useful for not wasting space, since the system can place parts of files in a block and not use the entire block for only a very small amount of data. This will also help keeping all files physically close, since the "spill-over" data can be placed in a block with other fragments, not having to be moved to an entirely new block which may be some distance away.
- **Sub-directory placement:** FFS attempt to allocate inodes for new files and directories in the same cylinder group as their parent directory. This is to keep the directory contents physically close on the disk, thus improving the performance of directory listings and file accesses within directories.
- **Data block placement:** When a file is created or extended, FFS attempts to place the data blocks near its inode and the other block of the file to reduce seek times. For larger files, spanning multiple cylinder groups, it will try to minimize the distance between groups.
- **Rotational layout:** For magnetic hard disk drives, FFS considers the rotational latency when placing data. It will try to place the data in such a way as to take into account the rotation of the disk, and arrange the data so that as the read/write head moves to the next sequential block, that same block will come under the head simply due to disk rotation.

Illustration of the cylinder group:



Most of these techniques were developed for the spinning drives (mechanical hard disk drives, HDD), which means that some are not as applicable for modern solid state drives (SSD). E.g. in an SSD, there's no spinning disks, and no R/W head which needs to be factored in. However, much of the theory and implementation still holds:

- Data is still organized into block, though not cylinders as there's no stack of disk nor a magnetic read/write head.
- Allocation policy does not need to account for R/W head movement, but allocation is still relevant for wear leveling<sup>2</sup> and efficient use of the NAND flash memory cells of the SSD.
  - <sup>2</sup>Wear leveling is the process of distributing write and erase cycles across the memory, to avoid using "the same area" over and over. Read/write/erase is still a physical process, incurring wear, even if the moving parts like R/W head and spinning disks are not present.
- Block and fragment allocation have less performance impact on an SSD, but are still important for efficient space utilization. Allocation strategy will impact the wear leveling and garbage collection process.

## Source references for part 2.1

- [RedHat.com - Hard links and soft links in Linux explained](#)
- [Wikipedia - Hard link](#)
- [Wikipedia - Symbolic link](#)
- [Wikipedia - Aliasing](#)
- [Everything Important You Need to Know About Hard Link in Linux](#)
- [Locality and The Fast File System](#)
- [A Fast File System for UNIX](#)

## 2.2 NTFS - Flexible tree with extents

### 2.2.a

Explain the differences and use of resident versus non-resident attributes in NTFS.

Attributes in the filesystem are metadata properties associated with files and directories, e.g. size, time stamps, permissions and similar. NTFS categorizes them as either resident or non-resident.

**"Resident"** indicates that the data *resides* directly within the Master File Table (MFT) record of the file. The data "is a resident" of the MFT, i.e. "it lives there". This makes these attributes easily accessible by the file system, letting them be read quickly without having to do additional searching and reading from other parts of the disk. Since resident data from all files are stored in the MFT, there are limitations on the size of them, which means only small data is placed there, like names and timestamps.

**"Non-resident"** data is the data that does not reside in the MFT record itself. Instead, it lives outside, in various locations on the disk. For these data, the MFT record will act as a reference guide, pointing to where the data is located.

The MFT contains a record of every file and directory on an NTFS volume. This is essential for the performance of the NT file system, since this allows it to quickly read essential attributes, and point to where files are located. It also plays a role in file recovery, since it contains a record of all files and their locations, even if they've been deleted but not overwritten by new data.

## 2.2.b

Discuss the benefits of NTFS-style extents in relation to blocks used by FAT or FFS.

"Extents", as used by NTFS, offers several advantages over the "blocks" used by FAT or FFS.

Extent is a contiguous range in disk sectors. Instead of managing individual blocks, NTFS can allocate large, contiguous regions of space ("extents") for files. This makes it able to reserve a contiguous space for larger files, instead of relying on blocks that require files to be broken into smaller chunks and placed around the disk where there's room. In FFS, these blocks are more efficient, with block groups and cylinder groups, but there's still a need to break up the data. Having the files located in the same region, makes Read/Write operations more efficient for these files and reduce fragmentation. Additionally, the extents are flexible in size, which means that as a file grows (or shrinks) the space allocated can be adjusted accordingly, which means that disk space is used more efficiently than if all data were assigned to fixed size blocks. A small file will not "take up" a large block but not fill it, and a large file will not fill half a block with overspill from the last block. Although FFS implements a somewhat similar feature, with the ability to place "tails" of files without allocating an entire block to that single tail, which makes for a bit more space efficiency and can save on fragmentation, NTFS have the advantage of keeping the entire file in "one place".

## 3. Copy-on-Write

Explain how copy-on-write (COW) helps guard against data corruption.

To guard against data corruption, Copy-on-Write is a technique for ensuring that data is not overwritten during updates or modification. The principle is based on that when a change is made to a file or data object, instead of altering the existing data, COW will create a copy (duplicate) of the data and apply the changes to this new version. The original data will remain unaltered until the new version have been fully and successfully copied and updated. This works really well to protect against partial updates or incomplete write operations, which may occur if the system experiences a crash or power failure during such operations. If that happens, the system will still have the original version, which will be intact and uncorrupted.

Additionally, COW betters the atomic side of write operations, in that updates are treated as atomic transactions. This means that updates or modifications are completed in their entirety or not at all. By copying the original file, and applying the full update to the copy before committing this copy back to replace the original file, COW eliminates the risk of having data be left in a partially updated state.

## 3 - Security

---

### 3.1. Authentication

#### 3.1.a

Why is it important to hash passwords with a unique salt, even if the salt can be publicly known?

By hashing passwords with a **unique** salt, the hash value will be unique for every password+salt combination. By adding a unique salt to each password, randomness will be introduced into the hash value, which in turn will ensure that every password+salt combination represents a unique string, even if there are many passwords that are identical. Note that the salt **must** be unique for every password. If not, then we're back to hashing identical strings, which is relatively easy to reverse engineer or brute-force.

When running a hashing algorithm on the **unique** password+salt combination, the result will be a unique hash for every combination.

The salt can be publicly known, because there is no way to reverse engineer the final hash back to the password+salt combo. The only way would be to try a brute-force-approach using both the salt table *and* conventional password brute force or dictionary attack, which is very time consuming and in most cases a lost cause.

#### 3.1.b

Explain how a user can use a program to update the password database, while at the same time does not have read or write permissions to the password database file itself. What are the caveats of this?

A user can update the password database without direct read/write permissions to the file by interacting with it through a privileged program or service that manages the permissions required. This program will often be running with elevated privileges and acts as an intermediary between the password database and the user. Such programs would likely be running as root on Linux. A user will send a request to change the password through the program, which will authenticate the user, hash the new password and then update the database. The actual database, e.g. `/etc/shadow` on Linux, is inaccessible to normal users.

**Privileged program:** Program or service running with elevated privileges, usually as root user, with necessary read/write access to the password database. Acts as a middle layer, managing interactions between user and password database.

**Controlled access:** The user interacts with the above mentioned program, not directly with the password database file. The user can send requests to update a password, which is then processed by the program.

**Validation and processing:** The program will then validate the users identity, e.g. by requesting the user to supply the root (admin) user password, and then process the requested password change. The password is salted, hashed and updated in the password database.

The caveats of this approach is the reliance on the integrity and security of the intermediary program. If this program have vulnerabilities, i.e. security holes, it could be exploited by malicious actors or processes to gain unauthorized access to the password database. Also, since users aren't allowed to interact directly with the database, they're limited to the actions allowed by the program, which may or may not be less than ideal, depending on how one looks at it. This system also requires a lot of attention and well thought-out design in order for it to remain secure and easily maintained.

**Program security:** How secure this mechanism/process is, depends on the security of the privileged program itself. If there are vulnerabilities that can be exploited, a malicious actor or rogue process can use this to gain unauthorized access to the password database.

**Program integrity:** For the security to be good, the software integrity is crucial, and the maintainers must always be on top of maintenance to ensure any and all security holes are found and plugged.

**Error handling:** Error handling must be very robust, and handled in a secure manner. E.g. it is imperative that error messages does not expose sensitive information in any way, or than thrown error opens up an attack vector which can be exploited.

**Limited control:** Since interaction must be done through the privileged program/service, the user is limited in their ability to interact with the database within the restrictions set by the program. This makes for better security, and protects the database from user error or users who simply don't know what they're doing, but may be restrictive to user that do know what they are doing.

## 3.2. Software vulnerabilities

### 3.2.a

Describe the problem with the well-known `gets()` library call. Name another library call that is safe to use that accomplishes the same thing.

The `gets()` function is known for being inherently unsafe due to its inability to prevent buffer overflow, as demonstrated during a lecture.

It reads a line from `stdin` and stores it in the buffer pointed to by its argument. The problem arises in that `gets()` does not check the size of the buffer, which means it will continue to read and store characters beyond the buffer's capacity, if the input is too long. This may write into the adjacent memory space, which in turn may lead to "unpredictable behavior". "Unpredictable behavior" is a catch-all term used for when "unintended things can be done", which includes everything from the system simply crashing to more serious issues like exposing restricted regions of memory, exposing security vulnerabilities or the unintended execution of malicious code, here by way of injecting it into the new area "unlocked" by overflowing the buffer.



The safe(r) alternative to `gets()` is `fgets()`. It will take an additional argument which specifies the maximum number of characters to be read, including the null terminator. This will make `fgets()` prevent buffer overflow by ensuring that it does not read more characters than the buffer can hold.

Example:

```
char buffer[100];
fgets(buffer, sizeof(buffer), stdin);
```

In this example, `fgets()` reads up to 99 characters from `stdin` and stores them in `buffer`, reserving the final character for the null terminator.

### 3.2.b

Explain why a microkernel is statistically more secure than a monolithic kernel.

The main reason for microkernels being considered more secure than a monolithic kernel, is due to its "components" being separate from each other. This means there's much less interdependencies between the internal workings of the microkernel than compared to a monolithic kernel.

Functionality is compartmentalized, and security holes or instabilities are less prone to spill over to other areas of the kernel. Instead of one component service crashing and taking down the entire house, in a microkernel there's more chance that only the component itself experiences the crash, leaving the kernel structure as a whole still operating.

**Minimal vs broad functionality:** Microkernels provide the bare minimum of essential functionality, e.g. software-hardware communication and basic process and memory management. Most other services, like drivers and file systems, run in userspace. A monolithic kernel, on the other hand, will run a wide range of its functionality directly in the kernel, including drivers, file system management and network stacks.

**Isolation vs not isolated:** Since many services, like drivers and file systems, run in userspace, a security breach in them are less likely to expose the kernel space, as they are isolated from it. For a monolithic architecture, all these are running in kernel space, and a bug or vulnerability in any of them may compromise the entire system, since they provide a much larger attack surface.

**Simplicity vs complexity:** Microkernels have more simplicity, as in a smaller code base, since all services are separate. A smaller code footprint makes it easier to keep on top of the code, making it easier to maintain and less likely to contain bugs (simply due to there being less code for bugs to appear in). A monolithic kernel will have a much larger code base, making it harder to maintain and audit. Higher complexity is usually correlated with an increase in bugs and vulnerabilities, since there's that much more code and interdependencies to keep track of.

**Fault isolation:** Since services are isolated, there is less chance for a crash in a single component to also take down other components. This reduces the chance for a system wide crash or security breaches.

**Reduced privileges:** The separate services of a microkernel will each run with less privilege than in a monolithic kernel, which limits the potential for damage if compromised.

**Modularity:** Tying into the point about simplicity, each component is a "module" of a larger construct. This ability to scrutinize and update modules separately, without necessarily affecting the entire kernel structure, makes it easier to ensure module and system security and integrity.