Author: Ramon Andrade Sousa Carvalho

1) System Software

System software is software that provides a platform for other software [1]. Some examples can be operating systems, antivirus software, disk formatting software, computer language translators, etc [1].

2) System Calls

System calls, also known as *syscalls*, are how a program enters the kernel to perform some task [2]. Programs use system calls to perform a variety of operations such as: creating processes, doing network and file IO, and much more [2].

A system call is a request made by a user-level program to the OS to perform a privileged operation [3]. To make a system call, the program executes a special instruction that triggers a software interrupt; the OS then transfers control to the kernel and executes a system call handler; the system call handler checks the type of system call and takes appropriate action, such as reading from a file or allocating memory [3].

4) Traps

Traps and system calls are two mechanisms used by an operating system (OS) to perform privileged operations and interact with user-level programs [3].

A trap is an interrupt generated by the CPU when a user-level program attempts to execute a privileged instruction or encounters an error. When a trap occurs, the CPU transfers control to the kernel and executes a trap handler. The trap handler checks the type of trap and takes appropriate action, such as terminating the program or performing a privileged operation on behalf of the program [3].

Traps and system calls are similar in that they both involve transferring control to the kernel to perform privileged operations. However, traps are usually generated automatically by the CPU when a program encounters an error or attempts to execute a privileged instruction, while system calls are initiated by the program itself to request privileged operations [3].

3) C Library

The GNU C Library project provides the core libraries for the GNU system and GNU/Linux systems, as well as many other systems that use Linux as the kernel. These libraries provide critical APIs including ISO C11, POSIX.1-2008, BSD, OS-specific APIs and more [4]. These APIs include such foundational facilities as *open*, *read*, *write*, *malloc*, *printf*, *getaddrinfo*, *dlopen*, *pthread\_create*, *crypt*, *login*, *exit* and more [4].

4) API

API stands for *application programming interface*, which is a set of definitions and protocols for building and integrating application software [5].

APIs are sometimes thought of as contracts, with documentation that represents an agreement between parties [5]. APIs let your product or service communicate with other products and services without having to know how they’re implemented [5]; providing flexibility; simplifying design, administration, and use [5].

All the programs in C use functions from the standard libraries of the language. The set of functions of each library is described in an interface file, or header file [6] – which is an API.

5) ABI

In computer software, an application binary interface (ABI) is an interface between two binary program modules [7]. Often, one of these modules is a library or operating system facility, and the other is a program that is being run by a user [7]. An ABI defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format [7].

A common aspect of an ABI is the calling convention, which determines how data is provided as input to, or read as output from, computational routines [7]. Examples of this are the x86 calling conventions [7].

Adhering to an ABI (which may or may not be officially standardized) is usually the job of a compiler, operating system, or library author [7]. However, an application programmer may have to deal with an ABI directly when writing a program in a mix of programming languages, or even compiling a program written in the same language with different compilers [7].

An ABI is as important as the underlying hardware architecture [7]. The program will fail equally if it violates any constraints of these two [7].

For instance, a library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile [8]. Binary compatibility saves a lot of trouble [8]. It makes it much easier to distribute software for a certain platform. Without ensuring binary compatibility between releases, people will be forced to provide statically linked binaries [8].

Other example, the CPUs found in Apple products define rules for how to call functions, manage the stack, and perform other operations [9]. If your code includes assembly instructions, you must adhere to these rules to operate correctly with compiler-generated code [9]. If you don’t adhere to them, your code might crash or behave unexpectedly [9].

6) POSIX

Short for "Portable Operating System Interface for UNIX", POSIX is a set of standards codified by the IEEE and issued by ANSI and ISO [10]. The goal of POSIX is to ease the task of cross-platform software development by establishing a set of guidelines for operating system vendors to follow [10].

Ideally, a developer should have to write a program only once to run on all POSIX-compliant systems [10]. Most modern commercial Unix implementations and many free ones are POSIX compliant [10]. There are actually several different POSIX releases, but the most important are POSIX.1 and POSIX.2, which define system calls and command-line interface, respectively [10].

7) “Everything is a file”

“Everything is a file” is the concept of how the system records and keeps everything in its structure [11]. Unlike other operating systems, UNIX-like operating systems treat everything as a common file [11].

his means that not only partitions are mounted as files but directories of specific devices such as RAM, smartphones, external disks, and optical media discs are all files [11]. Besides these, sockets and pipes are also files [11]. In addition, each TTY terminal – the ones you open with CTRL ALT F1 – are “files” in the eyes of the system [11].

8) Paths

On Linux, there are absolute paths and there are relative paths. The way to differentiate them is looking for a backslash at the beginning of the path: if there is one (“/path/to/file”), it is an absolute path, otherwise, it is a relative path (“path/to/file”) [12].

9) Files, Directories and Links

In Linux,

* Files are treated as bytes of data in a byte stream, no structure enforced at the system level [12].
* Files can be opened more than once, by different or same process [12].
* Files are referenced and accessed by *inodes* in the filesystem [12].
* Directories maps human-readable names to inode numbers [12].
* A directory is also a file, with their own inodes [12].
* Links allows different file names to share the same content without duplicating inode content [12].
* There are 2 types of links: Hard Links and Symbolic (or Soft) Links (*symlinks*) [12].
* Symbolic links are not updated (they merely contain a string which is the path name of its target); hard links always refer to the source, even if moved or removed [13].

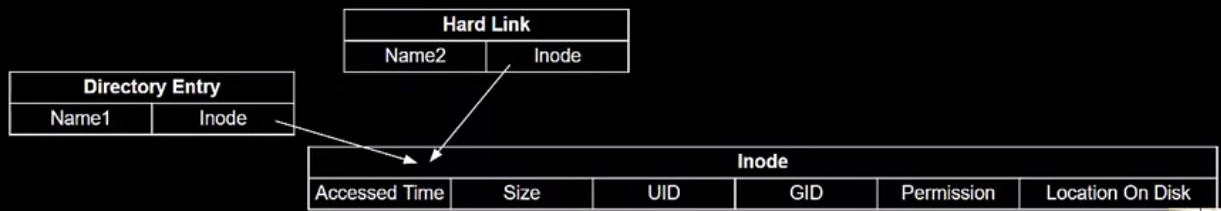


Figure 1. ilustration showing a same file being referenced by a directory entry and a link [12]

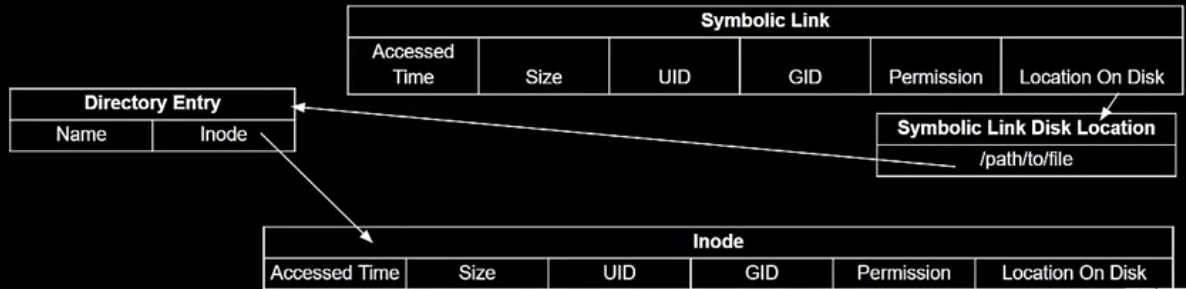


Figure 2. ilustration showing a same file being referenced by a directory entry and a symbolic link [12]

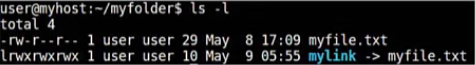


Figure 3. Link example [12]

10) Linux Inode (Index Node)

An inode is a data structure that keeps track of all the files and directories within a Linux or UNIX-based filesystem [14]. So, every file and directory in a filesystem is allocated an inode, which is identified by an integer known as “inode number”. These unique identifiers store metadata about each file and directory [14].

It is important to keep an eye on them to avoid issues related to inode shortage or excessive usage [14].

All inodes within the same filesystem are unique [14]. However, the same inode number can be used in different filesystems [14].