The Linux Concept Journey

version 3.0
September-2023

By Dr. Shlomi Boutnaru

Introduction

When starting to learn Linux I believe that they are basic concepts that everyone needs to know about. Because of that I have decided to write a series of short writeups aimed at providing the basic vocabulary and understanding for achieving that.

Overall, I wanted to create something that will improve the overall knowledge of Linux in writeups that can be read in 1-3 mins. I hope you are going to enjoy the ride.

Lastly, you can follow me on twitter - @boutnaru (https://twitter.com/boutnaru). Also, you can read my other writeups on medium - https://medium.com/@boutnaru.

Lets GO!!!!!!

The Auxiliary Vector (AUXV)

There are specific OS variables that a program would probably want to query such as the size of a page (part of memory management - for a future writeup). So how can it be done?

When the OS executes a program it exposes information about the environment in a key-value data store called "auxiliary vector" (in short auxv/AUXV). If we want to check which keys are available we can go over[1] (on older versions it was part of elf.h) and look for all the defines that start with "AT_".

Among the information that is included in AUXV we can find: the effective uid of the program, the real uid of the program, the system page size, number of program headers (of the ELF - more on that in the future), minimal stack size for signal delivery (and there is more).

If we want to see all the info of AUXV while running a program we can set the LD_SHOW_AUXV environment variable to 1 and execute the requested program (see the screenshot below, it was taken from JSLinux running Fedora 33 based on a riscv64 CPU[2]. We can see that the name of the variable starts with "LD_", it is because it is used/parsed by the dynamic linker/loader (aka ld.so).

Thus, if we statically link our program (like using the -static flag on gcc) setting the variable won't print the values of AUXV. Anyhow, we can also access the values in AUXV using the "unsigned long getauxval(unsigned long type)" library function[3]. A nice fact is that the auxiliary vector is located next to the environment variables check the following illustration[4].

```
[root@localhost ~]# export LD_SHOW_AUXV=1
[root@localhost ~]# uname -a
AT_SYSINFO_EHDR:      0x200001f000
AT_HWCAP:             112d
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x2aaaaaa040
AT_PHENT:             56
AT_PHNUM:             9
AT_BASE:              0x2000000000
AT_FLAGS:             0x0
AT_ENTRY:             0x2aaaaabd38
AT_UID:               0
AT_EUID:              0
AT_GID:               0
AT_EGID:              0
AT_SECURE:            0
AT_RANDOM:            0x3fffff6821
AT_EXECFN:            /bin/uname
Linux localhost 4.15.0-00049-ga3b1e7a-dirty #11 Thu Nov 8 20:30:26 CET 2018 risc
v64 riscv64 riscv64 GNU/Linux
```

---

[1] https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/auxvec.h#L10
[2] https://bellard.org/jslinux/
[3] https://man7.org/linux/man-pages/man3/getauxval.3.html
[4] https://static.lwn.net/images/2012/auxvec.png
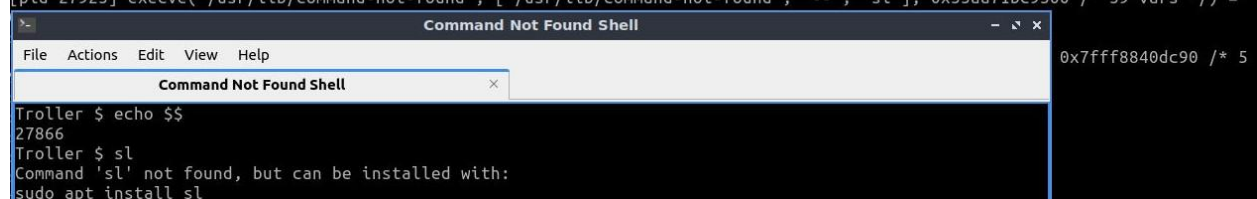
command not found

Have you ever asked yourself what happens when you see "command not found" on bash? This writeup is not going to talk about that and not about the flow which determines if a command is found or not (that is a topic for a different write up ;-).

I am going to focus my discussion on what happens in an environment based on bash + Ubuntu (version 22.04). I guess you at least once wrote "sl" instead of "ls" and you got a message "Command 'sl' not found, but can be installed with: sudo apt install sl" - how did bash know that there is such a package that could be installed? - as shown in the screenshot below

Overall, the magic happens with the python script "/usr/lib/command-not-found" which is executed when a bash does not find a command - as shown in the screenshot below. This feature is based on an sqlite database that has a connection between command and packages, it is sorted in "/var/lib/command-not-found/commands.db".

Lastly, there is a nice website https://command-not-found.com/ which allows you to search for a command and get a list of different ways of installing it (for different Linux distributions/Windows/MacOS/Docker/etc).

Out-of-Memory Killer (OOM killer)

The Linux kernel has a mechanism called "out-of-memory killer" (aka OOM killer) which is used to recover memory on a system. The OOM killer allows killing a single task (called also oom victim) while that task will terminate in a reasonable time and thus free up memory.

When OOM killer does its job we can find indications about that by searching the logs (like /var/log/messages and grepping for "Killed"). If you want to configure the "OOM killer"[5].

It is important to understand that the OOM killer chooses between processes based on the "oom_score". If you want to see the value for a specific process we can just read "/proc/[PID]/oom_score" - as shown in the screenshot below. If we want to alter the score we can do it using "/proc/[PID]/oom_score_adj" - as shown also in the screenshot below. The valid range is from 0 (never kill) to 1000 (always kill), the lower the value is the lower is the probability the process will be killed[6].

```
root@localhost:~# cat /proc/1/oom_score
0
root@localhost:~# cat /proc/self/oom_score
667
```

---

[5] https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-oom-killer.html
[6] https://man7.org/linux/man-pages/man5/proc.5.html

Why doesn't "ltrace" work on new versions of Ubuntu?

Many folks have asked me about that, so I have decided to write a short answer about it. Two well known command line tools on Linux which can help with dynamic analysis are "strace" and "ltrace". "strace" allows tracing of system calls ("man 2 syscalls") and signals ("man 7 signal"). I am not going to focus on "strace" in this writeup, you can read more about it using "man strace". On the other hand, "ltrace" allows the tracing of dynamic library calls and signals received by the traced process ("man 1 ltrace"). Moreover, it can also trace syscalls (like "strace") if you are using the "-S" flag.

If you have tried using "ltrace" in the new versions of Ubuntu you probably saw that the library calls are not shown (you can verify it using "ltrace `which ls`"). In order to demonstrate that I have created a small c program - as you can see in the screenshot below ("code.c").

First, if we compile "code.c" and run it using "ltrace" we don't get any information about a library call (see in the screenshot below). Second, if we compile "code.c" with "-z lazy" we can see when running the executable with "ltrace" we do get information about the library functions. So what is the difference between the two?

"ltrace" (and "strace") works by inserting a breakpoint[7] in the PLT for the relevant symbol (that is library function) we want to trace. So because by default the binaries are not compiled with "lazy loading" of symbols they are resolved when the application starts and thus the breakpoints set by "ltrace" are not triggered (and we don't see any library calls in the output - as shown in the screenshot below).   Also, you can read more about the internals of "ltrace" here - https://www.kernel.org/doc/ols/2007/ols2007v1-pages-41-52.pdf

---

[7]https://medium.com/@boutnaru/have-you-ever-asked-yourself-how-breakpoints-work-c72dd8619538

vDSO (Virtual Dynamic Shared Object)

vDSO is a shared library that the kernel maps into the memory address space of every user-mode application. It is not something that developers need to think about due to the fact it is used by the C library[8].

Overall, the reason for even having vDSO is the fact they are specific system calls that are used frequently by user-mode applications.Due to the time/cost of the context-switching between user-mode and kernel-mode in order to execute a syscall it might impact the overall performance of an application.

Thus, vDSO provides "virtual syscalls" due to the need for optimizing system calls implementations. The solution needed to not require libc to track CPU capabilities and or the kernel version. Let us take for example x86, which has two ways of invoking a syscall: "int 0x80" or "sysenter". The "sysenter" option is faster, due to the fact we don't need to through the IDT (Interrupt Descriptor Table). The problem is it is supported for CPUs newer than Pentium II and for kernel versions greater than 2.6[9].

If you vDSO the implementation of the syscall interface is defined by the kernel in the following manner. A set of CPU instructions formatted as ELF is mapped to the end of the user-mode address space of all processes - as shown in the screenshot below. When libc needs to execute a syscall it checks for the presence of vDSO and if it is relevant for the specific syscalls the implementation in vDSO is going to be used - as shown in the screenshot below[10].

Moreover, for the case of "virtual syscalls" (which are also part of vDSO) there is a frame mapped as two different pages. One in kernel space which is static/"readable & writeable" and the second one in user space which is marked as "read-only". Two examples for that are the syscalls "getpid()" (which is a static data example) and "gettimeofday()" (which is a dynamic read-write example).

Also, as part of the kernel compilation process the vDSO code is compiled and linked. We can most of the time find it using the following command "find arch/$ARCH/ -name '*vdso*.so*' -o -name '*gate*.so*'"[11]
If we want to enable/disable vDSO we can set "/proc/sys/vm/vdso_enable" to 1/0 respectively[12]. Lastly, a benchmark of different syscalls using different implementations is shown below.

[8] https://man7.org/linux/man-pages/man7/vdso.7.html
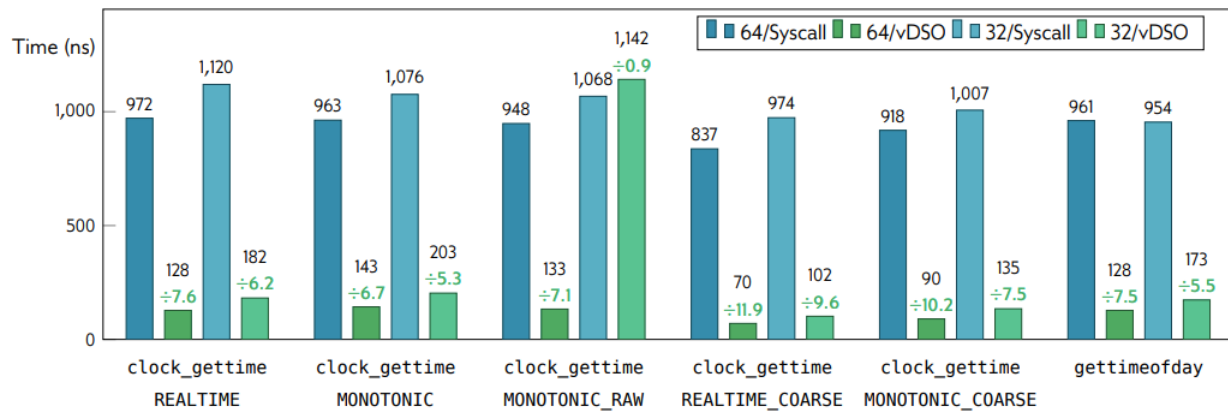[9] https://linux-kernel-labs.github.io/refs/heads/master/so2/lec2-syscalls.html
[10] https://hackmd.io/@sysprog/linux-vdso
[11] https://manpages.ubuntu.com/manpages/xenial/man7/vdso.7.html
[12] https://talk.maemo.org/showthread.php?t=32696

Kernel and userspace setup

```
execve()
   │
   ▼
┌─────────────┐      ┌─────────────────────────────────────────────────┐
│ ELF loader  │------│ ■ Maps vDSO pages (code + data)                  │
└─────────────┘      │ ■ Sets AT_SYSINFO_EHDR in the auxiliary vector   │
                     └─────────────────────────────────────────────────┘
KERNEL
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
USERSPACE
   │
   ▼
┌─────────────────┐  ┌─────────────────────────────────────────────────┐
│ Dynamic linker  │--│ ■ Looks up AT_SYSINFO_EHDR in the auxiliary vector│
└─────────────────┘  │ ■ If set, links the vDSO (→ [vdso])              │
                     └─────────────────────────────────────────────────┘
   │
   ▼
┌─────────────┐      ┌─────────────────────────────────────────────────┐
│ libc init   │------│ ■ Looks up function symbols (e.g. __vdso_gettimeofday)│
└─────────────┘      │     in [vdso]                                    │
                     │ ■ If found, sets global function pointers        │
                     └─────────────────────────────────────────────────┘
```

ARM

https://www.slideshare.net/vh21/twlkhlinuxvsyscallandvdso
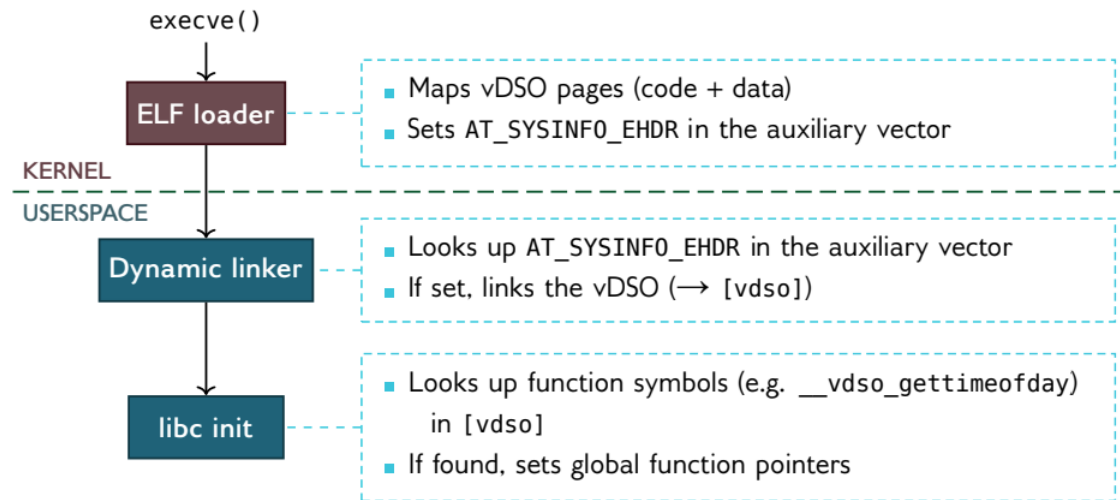
Calling syscalls from Python

Have you ever wanted a quick way to call a syscall (even if it is not exposed by libc)? There is a quick way of doing that using "ctypes" in Python.

We can do it using the "syscall" exported function by libc (check out 'man 2 syscall'[13] for more information). By calling that function we can call any syscall by passing its number and parameters.

How do we know what the number of the syscall is? We can just check https://filippo.io/linux-syscall-table/. What about the parameters? We can just go the the source code which is pointed in any entry of a syscall (from the previous link) or we can just use man (using the following pattern - 'man 2 {NameOfSyscall}', for example 'man 2 getpid').

Let us see an example, we will use the syscall getpid(), which does not get any arguments. Also, the number of the syscall is 39 (on x64 Linux). You can check the screenshot below for the full example. By the way, the example was made with https://www.tutorialspoint.com/linux_terminal_online.php and online Linux terminal (kernel 3.10).

```
$ python3
Python 3.8.6 (default, Jan 29 2021, 17:38:16)
[GCC 8.4.1 20200928 (Red Hat 8.4.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ctypes
>>> libc=ctypes.CDLL(None)
>>> libc.syscall(39)
47617
```

---

[13] https://man7.org/linux/man-pages/man2/syscall.2.html

Syscalls' Naming Rule: What if a syscall's name starts with "f"?

Due to the large number of syscalls, there are some naming rules used in order to help in understanding the operation performed by each of them. Let me go over some of them to give more clarity.

If we have a syscall "<syscall_name>" so we could also have "f<syscall_name>" which means that "f<syscall_name>" does the same operation as "<syscall_name>" but on a file referenced by an fd (file descriptor). Some examples are ("chown", "fchown") and ("stat", "fstat"). It is important to understand that not every syscall which starts with "f" is part of such a pair, look at "fsync()" as an example, however in this case the prefix still denoting the input of the syscall is an fd. There are also examples in which the "f" prefix does not even refer to an fd like in the case of "fork()", it is just part of the syscall name.

Syscalls' Naming Rule: What if a syscall's name starts with "l"?

I want to talk about those syscalls starting with "l". If we have a syscall "<syscall_name>" so we could also have "l<syscall_name>" which means that "l<syscall_name>" does the same operation as "<syscall_name>" but in case of a symbolic link given as input the information is retrieved about the link itself and not the file that the link refers to (for example "getxattr" and "lgetxattr". Moreover, not every syscall that starts with "l" falls in this category (think about "listen").

I think the last rule is the most confusing one because there are cases in which the "l" prefix is not part of the original name of the syscall and is not relevant to any type of links. Let us look at "lseek", the reason for having the prefix is to emphasize that the offset is given as long as opposed to the old "seek" syscall.

RCU (Read Copy Update)

Because there are multiple kernel threads (check it out using 'ps -ef | grep rcu` - the output of the command is included in the screenshot at the end of the post) which are based on RCU (and other parts of the kernel) . I have decided to write a short explanation about it.

RCU is a sync mechanism which avoids the use of locking primitives in case multiple execution flows that read and write specific elements. Those elements are most of the times linked by pointers and are part of a specific data structure such as: has tables, binary trees, linked lists and more.

The main idea behind RCU is to break down the update phase into two different steps: "reclamation" and "removal" - let's detail those phases. In the "removal" phase we remove/unlink/delete a reference to an element in a data structure (can be also in case of replacing an element with a new one). That phase can be done concurrently with other readers. It is safe due to the fact that modern CPUs ensure that readers will see the new or the old data but not partially updated. In the "reclamation" step the goal is to free the element from the data structure during the removal process. Because of that this step can disrupt a reader which references that specific element. Thus, this step must start only after all readers don't hold a reference to the element we want to remove.

Due to the nature of the two steps an updater can finish the "removal" step immediately and defer the "reclamation" for the time all the active during this phase will complete (it can be done in various ways such as blocking or registering a callback).

RCU is used in cases where read performance is crucial but can bear the tradeoff of using more memory/space. Let's go over a sequence of an update to a data structure in place using RCU. First, we just create a new data structure. Second, we copy the old data structure into the new one (don't forget to save the pointer to the old data structure). Third, alter the new/copied data structure. Fourth, update the global pointer to reference the new data structure. Fifth, sleep until the kernel is sure they are no more readers using the old data structure (called also grace period, in Linux we can use synchronize_rcu()[14].

In summary, RCU is probably the most common "lock-free" technique for shared data structures. It is lock-free for any number of readers. There are implementations also for single-writer and even multi-writers (However, it is out of scope for now). Of course, RCU also has problems and it is not designed for cases in which there are update-only scenarios (it is better for "mostly-read" and "few-writes") - More about that in a future writeup.

---

[14] https://elixir.bootlin.com/linux/latest/source/kernel/rcu/tree.c#L3796

```
root             3         2  0 07:15 ?        00:00:00 [rcu_gp]
root             4         2  0 07:15 ?        00:00:00 [rcu_par_gp]
root            10         2  0 07:15 ?        00:00:00 [rcu_tasks_rude_]
root            11         2  0 07:15 ?        00:00:00 [rcu_tasks_trace]
root            13         2  0 07:15 ?        00:00:02 [rcu_sched]
```
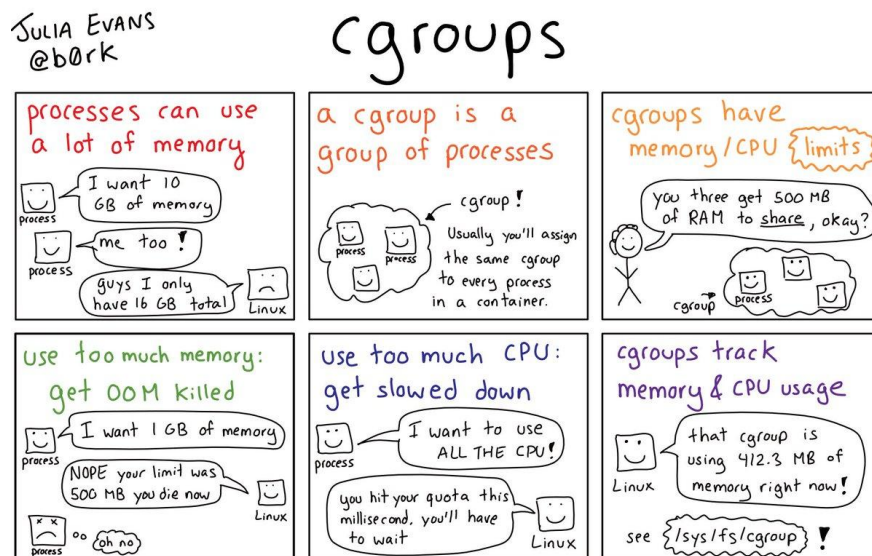
cgroups (Control Groups)

"Control Groups" (aka cgroups) is a Linux kernel feature that organizes processes into hierarchical groups. Based on those groups we can limit and monitor different types of OS resources. Among those resources are: disk I/O usage , network usage, memory usage, CPU usage and more (https://man7.org/linux/man-pages/man7/cgroups.7.html). cgroups are one of the building blocks used for creating containers (which include other stuff like namespaces, capabilities and overlay filesystems).

The cgroups functionality has been merged into the Linux kernel since version 2.6.24 (released in January 2008). Overall, cgroups provide the following features: resource limiting (as explained above), prioritization (some process groups can have larger shares of resources), control (freezing group of processes) and accounting[15].

Moreover, there are two versions of cgroups. cgroups v1 was created by Paul Menage and Rohit Seth. cgroups v2 was redesigned and rewritten by Tejun Heo[16]. The documentation for cgroups v2 first appeared in the Linux kernel 4.5 release on March 2016[17].

I will write on the differences between the two versions and how to use them in the upcoming writeups. A nice explanation regarding the concept of cgroups is shown in the image below[18]. By the way, since kernel 4.19 OOM killer[19] is aware of cgroups, which means the OS can kill a cgroup as a single unit.

---

[15] https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html
[16] https://www.wikiwand.com/en/Cgroups
[17]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/diff/Documentation/cgroup-v2.txt?id=v4.5&id2=v4.4
[18] https://twitter.com/b0rk/status/1214341831049252870
[19] https://medium.com/@boutnaru/linux-out-of-memory-killer-oom-killer-bb2523da15fc

Package Managers

Package manager (aka "Package Management System") is a set of software components which are responsible for tracking what software artifacts (executables, scripts, shared libraries and more). Packages are defined as a bundle of software artifacts that can be installed/removed as a group (https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html).

Thus, we can say that a package manager automates the installation/upgrading/removing of computer programs from a system in a consistent manner. Moreover, package managers often manage a database that includes the dependencies between the software artifacts and version information in order to avoid conflicts (https://en.wikipedia.org/wiki/Package_manager).

Basically, there are different categories of package managers. The most common are: OS package managers (like dpkg, apk, rpm, dnf, pacman and more - part are only frontends as we will describe in the future) and runtime package managers focused on specific programming languages (like maven, npm, PyPi, NuGet, Composer and more). Each package manager can also have its own package file format (more on those in future writeups). Moreover, package managers can have different front-ends CLI based or GUI based. Those package managers can also support downloading software artifacts from different repositories (https://devopedia.org/package-manager).

Overall, package managers can store different metadata for each package like: list of files, version, authorship, licenses, targeted architecture and checksums. An overview of the package management flow is shown in the diagram below (https://developerexperience.io/articles/package-management).
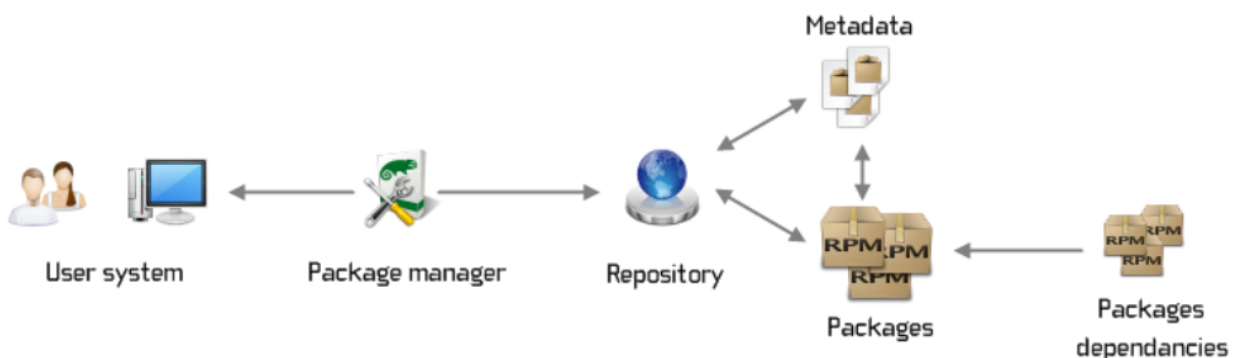
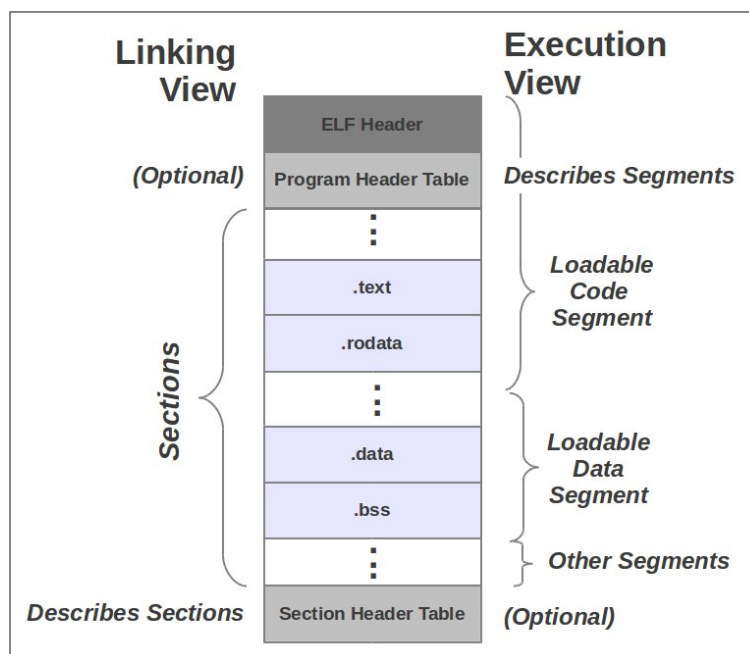# What is an ELF (Executable and Linkable Format) ?

Every generic/standard operating system has a binary format for its user mode executables/libraries, kernel code and more. Windows has PE (Portable Executable), OSX has MachO and Linux has ELF. We are going to start with ELF (I promise to go over the others also).

In Linux ELF among the others (but not limited to) for executables, kernel models, shared libraries, core dumps and object files. Although, Linux does not mandates an extension for files ELF files may have an extension of *.bin, *.elf, *.ko, *.so, *.mod, *.o, *.bin and more (it could also be without an extension).

Moreover, today ELF is a common executable format for a variety of operating systems (and not only Linux) like: QNX, Android, VxWorks, OpenBSD, NetBSD, FreeBSD, Fuchsia, BeOS. Also, it is used in different platforms such as: Wii, Dreamcast and Playstation Portable.

ELF, might include 3 types of headers: ELF header (which is mandatory), program headers and sections header . The appearance of the last two is based on the goal of the file: Is it for linking only? Is it execution only? Both? (More on the difference between the two in the next chapters). You can see the different layouts of ELF in the image below[20].

In the next parts we will go over each header in more detail. By the way, a great source for more information about ELF is man ("man 5 elf").

---

[20] https://i.stack.imgur.com/RMV0g.png

The ELF (Executable and Linkable Format) Header

Now we are going to start with the ELF header. Total size of the header is 32 bytes. The header starts with the magic "ELF" (0x7f 0x45 0x4c 0x46).

From the information contained in the header we can answer the following questions: Is the file 32 or 64 bit? Does the file store data in big or little endian? What is the ELF version? The type of the file (executable,relocatable,shared library, etc)? What is the target CPU? What is the address of the entry point? What is the size of the other headers (program/section)? - and more.

If we want to parse the header of a specific ELF file we can use the command "readelf" (we are going to use it across all of the next parts to parse ELFs). In order to show the header of an ELF file we can run "readelf -h {PATH_TO_ELF_FILE}". In the image below we can see the ELF header of "ls". The image was taken from an online Arch Linux in a browser (copy.sh).

```
root@localhost:~# readelf -h `which ls`
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x3c60
  Start of program headers:          52 (bytes into file)
  Start of section headers:          156320 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         12
  Size of section headers:           40 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

File System Hierarchy in Linux

As it turns out, there is a standard which is a reference that describes the conventions used by Unix/Linux systems for the organization and layout of their filesystem. This standard was created about 28 years ago (14 Feb 1994) and the last version (3.0) was published 7 years ago (3 Jun 2015). If you want to go over the specification for more details use the following link - https://refspecs.linuxfoundation.org/fhs.shtml.

We are going to give a short description for each directory (a detailed description for some of them will be in a dedicated write-up). We are going to list all the directories based on a lexicographic order . All the examples that I am going to share are based on a VM running Ubuntu 22.04.1 (below is a screenshot showing the directories for that VM). So let the fun begin ;-)

 "/", is the root directory of the entire system (the start of everything).
"/bin", basic command mostly binaries (there are also scripts like zgrep) that are needed for every user. Examples are: ls, ip and id.
"/boot", contains files needed for boot like the kernel (vmlinuz), initrd and boot loader configuration (like for grub). It may also contain metadata information about the kernel build process like the config that was used (A detailed writeup is going to be shared about "/boot" in the future) .
"/dev", device files, for now you should think about it as an interface to a device driver which is located on a filesystem (more on that in the future). Examples are: /dev/null, /dev/zero and /dev/random.
"/etc", contains configuration about the system or an installed application. Examples are: /etc/adduser.conf (configuration file for adduser and addgroup commands) and /etc/sudo.conf.
"/home", is the default location of the users' home directory (it can be modified in /etc/passwd per user). The directory might contain personal settings of the user, saved files by the user and more. Example is .bash_history which is a hidden file that contains the historical commands entered by the user (while using the bash shell).
"/lib", contains libraries needed by binaries mostly (but not limited to) in "/bin" and "/sbin". On 64 bit systems we can also have "lib64".
"/media", used as a mount point for removable media (like CD-ROMs and USBs).
"/mnt", can be used for temporary mounted filesystems.
"/opt", should include applications installed by the user as add-ons (in reality not all of the addons are installed there).
"/lost+found", this directory holds files that have been deleted or lost during a file operation. It means that we have an inode for those files, however we don't have a filename on disk for them (think about cases of kernel panic or an unplanned shutdown). It is handled by tools like fsck - more on that is a future writeup.

"/proc", it is a pseudo filesystem which enables retrieval of information about kernel data structures from user space using file operations, for example "ps" reads information for there to build the process list. Due to the fact it is a crucial part of Linux I am going to dedicate an entire writeup about it.

"/root", it's the default home directory of the root account.

"/run", it is used for runtime data like: running daemons, logged users and more. It should be erased/initialized every time on reboot/boot.

"/sbin", similar to "/bin" but contains system binaries like: lsmod, init (in Ubuntu by the way it is a link to systemd) and dhclient.

"/srv", contains information which is published by the system to the outside world using FTP/web server/other.

"/sys", also a pseudo filesystem (similar to /proc) which exports information about hardware devices, device drivers and kernel subsystems. It can also allow configuration of different subsystems (like tracing for ftrace). I will cover it separately in more detail in the near future.

"/tmp", the goal of the directory is to contain temporary files. Most of the time the content is not saved between reboots. Remember that there is also "/var/tmp".

"/usr", it is referred to by multiple names "User Programs" or "User System Resources". It has several subdirectories containing binaries, libs, doc files and also can contain source code. Historically, it was meant to be read-only and shared between FHS-compliant hosts (https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/usr.html). Due to the nature of its complexity today and the large amount of files it contains we will go over it also in a different writeup.

"/var", aka variable files. It contains files which by design are going to change during the normal operation of the system (think about spool files, logs and more). More on this directory in the future.

It is important to note that those are not all the directories and subdirectories included on a clean Linux installation, but the major ones I have decided to start with (more information will be shared in the future). See you soon ;-)

```
drwxr-xr-x   3 root root    4096 Sep 16 22:02 boot
drwxr-xr-x  19 root root    4120 Sep 18 05:47 dev
drwxr-xr-x 135 root root   12288 Sep 16 22:02 etc
drwxr-xr-x   3 root root    4096 Jul 29 07:13 home
lrwxrwxrwx   1 root root       7 Apr 19 06:02 lib -> usr/lib
lrwxrwxrwx   1 root root       9 Apr 19 06:02 lib32 -> usr/lib32
lrwxrwxrwx   1 root root       9 Apr 19 06:02 lib64 -> usr/lib64
lrwxrwxrwx   1 root root      10 Apr 19 06:02 libx32 -> usr/libx32
drwx------   2 root root   16384 Jul 29 07:10 lost+found
drwxr-xr-x   2 root root    4096 Apr 19 06:02 media
drwxr-xr-x   2 root root    4096 Apr 19 06:02 mnt
drwxr-xr-x   2 root root    4096 Apr 19 06:02 opt
dr-xr-xr-x 273 root root       0 Sep  9 12:31 proc
drwx------   6 root root    4096 Sep 12 18:40 root
drwxr-xr-x  29 root root     840 Sep 18 00:54 run
drwxr-xr-x   9 root root    4096 Apr 19 06:08 snap
drwxr-xr-x   2 root root    4096 Apr 19 06:02 srv
dr-xr-xr-x  13 root root       0 Sep  9 12:31 sys
drwxrwxrwt  19 root root    4096 Sep 18 07:20 tmp
drwxr-xr-x  14 root root    4096 Apr 19 06:02 usr
drwxr-xr-x  14 root root    4096 Apr 19 06:07 var
```

/boot/config-$(uname-r)

"/boot/config-$(uname-r)" is a text file that contains a configuration (feature/options) that the kernel was compiled with. The "uname -r" is replaced by the kernel release[21]. It is important to understand that the file is only needed for the compilation phase and not for loading the kernel, so it can be removed or even altered by a root user and therefore not reflect the specific configuration that was used.Overall, any time one of the following "make menuconfig"/make xconfig", "make localconfig", "make oldconfig", "make XXX_defconfig" or other "make XXXconfig" creates a ".config" file. This file is not erased (unless using "make mrproper"). Also, many distributions are copying that file to "/boot"[22].

The build system will read the configuration file and use it to generate the kernel by compiling the relevant source code. By using the configuration file we can customize the Linux kernel to your needs[23]. The configuration file is based on key values - as shown in the screenshot below[24]. Using the configuration we can enable/disable features like sound/networking/USB support as we can see with the "CONFIG_MMU=y" in the screenshot below[25]. Also, we can adjust a specific value of features like the "CONFIG_ARCH_MMAP_RND_BITS_MIN=28"[26].

Moreover, in case of kernel modules we can add/remove modules and decide if we want to compile them into the kernel itself or as a separate ".ko" file. In case the setting is "y" it means to compile inside the kernel, "m" means as a separate file and "n" means not to compile[27]. Thus, if "CONFIG_DRM_TTM=m" then the "TTM memory manager subsystem" is going to be compiled outside of the kernel[28]. If "ttm" is loaded it will be shown in the output of "lsmod"[29].


```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux/x86 4.15.0-117-generic Kernel Configuration
4 #
5 CONFIG_64BIT=y
6 CONFIG_X86_64=y
7 CONFIG_X86=y
8 CONFIG_INSTRUCTION_DECODER=y
9 CONFIG_OUTPUT_FORMAT="elf64-x86-64"
10 CONFIG_ARCH_DEFCONFIG="arch/x86/configs/x86_64_defconfig"
11 CONFIG_LOCKDEP_SUPPORT=y
12 CONFIG_STACKTRACE_SUPPORT=y
13 CONFIG_MMU=y
14 CONFIG_ARCH_MMAP_RND_BITS_MIN=28
15 CONFIG_ARCH_MMAP_RND_BITS_MAX=32
16 CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MIN=8
17 CONFIG_ARCH_MMAP_RND_COMPAT_BITS_MAX=16
18 CONFIG_NEED_DMA_MAP_STATE=y
19 CONFIG_NEED_SG_DMA_LENGTH=y
20 CONFIG_GENERIC_ISA_DMA=y
21 CONFIG_GENERIC_BUG=y
22 CONFIG_GENERIC_BUG_RELATIVE_POINTERS=y
23 CONFIG_GENERIC_HWEIGHT=y
24 CONFIG_ARCH_MAY_HAVE_PC_FDC=y
25 CONFIG_RWSEM_XCHGADD_ALGORITHM=y
26 CONFIG_GENERIC_CALIBRATE_DELAY=y
27 CONFIG_ARCH_HAS_CPU_RELAX=y
28 CONFIG_ARCH_HAS_CACHE_LINE_SIZE=y
29 CONFIG_ARCH_HAS_FILTER_PGPROT=y
30 CONFIG_HAVE_SETUP_PER_CPU_AREA=y
31 CONFIG_NEED_PER_CPU_EMBED_FIRST_CHUNK=y
"/boot/config-4.15.0-117-generic" 9621 fj --0%--
```

[21] https://linux.die.net/man/1/uname
[22] https://unix.stackexchange.com/questions/123026/where-kernel-configuration-file-is-stored
[23] https://linuxconfig.org/in-depth-howto-on-linux-kernel-configuration
[24] https://blog.csdn.net/weixin_43644245/article/details/121578858
[25] https://elixir.bootlin.com/linux/v6.4.11/source/arch/um/Kconfig#L36
[26] https://elixir.bootlin.com/linux/v6.4.11/source/arch/x86/Kconfig#L322
[27] https://stackoverflow.com/questions/14587251/understanding-boot-config-file
[28] https://github.com/torvalds/linux/blob/master/drivers/gpu/drm/ttm/ttm_module.c
[29] https://man7.org/linux/man-pages/man8/lsmod.8.html

/proc/config.gz

Since kernel version 2.6 the configuration options that were used to build the current running kernel are exposed using procfs in the following path "/proc/config.gz". The format of the content is the same as the .config file which is copied by different distribution to "/boot"[30].

Overall, as opposed to the ".config" file the data of "config.gz" is compressed. Due to that, if we want to view it content we can use zcat[31] or zgrep[32] which allow reading/searching inside compressed files. As shown in the screenshot below (taken from copy.sh).

Lastly, in order for "config.gz" to be supported and exported by "/proc" the kernel needs to be build with "CONFIG_IKCONFIG_PROC" enabled[33] - as also shown in the screenshot below. We can also go overt the creation of the "/proc" entry[34] and the function that returns the data when reading that entry[35].

```
root@localhost:~# file /proc/config.gz
/proc/config.gz: gzip compressed data, max compression, from Unix, original size modulo 2^32 265269
root@localhost:~# zcat /proc/config.gz | head -25
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 5.19.7-arch1 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="gcc (GCC) 12.2.0"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=120200
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=23900
CONFIG_LD_IS_BFD=y
CONFIG_LD_VERSION=23900
CONFIG_LLD_VERSION=0
CONFIG_CC_CAN_LINK=y
CONFIG_CC_CAN_LINK_STATIC=y
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_CC_HAS_ASM_GOTO_OUTPUT=y
CONFIG_CC_HAS_ASM_GOTO_TIED_OUTPUT=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_CC_HAS_NO_PROFILE_FN_ATTR=y
CONFIG_PAHOLE_VERSION=123
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_TABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

root@localhost:~# zcat /proc/config.gz | grep IKCONFIG_PROC
CONFIG_IKCONFIG_PROC=y
```

---

[30] https://medium.com/@boutnaru/the-linux-concept-journey-boot-config-uname-r-6a4dd16048c4
[31] https://linux.die.net/man/1/zcat
[32] https://linux.die.net/man/1/zgrep
[33] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L35
[34] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L60
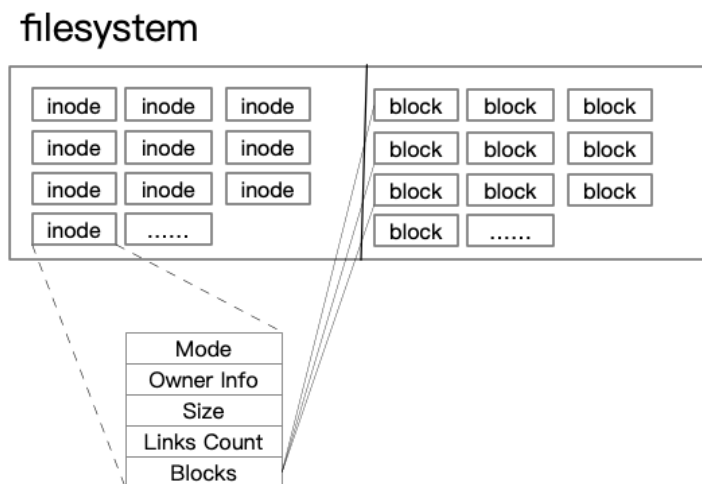[35] https://elixir.bootlin.com/linux/v6.5/source/kernel/configs.c#L41

What is an inode?

An inode (aka index node) is a data structure used by Unix/Linux like filesystems in order to describe a filesystem object. Such an object could be a file or a directory. Every inode stores pointers to the disk blocks locations of the object's data and metadata[36]. An illustration of that is shown below[37].

Overall, the metadata contained in an inode is: file type (regular file/directory/symbolic link/block special file/character special file/etc), permissions, owner id, group id, size, last accessed time, last modified time, change time and number of hard links[38].

By using inodes the filesystem tracks all files/directories saved on disk. Also, by using inodes we can read any specific byte in the data of a file very effectively. We can see the number of total inodes per mounted filesystem using the command "df -i"[39]. Also, we can see the inode of a file/directory and other metadata of the file using the command "ls -i"[40] or "stat"[41]. By the way, the "stat" command can use different syscalls (depending on the filesystem and the specific version) like "stat"[42], "lstat"[43] or "statx"[44].

Lastly, you can check out "struct inode" in the source code of the Linux kernel[45]. Not all the points/links are directly connected to the data blocks, however I will elaborate on that in a future writeup.

---

[36] https://www.bluematador.com/blog/what-is-an-inode-and-what-are-they-used-for
[37] https://www.sobyte.net/post/2022-05/linux-inode/
[38] https://www.stackscale.com/blog/inodes-linux/
[39] https://linux.die.net/man/1/df
[40] https://man7.org/linux/man-pages/man1/ls.1.html
[41] https://linux.die.net/man/1/stat
[42] https://linux.die.net/man/2/stat
[43] https://linux.die.net/man/2/lstat
[44] https://man7.org/linux/man-pages/man2/statx.2.html
[45] https://elixir.bootlin.com/linux/v6.4.2/source/include/linux/fs.h#L612

Why is removing a file not dependent on the file's permissions?

Something which is not always understood correctly by Linux users is the fact that removing a file is not dependent on the permissions of the file itself. As you can see in the screenshot below even if a user has full permission (read+write+execute) it can't remove a file. By the way, removing a file is done by using the "unlink" syscall[46] or the "unlinkat" syscall[47].

The reason for that is because the data that states a file belongs to a directory is saved as part of the directory itself. We can think about a directory as a "special file" whose data is the name and the inode[48] numbers of the files that are part of that specific directory.

Thus, if we add write permissions to the directory even if the user has no permissions to the file ("chmod 000") the file can be removed (from the directory) - as shown in the screenshot below.

---

[46] https://linux.die.net/man/2/unlink
[47] https://linux.die.net/man/2/unlinkat
[48] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940

VFS (Virtual File System)

VFS (Virtual File System, aka Virtual File Switch) is a software component of Linux which is responsible for the filesystem interface between the user-mode and kernel mode. Using it allows the kernel to provide an abstraction layer that makes implementation of different filesystems very easy[49].

Overall, VFS is masking the implementation details of a specific filesystem behind generic system calls (open/read/write/close/etc), which are mostly exposed to user-mode application by some wrappers in libc - as shown in the diagram below[50].

Moreover, we can say that the main goal of VFS is to allow user-mode applications to access different filesystems (think about NTFS, FAT, etc.) in the same way. There are four main objects in VFS: superblock, dentries, inodes and files[51].

Thus, "inode"[52] is what the kernel uses to keep track of files. Because a file can have several names there are "dentries" ("directory entries") which represent pathnames. Also, due to the fact a couple of processes can have the same file opened (for read/write) there is a "file" structure that holds the information for each one (such as the cursor position). The "superblock" structure holds data which is needed for performing actions on the filesystem - more details about all of those and more (like mounting) are going to be published in the near future.

Lastly, there are also other relevant data structures that I will post on in the near future ("filesystem", "vfsmount", "nameidata" and "address_space").

[49] https://www.kernel.org/doc/html/next/filesystems/vfs.html
[50] https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour
[51] https://www.win.tue.nl/~aeb/linux/lk/lk-8.html
[52] https://medium.com/@boutnaru/linux-what-is-an-inode-7ba47a519940
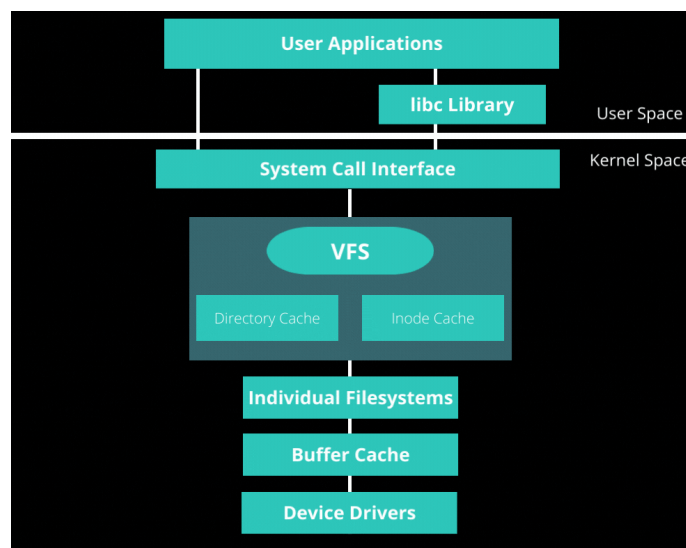
tmpfs (Temporary Filesystem)

"tmpfs" is a filesystem that saves all of its files in virtual memory. By using it none of the files created on it are saved to the system's hard drive. Thus, if we unmount a tmpfs mounting point every file which is stored there is lost. tmpfs holds all of the data into the kernel internal caches[53]. By the way, it used to be called "shm fs"[54].

Moreover, tmpfs is able to swap space if needed (it can also leverage "Transparent Huge Pages"), it will fill up until it reaches the maximum limit of the filesystem - as shown in the screenshot below. tmpfs supports both POSIX ACLs and extended attributes[55]. Overall, if we want to use tmpfs we can use the following command: "mount -t tmpfs tmpfs [LOCATION]". We can also set a size using "-o size=[REQUESTED_SIZE]" - as shown in the screenshot below.

Lastly, there are different directories which are based on "tmpfs" like: "/run" and "/dev/shm" (more on them in future writeups). To add support for "tmpfs" we should enable "CONFIG_TMPFS" when building the Linux kernel[56]. We can see the implementation as part of the Linux's kernel source code[57].

[53] https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html
[54] https://cateee.net/lkddb/web-lkddb/TMPFS.html
[55] https://man7.org/linux/man-pages/man5/tmpfs.5.html
[56] https://cateee.net/lkddb/web-lkddb/TMPFS.html
[57] https://elixir.bootlin.com/linux/v6.6-rc1/source/mm/shmem.c#L133

ramfs (Random Access Memory Filesystem)

"ramfs" is a filesystem that exports the Linux caching mechanism (page cache/dentry cache) as a dynamically resizable RAM based filesystem. The data is saved in RAM only and there is no backing store for it[58].

Thus, if we unmount a "ramfs" mounting point every file which is stored there is lost - as shown in the screenshot below. By the way, the trick is that files written to "ramfs" allocate dentries and page cache as usual, but because they are not written they are never marked as being available for freeing[59].

Moreover, with "ramfs" we can keep on writing until we fill-up the entire physical memory. Due to that, it is recommended that only root users will be able to write to a mounting point which is based on "ramfs". The differences between "ramfs" and "tmpfs"[60] is that "tmpfs" is limited in size and can also be swapped[61].

Lastly, we can go over the implementation of "ramfs" as part of Linux's kernel source code[62]. There are two implementations, one in case of an MMU[63] and one in case there is no MMU[64]. A good example for using "ramfs" is "initramfs".

[58] https://docs.kernel.org/filesystems/ramfs-rootfs-initramfs.html
[59] https://lwn.net/Articles/157676/
[60] https://medium.com/@boutnaru/the-linux-concept-journey-tmpfs-temporary-filesystem-886b61a545a0
[61] https://wiki.debian.org/ramfs
[62] https://elixir.bootlin.com/linux/v6.5.3/source/fs/ramfs
[63] https://elixir.bootlin.com/linux/v6.5.3/source/fs/ramfs/file-mmu.c
[64] https://elixir.bootlin.com/linux/v6.5.3/source/fs/ramfs/file-nommu.c
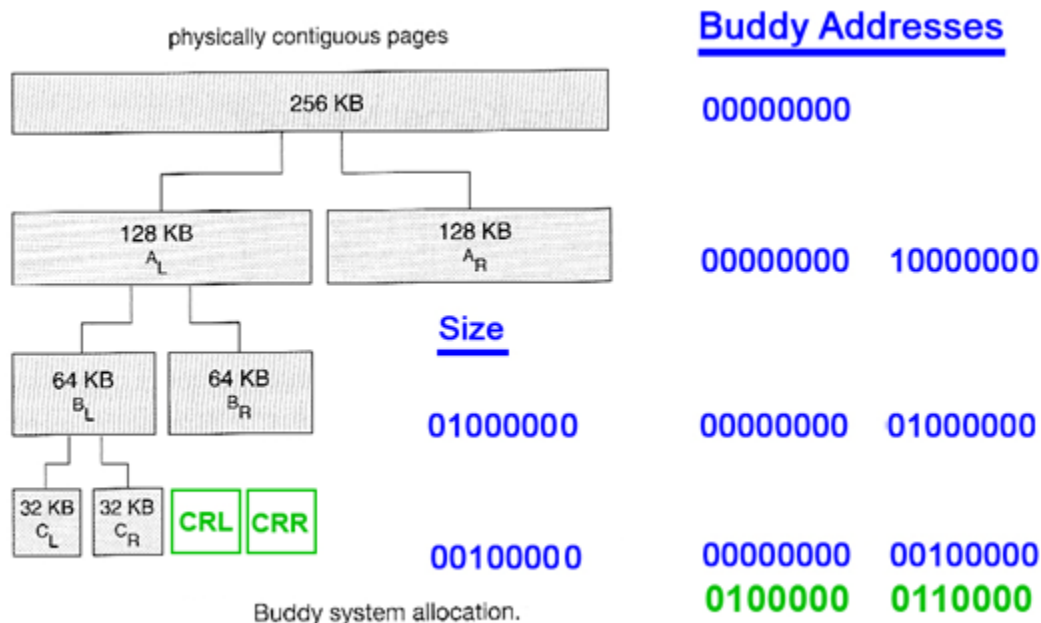
Buddy Memory Allocation

Basically, "buddy system" is a memory allocation algorithm. It works by dividing memory into blocks of a fixed size. Each block of allocated memory is a power of two in size. Every memory block in this system has an "order" (an integer ranging from 0 to a specified upper limit). The size of a block of order n is proportional to 2^n, so that the blocks are exactly twice the size of blocks that are one order lower[65]

Thus, when a request for memory is made, the algorithm finds the smallest available block of memory (that is sufficient to satisfy the request). If the block is larger than the requested size, it is split into two smaller blocks of equal size (aka "buddies"). One of them is marked as free and the second one as allocated. The algorithm then continues recursively until it finds the exact size of the requested memory or a block that is the smallest possible size[66].

Moreover, the advantages of such a system is that it is easy to implement and can handle a wide range of memory sizes. The disadvantages are that it can lead to memory fragmentation and is inefficient for allocating small amounts of memory. By the way, when the used "buddy" is freed, if it's also free they can be merged together - a diagram of such relationship is shown below[67]. Lastly, the Linux implementation of the "buddy system" is a little different than what is described here, I am going to elaborate about it in a detected writeup.

Buddy system allocation.

---

[65] https://en.wikipedia.org/wiki/Buddy_memory_allocation
[66] https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/
[67] https://www.expertsmind.com/questions/describe-the-buddy-system-of-memory-allocation-3019462.aspx